User Guide

Amazon Neptune

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon Neptune: User Guide

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Services or capabilities described in AWS documentation might vary by Region. To see the differences applicable to the AWS European Sovereign Cloud Region, see the AWS European Sovereign Cloud User Guide.

Table of Contents

What is Neptune? ... 1
Latest Updates ... 4
Getting started with Neptune ... 66
What's a graph database? ... 66
Why use graphs? ... 67
Graph database applications ... 68
Graph query languages ... 70
Query examples ... 71
Online Neptune course ... 73
Digging deeper ... 73
Quick start using CloudShell ... 74
Prerequisites ... 74
Step 1: Connect to your Neptune cluster ... 74
Step 2: Insert and query data ... 75
Next steps ... 76
Troubleshooting notebooks ... 76
Widget state desynchronization ... 76
JupyterLab 4.x initial loading screen delay ... 77
Graph Explorer access issues ... 77
General tips ... 77
Neptune setup ... 78
DB instance types ... 78
Instance resource allocation ... 79
`t3` and `t4g` ... 80
`r4` instances ... 80
`r5` instances ... 81
`r5d` instances ... 81
`r6g` instances ... 81
`r6i` instances ... 82
`x2g` instances ... 82
`x2iezn` instances ... 82
`x2iedn` instances ... 83
`r8g` instances ... 83
`r7g` instances ... 84

- `r7i` instances ... 84
- `serverless` instances ... 85
- Storage types ... 85
  - I/O–Optimized storage ... 86
- Create Neptune cluster ... 87
  - Prerequisites ... 88
  - Create a cluster ... 93
- Configure the VPC ... 96
  - Add subnets ... 97
  - Configure VPC ... 97
  - Create a security group ... 98
  - DNS in Your VPC ... 99
  - Dual-stack mode ... 100
- Connect to a cluster ... 102
  - Network connectivity ... 102
  - Verify connection ... 103
  - In the VPC ... 105
  - From a different VPC ... 107
  - Over a private network ... 107
  - Public Endpoints ... 108
- Securing access to Neptune ... 118
  - IAM policies ... 118
  - VPC security groups ... 119
  - IAM authentication ... 119
- Accessing graph data ... 120
  - Tools ... 120
  - Query languages ... 121
  - Using command-line tools ... 122
  - Using Gremlin ... 126
  - Using openCypher ... 132
  - Using SPARQL ... 132
- Loading data ... 133
- Monitoring Neptune cluster ... 134
- Troubleshooting Neptune cluster ... 134

Neptune Serverless ... 135

- Serverless use cases ... 135

Constraints ... 136
Capacity scaling ... 137
Setting the minimum ... 139
Setting the maximum ... 139
Estimate capacity settings ... 140
Additional configuration ... 141
Mixed configuration ... 141
Setting promotion tiers ... 142
Reader-writer alignment ... 143
Avoid very large timeout values ... 143
Optimizing your configuration ... 144
Using Serverless ... 144
Creating a serverless cluster ... 144
Converting to Serverless ... 145
Modifying the capacity range ... 146
Changing an instance to provisioned ... 147
Client configuration ... 147
Monitoring ... 147
Neptune overview ... 149
Standards Compliance ... 150
Gremlin standards compliance ... 150
SPARQL standards compliance ... 168
openCypher specification compliance ... 174
Graph Data Model ... 189
The dictionary ... 189
Indexing Strategy ... 190
Graph schema and metadata ... 193
Graph summary API ... 193
Property graph schema ... 208
DFE statistics ... 221
Lookup cache ... 239
Use cases for the lookup cache ... 239
Using the cache ... 240
Transaction Semantics ... 242
Isolation Levels ... 242
Neptune Isolation Levels ... 243

Transaction examples ... 250
Clusters and Instances ... 255
The primary DB instance ... 255
Read-replica instances ... 255
Sizing instances ... 256
Monitoring instances ... 257
Storage, reliability and availability ... 258
Endpoint Connections ... 259
Cluster endpoints ... 259
Reader endpoints ... 259
Instance endpoints ... 261
Custom endpoints ... 261
Endpoint considerations ... 262
Working with custom endpoints ... 263
Lab Mode ... 267
Using Lab Mode ... 267
OSGP index ... 268
Dictionary garbage collection ... 269
Transaction Semantics ... 269
Extended datetime support ... 269
StrictTimeoutValidation ... 270
AccurateQRCMemoryEstimation ... 270
UndoLogPurgeConfig ... 271
Neptune DFE engine ... 272
Controlling DFE use ... 272
Queries executed by DFE ... 273
JDBC connectivity ... 275
Getting started ... 275
Using Tableau ... 276
Troubleshooting ... 278
Neptune engine updates ... 280
Exceptions and Retries ... 281
Why retry logic matters ... 281
The risks of indiscriminate retries ... 281
Developing a practical retry strategy ... 281
Example scenarios ... 282

Monitoring and observability ... 284
Security ... 286
Amazon Neptune operating system upgrades ... 287
Minimizing downtime for OS upgrades that require a restart ... 287
Applying OS upgrades to your Neptune DB instance ... 287
Protecting data in Neptune databases ... 290
Securing Neptune databases with Amazon VPC ... 291
Encrypting connects to Neptune databases ... 292
Encrypting data at Rest ... 293
Authenticating with IAM ... 297
Different roles ... 298
Using Identities ... 298
Enabling IAM ... 299
Connecting with IAM authentication ... 300
Prerequisites ... 302
Connecting from the command line ... 303
Connecting from Gremlin Console ... 305
Connecting using Gremlin Java ... 307
Connecting with SPARQL Java ... 312
Conncting with SPARQL and Node.js ... 315
Connecting with Python ... 318
Gremlin Python ... 322
Gremlin Javascript ... 323
Gremlin Go ... 325
Gremlin .NET ... 330
Using IAM policies ... 333
Identity-Based Policies ... 333
Service Control Policies (SCP) ... 333
Neptune Console Access ... 334
Attaching a Policy ... 334
Types of IAM policies ... 335
Using condition Keys ... 335
IAM feature support ... 336
IAM Policy Limitations ... 337
Using managed policies ... 337
Using condition Keys ... 353

IAM administrative policy statements ... 354
Custom data-access policy statements ... 378
Service-Linked Roles ... 409
Role Permissions ... 409
Creating a Service-Linked Role ... 411
Editing a Service-Linked Role ... 411
Deleting a Service-Linked Role ... 412
Temporary credentials ... 414
Get Credentials with the AWS CLI ... 415
Setting Up Lambda ... 418
Set Up Amazon EC2 ... 420
Logging and Monitoring ... 422
Interface VPC endpoints (AWS PrivateLink) ... 422
Compliance considerations ... 423
Building resilient deployments ... 423
Migrating to Neptune ... 424
Migrating from Neo4j ... 425
General information ... 425
Preparing for migration ... 428
Provisioning infrastructure ... 434
Data migration ... 437
Application migration ... 443
Neptune compatibility ... 447
Cypher rewrites ... 452
Migration resources ... 460
Migrating from TinkerPop ... 461
Migrating from RDF ... 462
Using AWS DMS to migrate ... 463
Migrating from Blazegraph ... 465
Neptune compatibility ... 465
Provisioning infrastructure ... 466
Exporting data ... 467
Create an Amazon S3 bucket ... 469
Loading data ... 470
Neptune Bulk Loader ... 470
IAM Role and Amazon S3 Access ... 472

Data Formats ... 482
Loading Example ... 497
Optimizing a bulk load ... 506
Fast bulk load ... 508
Loader Reference ... 516
Load data using DMS ... 554
GraphMappingConfig ... 555
Replicating to Neptune ... 559
Load data using queries ... 564
openCypher: neptune.read() ... 565
SPARQL: LOAD and UNLOAD ... 565
Gremlin: io() step ... 565
Storage ... 567
I/O–Optimized storage ... 567
Allocation ... 567
Storage billing ... 568
Storage usage breakdown ... 568
Prerequisites ... 569
Request ... 569
Response ... 571
Errors ... 575
Storage best practices ... 575
Reliability and high availability ... 576
Storage cleanup ... 577
Dictionary garbage collection ... 577
Online index rebuild ... 579
Inlined server-generated edge ID ... 599
Querying ... 602
Query queuing ... 603
Finding how many queries are in the queue ... 603
Query timeouts ... 603
Query plan cache ... 603
How to force enable or disable query plan cache ... 604
How to determine if a plan is cached or not ... 606
Eviction ... 608
Conditions causing the plan not to be cached ... 608

Custom queryId ... 609
Using the HTTP Header ... 609
Using a SPARQL Query Hint ... 609
Using queryId to Check Status ... 610
Gremlin ... 610
Installing the Gremlin console ... 613
HTTPS REST ... 618
Java ... 622
Python ... 635
.NET ... 637
Node.js ... 640
Go ... 643
AWS SDK ... 646
Query hints ... 650
Query status ... 660
Query cancellation ... 665
Gremlin script-based sessions ... 667
Gremlin transactions ... 669
Streaming results ... 673
Using the Gremlin API ... 678
Caching query results ... 678
Efficient upserts from 3.6.x forward ... 686
Efficient upserts before 3.6.x ... 693
Gremlin `explain` ... 707
Gremlin and DFE ... 760
openCypher ... 764
Gremlin vs. openCypher ... 765
Using openCypher ... 765
Status endpoint ... 766
HTTPS endpoint ... 774
AWS SDK ... 782
Using the Bolt protocol ... 786
Parameterized examples ... 811
Data model ... 813
openCypher `explain` ... 814
Transactions ... 841

Streaming results ... 849
Query hints ... 855
Restrictions ... 862
Exceptions ... 862
Extensions ... 867
Spatial Data ... 896
SPARQL ... 909
RDF4J Console ... 910
RDF4J Workbench ... 913
Java ... 915
HTTP API ... 919
Query hints ... 932
DESCRIBE and the default graph ... 949
Query status ... 951
Query cancellation ... 954
Graph store protocol ... 955
SPARQL `explain` ... 957
SPARQL `SERVICE` Extension ... 989
Exporting data ... 992
Native export ... 992
How it works ... 993
Prerequisites ... 993
Recommendations ... 993
IAM permissions ... 994
Export endpoint ... 996
Request syntax ... 996
Status endpoint ... 998
List exports ... 1000
Cancel export ... 1001
Output format ... 1001
Errors ... 1002
Examples ... 1002
Limitations ... 1003
Exporting Gremlin query results to Amazon S3 ... 1004
Examples ... 1005
Prerequisites ... 1005

Response format ... 1007
Granting access for Gremlin Amazon S3 export feature ... 1010
External neptune-export tool ... 1012
Neptune-Export service ... 1013
neptune-export utility ... 1021
Exported files ... 1024
Export parameters ... 1025
Troubleshooting ... 1046
Managing Neptune ... 1050
Neptune Blue/Green solution ... 1051
Neptune Blue/Green prerequisites ... 1052
Use CloudFormation to run the solution ... 1053
Monitoring progress ... 1054
Cutting over to the updated cluster ... 1056
Cleanup ... 1057
Best practices ... 1057
Troubleshooting ... 1058
IAM user permissions ... 1059
Service-linked role policy ... 1059
Create a new IAM user ... 1060
Parameter groups ... 1062
Editing a Parameter Group ... 1064
Creating a parameter group ... 1065
Parameters ... 1066
neptune_enable_audit_log ... 1067
neptune_enable_slow_query_log ... 1067
neptune_slow_query_log_threshold ... 1067
neptune_lab_mode ... 1068
neptune_query_timeout ... 1068
neptune_streams ... 1069
neptune_streams_expiry_days ... 1069
neptune_lookup_cache ... 1070
neptune_autoscaling_config ... 1070
neptune_ml_iam_role ... 1071
neptune_ml_endpoint ... 1071
neptune_enable_inline_server_generated_edge_id ... 1071

neptune_dfe_query_engine ... 1071
neptune_query_timeout ... 1072
neptune_result_cache ... 1073
neptune_enforce_ssl ... 1073
Launch using the console ... 1074
Stopping and starting a cluster ... 1080
Stopping and starting overview ... 1080
Stopping a cluster ... 1081
Starting a DB cluster ... 1082
Fast reset API ... 1084
Using IAM-Auth ... 1087
Common errors ... 1087
Adding reader instances ... 1089
Creating a reader instance ... 1090
Modifying a DB Cluster ... 1092
Modify an Instance ... 1093
Performance and Scaling ... 1095
Storage Scaling ... 1095
Instance Scaling; ... 1095
Read Scaling ... 1095
Auto-scaling ... 1096
Auto-scaling and serverless ... 1098
Enabling auto-scaling ... 1098
Remove auto-scaling ... 1102
Cluster maintenance ... 1103
Version numbers ... 1103
Release types ... 1104
Engine version life-spans ... 1106
Managing engine updates ... 1107
Upgrade process ... 1112
Upgrading to 1.2.0.0 or above ... 1114
Update via CloudFormation ... 1116
1.2.0.1 to 1.2.0.2 ... 1117
1.1.1.0 to 1.2.0.2, default ... 1120
1.1.1.0 to 1.2.0.2, custom ... 1121
1.1.1.0 to 1.2.0.2, mixed ... 1124

Cloning a DB Cluster ... 1128
Limitations ... 1130
Copy-on-Write Protocol ... 1130
Deleting a Source Database ... 1132
Managing Instances ... 1133
T3 Burstable Instances ... 1134
Modifying an Instance ... 1136
Renaming a Neptune DB Instance ... 1141
Rebooting a DB instance ... 1143
Deleting a DB Instance ... 1145
Neptune streams ... 1148
Using Streams ... 1150
Enabling Streams ... 1150
Disabling Streams ... 1151
Calling the Streams API ... 1151
Streams Response ... 1153
Reading stream data ... 1156
Streams Exceptions ... 1158
Streams Record Formats ... 1159
PG_JSON ... 1160
RDF-NQUADS ... 1163
Streams Examples ... 1163
AT_SEQUENCE_NUMBER Examples ... 1164
AFTER_SEQUENCE_NUMBER Example ... 1165
TRIM_HORIZON Example ... 1166
LATEST Example ... 1166
Compression Example ... 1168
Neptune-to-Neptune Replication Setup ... 1169
Choose an CloudFormation template ... 1169
Add stack details ... 1171
Run the Template ... 1175
Updating the stream poller ... 1175
Neptune full text search ... 1177
Full-text search setup ... 1179
CloudFormation template ... 1180
Existing databases ... 1186

Updating the poller ... 1188
Stopping and starting the poller ... 1189
OpenSearch Serverless ... 1190
Changes required for poller AWS Lambda function ... 1190
Changes required for bulk import utility ... 1190
Changes needed to the IAM role used to query Neptune ... 1191
Querying with fine-grained access control ... 1191
Use of Lucene syntax ... 1192
Neptune Full-text search data model ... 1193
SPARQL sample document ... 1194
Gremlin sample document ... 1196
Full-text search parameters ... 1197
Non-string indexing ... 1202
Updating an existing stack ... 1203
Excluding fields ... 1204
Datatype mappings ... 1207
Datatype validation ... 1208
Sample queries ... 1214
Full-text-search query execution ... 1216
Sample SPARQL full-text search queries ... 1218
match Query ... 1218
prefix ... 1218
fuzzy ... 1219
term ... 1219
query_string ... 1219
simple_query_string ... 1220
sort by string field ... 1220
sort by non-string field ... 1220
sort by ID ... 1221
sort by label ... 1221
sort by doc_type ... 1222
Lucene syntax ... 1222
Sample Gremlin full-text search queries ... 1223
Basic `match` ... 1223
`match` ... 1224
`fuzzy` ... 1224

query_string fuzzy 1224
query_string regex 1225
Hybrid query 1225
Full-text search example 1225
query_string, '+' and '-' 1226
query_string, AND and OR 1227
term 1228
prefix 1228
Lucene syntax 1228
Modern TinkerPop graph 1230
Sort by string field 1230
Sort by non-string field 1230
Sort by ID field 1231
Sort by label field 1231
Sort by document_type field 1231
Troubleshooting and metrics 1231
Reads troubleshooting 1232
Writes troubleshooting 1233
Out-of-sync issues 1233
AWS Lambda functions 1235
Gremlin WebSocket connections 1235
Gremlin Lambda recommendations 1236
Write-request recommendations 1237
Read-request recommendations 1237
Cold-start latency 1238
Creating a Lambda Function 1239
Lambda function examples 1242
Java example 1242
JavaScript example 1248
Python example 1252
Neptune machine learning 1257
Neptune ML capabilities 1257
Neptune ML setup 1259
Setup using CloudFormation 1260
Manual setup 1264
Using the AWS CLI 1273

Using Neptune ML 1277
Starting workflow 1277
Handling evolving data 1279
Updating the model artifacts 1280
Custom model workflow 1281
Instance selection 1282
For data processing 1282
For model training and model transform 1282
For an inference endpoint 1282
Data export 1284
Neptune-Export examples 1284
Params settings 1287
AdditionalParams 1288
targets 1291
features 1297
Examples 1306
Data processing 1321
Managing data processing 1321
Updated processing 1323
Feature encoding 1326
Editing a training data file 1334
Model training 1345
Models and training 1352
Customizing hyperparameters 1355
Training best practices 1368
Model transform 1372
Incremental inference 1372
Model transform for any job 1374
Model artifacts 1376
Artifacts for different tasks 1376
Generating new artifacts 1376
Custom models 1379
Custom model overview 1380
Custom model development 1387
Inference endpoint 1392
Managing inference endpoints 1392

Inference queries ... 1396
Gremlin inference queries ... 1397
SPARQL inference queries ... 1421
Neptune ML API ... 1428
The dataprocessing command ... 1429
The modeltraining command ... 1440
The modeltransform command ... 1455
The endpoints command ... 1469
Exceptions ... 1482
Limits ... 1483
SageMaker AI limits ... 1484
Monitoring Neptune ... 1485
Instance Status ... 1486
Sample output ... 1488
Using CloudWatch ... 1489
Using the Console ... 1490
Using the AWS CLI ... 1490
Using the CloudWatch API ... 1491
Monitor instance performance ... 1492
Neptune Metrics ... 1493
Neptune Dimensions ... 1521
Audit Logs with Neptune ... 1522
Enable Audit Logs ... 1522
Viewing Audit Logs ... 1522
Audit Log Details ... 1522
Neptune CloudWatch Logs ... 1524
Publish Logs to CloudWatch Logs (Console) ... 1524
Publish audit logs to CloudWatch Logs (CLI) ... 1525
Publish slow-query logs to CloudWatch Logs (CLI) ... 1525
Monitoring Log Events ... 1526
Slow-query logs ... 1526
Viewing logs in the console ... 1527
Slow-query log files ... 1527
`info` mode attributes ... 1528
`debug` mode attributes ... 1531
Output example ... 1533

Logging Neptune API Calls with AWS CloudTrail 1534
Neptune Information in CloudTrail 1535
Understanding log file entries 1536
Events 1538
Categories and messages 1539
Viewing events 1552
Subscribing to events 1554
Manage subscriptions 1554
Tagging Neptune resources 1555
Tagging overview 1556
Tagging in the console 1558
Tagging with the CLI 1559
Tagging with the API 1560
Working with ARNs 1561
Backing up and restoring 1566
Backups 1566
Fault Tolerance 1567
Backup metrics 1568
Restoring Data 1569
Backup window 1570
Creating a Snapshot 1572
Using the Console 1572
Restoring from a Snapshot 1573
Important restore considerations 1573
Restoring 1575
Copying a Snapshot 1576
Limitations 1576
Snapshot Copy Retention 1576
Encryption 1577
Cross-Region Snapshot Copying 1577
Copying a Snapshot on the Console 1577
Copying a Snapshot with the AWS CLI 1578
Sharing a Snapshot 1580
Encrypted Snapshots 1581
Sharing 1584
Deleting a Snapshot 1586

Using the Console ... 1586
Using the AWS CLI ... 1586
Using the Neptune API ... 1586
Best practices ... 1587
Basic operational guidelines ... 1589
Security ... 1590
Avoid different instance sizes ... 1591
Avoid bulk-load restarts ... 1591
If you have many predicates ... 1592
Avoid long-running transactions ... 1592
Using Metrics ... 1592
Tuning Queries ... 1593
Load balancing ... 1594
Use a temporary instance ... 1594
Resizing an instance ... 1595
Task interrupted error ... 1595
Gremlin (General) ... 1596
Heartbeat Configuration ... 1597
Optimize upsert queries ... 1599
Execution differences ... 1599
Multithreaded Writes ... 1601
Pruning Records ... 1602
`datetime( )` ... 1603
Native Date and Time ... 1603
Gremlin (Java client) ... 1606
Re-use the client object ... 1606
Separate Clients for Reading and Writing ... 1607
Multiple replica endpoints ... 1607
Close the client when finished ... 1607
New connection after failover ... 1608
Set maxInProcessPerConnection = maxSimultaneousUsagePerConnection ... 1608
Send queries as bytecode ... 1608
Completely consume query results ... 1610
Bulk add vertices and edges ... 1610
Disable JVM DNS caching ... 1611
Per-query timeouts ... 1611

Handling a `TimeoutException` ... 1613
openCypher and Bolt ... 1614
Reconnect after failover ... 1615
Re-use the Driver object ... 1616
Lambda connection handling ... 1616
Prefer directed edges ... 1616
No concurrent transaction queries ... 1617
Close driver objects ... 1617
Use explicit transaction modes ... 1618
Retry logic ... 1620
Using the SET clause ... 1624
Use parameterized queries ... 1625
UNWIND clase ... 1626
Variable-Length Path (VLP) expressions ... 1627
Use granular relationship names ... 1628
Specify edge labels ... 1629
Avoid the WITH clause ... 1629
Restrictive filters ... 1630
Explicitly check properties ... 1631
Avoid named path ... 1631
Avoid COLLECT(DISTINCT()) ... 1632
Prefer the properties function ... 1632
Static computations ... 1633
Batching inputs ... 1634
Prefer custom IDs ... 1634
Avoid ~id computations ... 1635
Updating/Merging multiple nodes ... 1636
SPARQL ... 1637
Query All Named Graphs ... 1637
Specify a Named Graph to Load ... 1638
FILTER vs. VALUES ... 1638
Amazon ECS and Amazon EKS ... 1640
Connection lifecycle ... 1640
Host replacement ... 1642
Networking and IAM ... 1643
AWS CLI and SDKs ... 1645

Neptune Limits **1648**
Regions 1648
China regions 1649
Cluster volume size 1649
Instance sizes 1649
Per account 1649
VPC Required 1650
SSL Required 1650
Availability Zones and Subnet Groups 1650
HTTP request payload 1650
Gremlin 1651
No null characters 1651
SPARQL UPDATE LOAD 1651
Authentication and Access 1651
WebSockets Limits 1652
Properties and labels 1655
Bulk-loading 1655
Neptune integrations **1657**
Tools and utilities **1659**
GraphQL utility 1659
Installation and setup 1660
Using existing data 1661
Using a schema with no directives 1662
Working with directives 1667
Command-line arguments 1672
Nodestream 1676
Neptune Errors **1678**
Insufficient DB instances available 1678
Engine Error Codes 1678
Error Format 1679
Query Errors 1679
IAM Errors 1684
API Errors 1686
Loader Errors 1689
Engine releases **1691**
Engine version life-span planning 1694

Release: 1.5.0.0 (2026-10-13) 1696
Features 1696
Improvements 1696
Defects Fixed 1697
Supported Query-Language Versions 1697
Upgrade paths 1697
Upgrading 1698
Release: 1.4.8.1 (2026-10-06) 1700
Improvements 1700
Defects fixed 1700
Supported Query-Language Versions 1701
Upgrade paths 1701
Upgrading 1701
Release: 1.4.8.0 (2026-07-27) 1703
Features 1704
Improvements 1704
Defects Fixed 1704
Supported Query-Language Versions 1704
Upgrade paths 1705
Upgrading 1705
Release: 1.4.7.1 (2026-09-18) 1707
Defects fixed 1707
Supported Query-Language Versions 1708
Upgrade paths 1708
Upgrading 1708
Release: 1.4.7.0 (2026-03-03) 1711
Features 1711
Improvements 1711
Defects Fixed 1711
Supported Query-Language Versions 1712
Upgrade paths 1712
Upgrading 1712
Release: 1.4.6.3 (2025-12-18) 1714
Defects fixed 1714
Supported Query-Language Versions 1715
Upgrade paths 1715

Upgrading ... 1715
Release: 1.4.6.2 (2025-11-18) ... 1717
Defects fixed ... 1718
Supported Query-Language Versions ... 1718
Upgrade paths ... 1718
Upgrading ... 1719
Release: 1.4.6.1 (2025-09-18) ... 1721
Defects fixed ... 1721
Supported Query-Language Versions ... 1721
Upgrade paths ... 1722
Upgrading ... 1722
Release: 1.4.6.0 (2025-09-02) ... 1724
New features ... 1724
Improvements ... 1724
Defects fixed ... 1725
Supported Query-Language Versions ... 1725
Upgrade paths ... 1726
Upgrading ... 1726
Release: 1.4.5.1 (2025-06-30) ... 1728
Defects fixed ... 1728
Supported Query-Language Versions ... 1728
Upgrade paths ... 1729
Upgrading ... 1729
Release: 1.4.5.0 (2025-04-09) ... 1731
New features ... 1731
Improvements ... 1732
Defects fixed ... 1733
Supported Query-Language Versions ... 1734
Upgrade paths ... 1734
Upgrading ... 1734
Release: 1.4.4.0 (2025-02-24) ... 1736
Defects fixed ... 1737
Supported Query-Language Versions ... 1737
Upgrade paths ... 1737
Upgrading ... 1737
Release: 1.4.3.0 (2025-01-21) ... 1740

New features ... 1740
Improvements ... 1741
Defects fixed ... 1742
Supported Query-Language Versions ... 1742
Upgrade paths ... 1742
Upgrading ... 1742
Release: 1.4.2.0 (2024-12-19) ... 1744
Defects fixed ... 1745
Supported Query-Language Versions ... 1745
Upgrade paths ... 1746
Upgrading ... 1746
Release: 1.4.1.0 (2024-11-21) ... 1748
New features ... 1748
Improvements ... 1749
Defects fixed ... 1750
Supported Query-Language Versions ... 1751
Upgrade paths ... 1751
Upgrading ... 1751
Release: 1.4.0.0 (2024-11-06) ... 1753
New features ... 1754
Improvements ... 1754
Defects fixed ... 1756
Supported Query-Language Versions ... 1756
Upgrade paths ... 1757
Upgrading ... 1757
Release: 1.3.4.0 (2024-10-01) ... 1759
Improvements ... 1760
Defects fixed ... 1760
Supported Query-Language Versions ... 1761
Upgrade paths ... 1762
Upgrading ... 1762
Release: 1.3.3.0 (2024-08-05) ... 1764
Defects fixed ... 1764
Mitigating issues in release 1.3.3.0 ... 1765
Supported Query-Language Versions ... 1766
Upgrade paths ... 1766

Upgrading ... 1766
Release: 1.3.2.1 (2024-06-20) ... 1768
Defects fixed ... 1769
Changes in 1.3.2.1 carried over from 1.3.2.0 ... 1770
Upgrade paths ... 1775
Upgrading ... 1775
Release: 1.3.2.0 (2024-06-10) ... 1777
Improvements ... 1778
Defects fixed ... 1779
Mitigating issues in release 1.3.2.0 ... 1781
Supported Query-Language Versions ... 1783
Upgrade paths ... 1784
Upgrading ... 1784
Release: 1.3.1.0 (2024-03-06) ... 1786
Improvements ... 1786
Defects fixed ... 1787
Supported Query-Language Versions ... 1787
Upgrade paths ... 1788
Upgrading ... 1788
Release: 1.3.0.0 (2023-11-15) ... 1790
New Features ... 1790
Improvements ... 1791
Defects Fixed ... 1793
Supported Query-Language Versions ... 1794
Upgrade Paths ... 1794
Upgrading ... 1794
Release: 1.2.1.2 (2024-08-05) ... 1796
Defects Fixed ... 1798
Supported Query-Language Versions ... 1798
Upgrade Paths ... 1798
Upgrading ... 1798
Release: 1.2.1.1 (2024-03-11) ... 1800
Improvements ... 1801
Defects Fixed ... 1802
Supported Query-Language Versions ... 1802
Upgrade Paths ... 1803

Upgrading ... 1803
Release: 1.2.1.0 (2023-03-08) ... 1805
Patch Releases ... 1806
New Features ... 1806
Improvements ... 1808
Defects Fixed ... 1808
Supported Query-Language Versions ... 1809
Upgrade Paths ... 1810
Upgrading ... 1810
Release: 1.2.1.0.R7 (2023-10-06) ... 1812
Release: 1.2.1.0.R6 (2023-09-12) ... 1815
Release: 1.2.1.0.R5 (2023-09-02) ... 1819
Release: 1.2.1.0.R4 (2023-08-10) ... 1822
Release: 1.2.1.0.R3 (2023-06-13) ... 1826
Release: 1.2.1.0.R2 (2023-05-02) ... 1832
Release: 1.2.0.2 (2022-11-20) ... 1836
Patch Releases ... 1837
New Features ... 1837
Improvements ... 1838
Supported Query-Language Versions ... 1838
Upgrade Paths ... 1838
Upgrading ... 1838
Release: 1.2.0.2.R6 (2023-09-12) ... 1840
Release: 1.2.0.2.R5 (2023-08-16) ... 1843
Release: 1.2.0.2.R4 (2023-05-08) ... 1847
Release: 1.2.0.2.R3 (2023-03-27) ... 1851
Release: 1.2.0.2.R2 (2022-12-15) ... 1855
Release: 1.2.0.1 (2022-10-26) ... 1859
Patch Releases ... 1860
New Features ... 1860
Improvements ... 1860
Defects Fixed ... 1861
Supported Query-Language Versions ... 1861
Upgrade Paths ... 1862
Upgrading ... 1862
Maintenance Release: 1.2.0.1.R3 (2023-09-27) ... 1863

Maintenance Release: 1.2.0.1.R2 (2022-12-13) ... 1868
Release: 1.2.0.0 (2022-07-21) ... 1872
Patch Releases ... 1873
New Features ... 1873
Improvements ... 1873
Defects Fixed ... 1875
Supported Query-Language Versions ... 1876
Upgrade Paths ... 1877
Upgrading ... 1877
Release: 1.2.0.0.R4 (2023-09-29) ... 1880
Release: 1.2.0.0.R3 (2022-12-15) ... 1885
Release: 1.2.0.0.R2 (2022-10-14) ... 1889
Release: 1.1.1.0 (2022-04-19) ... 1895
Patch Releases ... 1896
New Features ... 1896
Improvements ... 1897
Defects Fixed ... 1898
Supported Query-Language Versions ... 1899
Upgrade Paths ... 1899
Upgrading ... 1899
Release: 1.1.1.0.R7 (2023-01-23) ... 1902
Release: 1.1.1.0.R6 (2022-09-23) ... 1907
Release: 1.1.1.0.R5 (2022-07-21) ... 1913
Release: 1.1.1.0.R4 (2022-06-23) ... 1917
Release: 1.1.1.0.R3 (2022-06-07) ... 1923
Maintenance release: 1.1.1.0.R2 (2022-05-16) ... 1928
Release: 1.1.0.0 (2021-11-19) ... 1932
Patch releases ... 1934
New Features ... 1934
Improvements ... 1934
Defects Fixed ... 1936
Supported Query-Language Versions ... 1936
Upgrade Paths ... 1936
Upgrading ... 1936
Maintenance release: 1.1.0.0.R3 (2022-12-23) ... 1938
Maintenance release: 1.1.0.0.R2 (2022-05-16) ... 1943

Release: 1.0.5.1 (2021-10-01) 1947
Patch Releases 1948
New Features 1948
Improvements 1948
Defects Fixed 1949
Supported Query-Language Versions 1949
Upgrade Paths 1949
Upgrading 1949
Maintenance release: 1.0.5.1.R4 (2022-05-16) 1951
Release: 1.0.5.1.R3 (2022-01-13) 1954
Release: 1.0.5.1.R2 (2021-10-26) 1956
Release: 1.0.5.0 (2021-07-27) 1959
Patch Releases 1959
New Features 1959
Improvements 1960
Defects Fixed 1961
Supported Query-Language Versions 1961
Upgrade Paths 1961
Upgrading 1961
Maintenance release: 1.0.5.0.R5 (2022-05-16) 1963
Release: 1.0.5.0.R3 (2021-09-15) 1966
Release: 1.0.5.0.R2 (2021-08-16) 1968
Release: 1.0.4.2 (2021-06-01) 1971
Release: 1.0.4.2.R5 (2021-08-16) 1971
Release: 1.0.4.2.R4 (2021-07-23) 1972
Release: 1.0.4.2.R3 (2021-06-28) 1973
Release: 1.0.4.2.R2 (2021-06-01) 1974
Release: 1.0.4.2.R1 (2021-05-27) 1978
Release: 1.0.4.1 (2020-12-08) 1978
Patch Releases 1978
New Features 1979
Improvements 1979
Defects Fixed 1979
Supported Query-Language Versions 1980
Upgrade Paths 1980
Upgrading 1980

Release: 1.0.4.1.R1.1 (2021-03-22) 1982
Release: 1.0.4.1.R2 (2021-02-24) 1985
Release: 1.0.4.0 (2020-10-12) 1990
Patch Releases 1991
New Features 1991
Improvements 1991
Defects Fixed 1992
Supported Query-Language Versions 1992
Upgrade Paths 1993
Upgrading 1993
Release: 1.0.4.0.R2 (2021-02-24) 1995
Release: 1.0.3.0 (2020-08-03) 1998
Patch Releases 1998
New Features 1998
Improvements 1999
Defects Fixed 1999
Supported Query-Language Versions 2000
Upgrade Paths 2000
Upgrading 2000
Release: 1.0.3.0.R3 (2021-02-19) 2002
Release: 1.0.3.0.R2 (2020-10-12) 2005
Release: 1.0.2.2 (2020-03-09) 2008
Patch Releases 2008
Improvements 2008
Defects Fixed 2009
Supported Query-Language Versions 2009
Upgrade Paths 2010
Upgrading 2010
Release: 1.0.2.2.R6 (2021-02-19) 2012
Release: 1.0.2.2.R5 (2020-10-12) 2014
Release: 1.0.2.2.R4 (2020-07-23) 2017
Release: 1.0.2.2.R3 (2020-07-22) 2020
Release: 1.0.2.2.R2 (2020-04-02) 2020
Release: 1.0.2.1 (2019-11-22) 2023
Patch Releases 2023
New Features 2024

Improvements ... 2024
Defects Fixed ... 2024
Supported Query-Language Versions ... 2025
Upgrade Paths ... 2025
Upgrading ... 2025
Release: 1.0.2.1.R6 (2020-04-22) ... 2027
Release: 1.0.2.1.R5 (2020-04-22) ... 2030
Release: 1.0.2.1.R4 (2019-12-20) ... 2030
Release: 1.0.2.1.R3 (2019-12-12) ... 2033
Release: 1.0.2.1.R2 (2019-11-25) ... 2036
Release: 1.0.2.0 (2019-11-08) ... 2038
IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED ... 2038
Patch Releases ... 2038
New Features ... 2039
Supported Query-Language Versions ... 2039
Upgrade Paths ... 2039
Upgrading ... 2039
Release: 1.0.2.0.R3 (2020-05-05) ... 2041
Release: 1.0.2.0.R2 (2019-11-21) ... 2044
Release: 1.0.1.2 (2020-06-10) ... 2047
IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED ... 2047
Improvements ... 2047
Defects Fixed ... 2047
Supported Query-Language Versions ... 2047
Release: 1.0.1.1 (2020-06-26) ... 2048
IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED ... 2048
Defects Fixed ... 2048
Supported Query-Language Versions ... 2048
Release: 1.0.1.0 (2019-07-02) ... 2048
IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED ... 2048
Release 1.0.1.0.200502.0 (2019-10-31) ... 2048
Release 1.0.1.0.200463.0 (2019-10-15) ... 2049
Release 1.0.1.0.200457.0 (2019-09-19) ... 2050
Release 1.0.1.0.200369.0 (2019-08-13) ... 2051
Release 1.0.1.0.200366.0 (2019-07-26) ... 2052
Release 1.0.1.0.200348.0 (2019-07-02) ... 2054

Earlier Releases ... 2054
Using Neptune APIs ... 2067
Shared IAM actions ... 2067
Management API reference ... 2074
Clusters ... 2081
CreateDBCluster ... 2081
DeleteDBCluster ... 2092
ModifyDBCluster ... 2099
StartDBCluster ... 2109
StopDBCluster ... 2115
AddRoleToDBCluster ... 2121
RemoveRoleFromDBCluster ... 2122
FailoverDBCluster ... 2123
PromoteReadReplicaDBCluster ... 2129
DescribeDBClusters ... 2134
_____ ... 2136
DBCluster ... 2136
DBClusterMember ... 2142
DBClusterRole ... 2143
CloudwatchLogsExportConfiguration ... 2143
PendingCloudwatchLogsExports ... 2144
ClusterPendingModifiedValues ... 2144
Global databases ... 2145
CreateGlobalCluster ... 2146
DeleteGlobalCluster ... 2148
ModifyGlobalCluster ... 2150
DescribeGlobalClusters ... 2153
FailoverGlobalCluster ... 2154
RemoveFromGlobalCluster ... 2156
_____ ... 2158
GlobalCluster ... 2158
GlobalClusterMember ... 2160
Instances ... 2160
CreateDBInstance ... 2161
DeleteDBInstance ... 2173
ModifyDBInstance ... 2179

RebootDBInstance 2191
DescribeDBInstances 2196
DescribeOrderableDBInstanceOptions 2198
DescribeValidDBInstanceModifications 2200
_____ 2201
DBInstance 2201
DBInstanceStatusInfo 2205
OrderableDBInstanceOption 2206
PendingModifiedValues 2208
ValidStorageOptions 2209
ValidDBInstanceModificationsMessage 2210
Parameters 2210
CopyDBParameterGroup 2211
CopyDBClusterParameterGroup 2213
CreateDBParameterGroup 2215
CreateDBClusterParameterGroup 2217
DeleteDBParameterGroup 2219
DeleteDBClusterParameterGroup 2220
ModifyDBParameterGroup 2221
ModifyDBClusterParameterGroup 2222
ResetDBParameterGroup 2224
ResetDBClusterParameterGroup 2225
DescribeDBParameters 2227
DescribeDBParameterGroups 2228
DescribeDBClusterParameters 2229
DescribeDBClusterParameterGroups 2231
DescribeEngineDefaultParameters 2232
DescribeEngineDefaultClusterParameters 2233
_____ 2234
Parameter 2234
DBParameterGroup 2235
DBClusterParameterGroup 2236
DBParameterGroupStatus 2237
Subnets 2237
CreateDBSubnetGroup 2238
DeleteDBSubnetGroup 2240

ModifyDBSubnetGroup 2241
DescribeDBSubnetGroups 2242
_____ 2243
Subnet 2243
DBSubnetGroup 2244
Snapshots 2245
CreateDBClusterSnapshot 2245
DeleteDBClusterSnapshot 2249
CopyDBClusterSnapshot 2252
ModifyDBClusterSnapshotAttribute 2256
RestoreDBClusterFromSnapshot 2258
RestoreDBClusterToPointInTime 2267
DescribeDBClusterSnapshots 2277
DescribeDBClusterSnapshotAttributes 2280
_____ 2281
DBClusterSnapshot 2281
DBClusterSnapshotAttribute 2283
DBClusterSnapshotAttributesResult 2284
Events 2285
CreateEventSubscription 2285
DeleteEventSubscription 2288
ModifyEventSubscription 2290
DescribeEventSubscriptions 2293
AddSourceIdentifierToSubscription 2294
RemoveSourceIdentifierFromSubscription 2296
DescribeEvents 2298
DescribeEventCategories 2300
_____ 2300
Event 2300
EventCategoriesMap 2301
EventSubscription 2301
Other 2303
AddTagsToResource 2304
ListTagsForResource 2304
RemoveTagsFromResource 2305
ApplyPendingMaintenanceAction 2306

DescribePendingMaintenanceActions 2307
DescribeDBEngineVersions 2309
_____ 2310
DBEngineVersion 2310
EngineDefaults 2312
PendingMaintenanceAction 2312
ResourcePendingMaintenanceActions 2313
UpgradeTarget 2314
Tag 2314
Datatypes 2315
AvailabilityZone 2315
DBSecurityGroupMembership 2316
DomainMembership 2316
DoubleRange 2316
Endpoint 2317
Filter 2317
Range 2318
ServerlessV2ScalingConfiguration 2318
ServerlessV2ScalingConfigurationInfo 2319
Timezone 2319
VpcSecurityGroupMembership 2319
API Faults 2320
AuthorizationAlreadyExistsFault 2322
AuthorizationNotFoundFault 2323
AuthorizationQuotaExceededFault 2323
CertificateNotFoundFault 2323
DBClusterAlreadyExistsFault 2324
DBClusterNotFoundFault 2324
DBClusterParameterGroupNotFoundFault 2324
DBClusterQuotaExceededFault 2324
DBClusterRoleAlreadyExistsFault 2325
DBClusterRoleNotFoundFault 2325
DBClusterRoleQuotaExceededFault 2325
DBClusterSnapshotAlreadyExistsFault 2326
DBClusterSnapshotNotFoundFault 2326
DBInstanceAlreadyExistsFault 2326

DBInstanceNotFoundFault 2327
DBLogFileNotFoundFault 2327
DBParameterGroupAlreadyExistsFault 2327
DBParameterGroupNotFoundFault 2327
DBParameterGroupQuotaExceededFault 2328
DBSecurityGroupAlreadyExistsFault 2328
DBSecurityGroupNotFoundFault 2328
DBSecurityGroupNotSupportedFault 2329
DBSecurityGroupQuotaExceededFault 2329
DBSnapshotAlreadyExistsFault 2329
DBSnapshotNotFoundFault 2329
DBSubnetGroupAlreadyExistsFault 2330
DBSubnetGroupDoesNotCoverEnoughAZs 2330
DBSubnetGroupNotAllowedFault 2330
DBSubnetGroupNotFoundFault 2331
DBSubnetGroupQuotaExceededFault 2331
DBSubnetQuotaExceededFault 2331
DBUpgradeDependencyFailureFault 2332
DomainNotFoundFault 2332
EventSubscriptionQuotaExceededFault 2332
GlobalClusterAlreadyExistsFault 2332
GlobalClusterNotFoundFault 2333
GlobalClusterQuotaExceededFault 2333
InstanceQuotaExceededFault 2333
InsufficientDBClusterCapacityFault 2334
InsufficientDBInstanceCapacityFault 2334
InsufficientStorageClusterCapacityFault 2334
InvalidDBClusterEndpointStateFault 2335
InvalidDBClusterSnapshotStateFault 2335
InvalidDBClusterStateFault 2335
InvalidDBInstanceStateFault 2336
InvalidDBParameterGroupStateFault 2336
InvalidDBSecurityGroupStateFault 2336
InvalidDBSnapshotStateFault 2336
InvalidDBSubnetGroupFault 2337
InvalidDBSubnetGroupStateFault 2337

InvalidDBSubnetStateFault 2337
InvalidEventSubscriptionStateFault 2338
InvalidGlobalClusterStateFault 2338
InvalidOptionGroupStateFault 2338
InvalidRestoreFault 2339
InvalidSubnet 2339
InvalidVPCNetworkStateFault 2339
KMSKeyNotAccessibleFault 2339
OptionGroupNotFoundFault 2340
PointInTimeRestoreNotEnabledFault 2340
ProvisionedIopsNotAvailableInAZFault 2340
ResourceNotFoundFault 2341
SNSInvalidTopicFault 2341
SNSNoAuthorizationFault 2341
SNSTopicArnNotFoundFault 2342
SharedSnapshotQuotaExceededFault 2342
SnapshotQuotaExceededFault 2342
SourceNotFoundFault 2342
StorageQuotaExceededFault 2343
StorageTypeNotSupportedFault 2343
SubnetAlreadyInUse 2343
SubscriptionAlreadyExistFault 2344
SubscriptionCategoryNotFoundFault 2344
SubscriptionNotFoundFault 2344
Data API reference 2345
General 2349
GetEngineStatus 2349
ExecuteFastReset 2351
_____ 2353
QueryLanguageVersion 2353
FastResetToken 2353
Querying 2354
ExecuteGremlinQuery 2354
ExecuteGremlinExplainQuery 2356
ExecuteGremlinProfileQuery 2358
ListGremlinQueries 2360

GetGremlinQueryStatus 2362
CancelGremlinQuery 2363
_____ 2364
ExecuteOpenCypherQuery 2364
ExecuteOpenCypherExplainQuery 2366
ListOpenCypherQueries 2368
GetOpenCypherQueryStatus 2369
CancelOpenCypherQuery 2371
_____ 2372
QueryEvalStats 2372
GremlinQueryStatus 2373
GremlinQueryStatusAttributes 2373
Bulk loader 2374
StartLoaderJob 2374
GetLoaderJobStatus 2381
ListLoaderJobs 2384
CancelLoaderJob 2385
_____ 2386
LoaderIdResult 2386
Streams 2386
GetPropertygraphStream 2387
_____ 2390
PropertygraphRecord 2390
PropertygraphData 2390
Statistics 2391
GetPropertygraphStatistics 2392
ManagePropertygraphStatistics 2393
DeletePropertygraphStatistics 2394
GetPropertygraphSummary 2396
_____ 2397
Statistics 2397
StatisticsSummary 2398
DeleteStatisticsValueMap 2398
RefreshStatisticsIdMap 2398
NodeStructure 2399
EdgeStructure 2399

SubjectStructure 2399
PropertygraphSummaryValueMap 2400
PropertygraphSummary 2400
ML data processing 2402
StartMLDataProcessingJob 2402
ListMLDataProcessingJobs 2405
GetMLDataProcessingJob 2406
CancelMLDataProcessingJob 2408
_____ 2409
MlResourceDefinition 2409
MlConfigDefinition 2410
ML model training 2410
StartMLModelTrainingJob 2410
ListMLModelTrainingJobs 2414
GetMLModelTrainingJob 2415
CancelMLModelTrainingJob 2417
_____ 2418
CustomModelTrainingParameters 2418
ML model transform 2419
StartMLModelTransformJob 2419
ListMLModelTransformJobs 2422
GetMLModelTransformJob 2423
CancelMLModelTransformJob 2424
_____ 2426
CustomModelTransformParameters 2426
ML Inference endpoint 2426
CreateMLEndpoint 2427
ListMLEndpoints 2429
GetMLEndpoint 2430
DeleteMLEndpoint 2432
Exceptions 2433
AccessDeniedException 2434
BadRequestException 2435
BulkLoadIdNotFoundException 2435
CancelledByUserException 2435
ClientTimeoutException 2436

ConcurrentModificationException 2436
ConstraintViolationException 2437
ExpiredStreamException 2437
FailureByQueryException 2437
IllegalArgumentException 2438
InternalFailureException 2438
InvalidArgumentException 2439
InvalidNumericDataException 2439
InvalidParameterException 2439
LoadUrlAccessDeniedException 2440
MalformedQueryException 2440
MemoryLimitExceededException 2441
MethodNotAllowedException 2441
MissingParameterException 2442
MLResourceNotFoundException 2442
ParsingException 2442
PreconditionsFailedException 2443
QueryLimitExceededException 2443
QueryLimitException 2444
QueryTooLargeException 2444
ReadOnlyViolationException 2444
S3Exception 2445
ServerShutdownException 2445
StatisticsNotAvailableException 2446
StreamRecordsNotFoundException 2446
ThrottlingException 2447
TimeLimitExceededException 2447
TooManyRequestsException 2447
UnsupportedOperationException 2448
UnloadUrlAccessDeniedException 2448

What Is Amazon Neptune?

Amazon Neptune is a fast, reliable, fully managed graph database service that makes it easy to build and run applications that work with highly connected datasets. The core of Neptune is a purpose-built, high-performance graph database engine. This engine is optimized for storing billions of relationships and querying the graph with milliseconds latency. Neptune supports the popular property-graph query languages Apache TinkerPop Gremlin and Neo4j's openCypher, and the W3C's RDF query language, SPARQL. This enables you to build queries that efficiently navigate highly connected datasets. Neptune powers graph use cases such as recommendation engines, fraud detection, knowledge graphs, drug discovery, and network security.

The Neptune database is highly available, with read replicas, point-in-time recovery, continuous backup to Amazon S3, and replication across Availability Zones. Neptune provides data security features, with support for encryption at rest and in transit. Neptune is fully managed, so you no longer need to worry about database management tasks like hardware provisioning, software patching, setup, configuration, or backups.

Neptune Analytics is an analytics database engine that complements Neptune database and that can quickly analyze large amounts of graph data in memory to get insights and find trends. Neptune Analytics is a solution for quickly analyzing existing graph databases or graph datasets stored in a data lake. It uses popular graph analytic algorithms and low-latency analytic queries.

To learn more about using Amazon Neptune, we recommend that you start with the following sections:

- Getting started with Amazon Neptune
- Overview of Amazon Neptune features

Also, before you begin designing a database, we recommend that you consult the GitHub repository AWS Reference Architectures for Using Graph Databases, where you can inform your choices about graph data models and query languages, and browse examples of reference deployment architectures.

Key Service Components

- *Primary DB instance* – Supports read and write operations, and performs all of the data modifications to the cluster volume. Each Neptune DB cluster has one primary DB instance that is responsible for writing (that is, loading or modifying) graph database contents.

- *Neptune replica* – Connects to the same storage volume as the primary DB instance and supports only read operations. Each Neptune DB cluster can have up to 15 Neptune Replicas in addition to the primary DB instance. This provides high availability by locating Neptune Replicas in separate Availability Zones and distribution load from reading clients.
- *Cluster volume* – Neptune data is stored in the cluster volume, which is designed for reliability and high availability. A cluster volume consists of copies of the data across multiple Availability Zones in a single AWS Region. Because your data is automatically replicated across Availability Zones, it is highly durable, and there is little possibility of data loss.

Supports Open Graph APIs

Amazon Neptune supports open graph APIs for both property graphs (Gremlin and openCypher) and RDF graphs (SPARQL). It provides high performance for both of these graph models and their query languages. You can choose the Property Graph (PG) model and access the same graph with both the openCypher query language and/or the Gremlin query language. If you use the W3C standard Resource Description Framework (RDF) model, you can access your graph using the standard SPARQL query language.

Highly Secure

Neptune provides multiple levels of security for your database. Security features include network isolation using Amazon VPC, and encryption at rest using keys that you create and control through AWS Key Management Service (AWS KMS). On an encrypted Neptune instance, data in the underlying storage is encrypted, as are the automated backups, snapshots, and replicas in the same cluster.

Fully Managed

With Amazon Neptune, you don’t have to worry about database management tasks like hardware provisioning, software patching, setup, configuration, or backups.

You can use Neptune to create sophisticated, interactive graph applications that can query billions of relationships in milliseconds. SQL queries for highly connected data are complex and hard to tune for performance. With Neptune, you can use the popular graph query languages Gremlin, openCypher, and SPARQL to execute powerful queries that are easy to write and perform well on connected data. This capability significantly reduces code complexity so that you can quickly create applications that process relationships.

Neptune is designed to offer greater than 99.99 percent availability. It increases database performance and availability by tightly integrating the database engine with an SSD-backed virtualized storage layer that is built for database workloads. Neptune storage is fault-tolerant and self-healing. Disk failures are repaired in the background without loss of database availability. Neptune automatically detects database crashes and restarts without the need for crash recovery or rebuilding the database cache. If the entire instance fails, Neptune automatically fails over to one of up to 15 read replicas.

Changes and Updates to Amazon Neptune

The following table describes important changes to Amazon Neptune.

Change	Description	Date
Engine version 1.5.0.0	As of 2026-10-13, engine version 1.5.0.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region. For more information about this engine version, see Neptune Engine Release 1.5.0.0.	October 13, 2026
Engine version 1.4.8.1	As of 2026-10-06, AWS is deploying engine version 1.4.8.1 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.8.1.	October 6, 2026
Engine version 1.4.7.1	As of 2026-09-18, AWS is deploying engine version 1.4.7.1 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.7.1.	September 18, 2026

Engine version 1.4.8.0	As of 2026-07-27, AWS is deploying engine version 1.4.8.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.8.0.	July 27, 2026
Dual-stack (IPv6) support	Amazon Neptune now supports dual-stack mode, which allows DB clusters to communicate over both IPv4 and IPv6 protocols. For more information, see Working with a Neptune DB cluster in dual-stack mode.	June 12, 2026
Neptune storage managemen t APIs	Amazon Neptune now provides storage managemen t APIs: the GetStorageInfo API for analyzing storage usage, and the online index rebuild APIs for reclaiming index storage without downtime. These APIs are available starting with engine release 1.5.0.0.	June 9, 2026
Neptune in Asia Pacific (Hyderabad) region	Amazon Neptune is now available in the Asia Pacific (Hyderabad) (`ap-south-2`) region.	March 10, 2026

Engine version 1.4.7.0	As of 2026-03-03, AWS is deploying engine version 1.4.7.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.7.0.	March 3, 2026
Engine version 1.4.6.3	As of 2025-12-18, AWS is deploying engine version 1.4.6.3 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.6.3.	December 18, 2025
Engine version 1.4.6.2	As of 2025-11-18, AWS is deploying engine version 1.4.6.2 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.6.2.	November 18, 2025

Engine version 1.4.6.1	As of 2025-09-18, AWS is deploying engine version 1.4.6.1 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.6.1.	September 18, 2025
Engine version 1.4.6.0	As of 2025-09-02, AWS is deploying engine version 1.4.6.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.6.0.	September 2, 2025
Public endpoints	Amazon Neptune now supports public endpoints . For more information see Public endpoints.	August 27, 2025
Engine version 1.4.5.1	As of 2025-06-30, AWS is deploying engine version 1.4.5.1 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.5.1.	June 30, 2025

Neptune launched in the Asia Pacific (Melbourne) and Canada West (Calgary) regions	Amazon Neptune is now available in the Asia Pacific (Melbourne) (`ap-southeast-4`) and Canada West (Calgary) (`ca-west-1`)regions.	May 28, 2025
Engine version 1.4.5.0	As of 2025-04-09, AWS is deploying engine version 1.4.5.0 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.4.5.0.	April 9, 2025
Neptune launched in the Asia Pacific (Malaysia) region	Amazon Neptune is now available in the Asia Pacific (Malaysia) (`ap-southeast-5`) region. Managed notebooks are not supported for the Asia Pacific (Malaysia) region at this time, but can be setup manually. See Setting up Neptune notebooks manually for more information.	February 28, 2025

Engine version 1.4.4.0	As of 2025-02-24, AWS is deploying engine version 1.4.4.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.4.0.	February 24, 2025
Engine version 1.4.3.0	As of 2025-01-21, AWS is deploying engine version 1.4.3.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.3.0.	January 21, 2025
Engine version 1.4.2.0	As of 2024-12-19, AWS is deploying engine version 1.4.2.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.2.0.	December 19, 2024

Engine version 1.4.1.0	As of 2024-11-21, AWS is deploying engine version 1.4.1.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.1.0.	November 21, 2024
Engine version 1.4.0.0	As of 2024-11-06, AWS is deploying engine version 1.4.0.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.4.0.0.	November 6, 2024
Engine version 1.3.4.0	As of 2024-10-01, AWS is deploying engine version 1.3.4.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.3.4.0.	October 1, 2024

Engine version 1.3.3.0	As of 2024-08-05, AWS is deploying engine version 1.3.3.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.3.3.0.	August 5, 2024
Engine version 1.2.1.2	As of 2024-08-05, AWS is deploying engine version 1.2.1.2 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.2.1.2.	August 5, 2024
Engine version 1.3.2.1	As of 2024-06-20, AWS is deploying engine version 1.3.2.1 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.3.2.1.	June 20, 2024

Engine version 1.3.2.0	As of 2024-06-10, AWS is deploying engine version 1.3.2.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.3.2.0.	June 10, 2024
Engine version 1.2.1.1	As of 2024-03-11, AWS is deploying engine version 1.2.1.1 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.2.1.1.	March 11, 2024
Engine version 1.3.1.0	As of 2024-03-06, AWS is deploying engine version 1.3.1.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.3.1.0.	March 6, 2024
Update to AWS managed policy permissions	The `NeptuneReadOnlyAccess` and `NeptuneFullAccess` managed policies now include `Sid` (statement ID) as an identifier in the policy statement.	January 22, 2024

Neptune now offers I/O-Optimized storage	With I/O-Optimized storage, you pay for the storage and instances you are using. The storage costs are higher than for standard storage, but you pay nothing for any I/O that you use.	December 13, 2023
IAM Managed Policy changes for Neptune	The **NeptuneConsoleFullAccess** IAM managed policy has been updated to grant the permissions needed to interact with Neptune Analytics graphs, a new **NeptuneGraphReadOnlyAccess** policy has been added to provide read-only access to Neptune Analytics graph resources, and a new `AWSServiceRoleForNeptuneGraphPolicy` policy has been added to let Neptune Analytics graphs to publish CloudWatch operational and usage metrics and logs.	November 29, 2023
Engine version 1.3.0.0	As of 2023-11-15, AWS is deploying engine version 1.3.0.0 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.3.0.0.	November 15, 2023

Neptune launched in the Israel (Tel Aviv) region	Amazon Neptune is now available in the Israel (Tel Aviv) (`il-central-1`) region.	November 13, 2023
Blog post about implementing time-to-live in Neptune property graphs	See Implement Time to Live in Amazon Neptune, Part 1: Property Graph by Melissa Kwok, Mike Havey, and Kevin Phillips.	October 27, 2023
Engine version 1.2.1.0.R7	As of 2023-10-06, AWS is deploying engine version 1.2.1.0.R7 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.1.0.R7.	October 6, 2023
Engine version 1.2.0.0.R4	As of 2023-09-29, AWS is deploying engine version 1.2.0.0.R4 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.0.0.R4.	September 29, 2023

Engine version 1.2.0.1.R3	As of 2023-09-27, AWS is deploying engine version 1.2.0.1.R3 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.0.1.R3.	September 27, 2023
Engine version 1.2.1.0.R6	As of 2023-09-12, AWS is deploying engine version 1.2.1.0.R6 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.1.0.R6.	September 12, 2023
Engine version 1.2.0.2.R6	As of 2023-09-12, AWS is deploying engine version 1.2.0.2.R6 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.0.2.R6.	September 12, 2023
Blog post about using a blue/green deployment strategy for Neptune engine upgrades	See Improve availability of Amazon Neptune during engine upgrade using blue/green deployment by Ankit Gupta and Abhishek Mishra.	September 11, 2023

Engine version 1.2.1.0.R5	As of 2023-09-02, AWS is deploying engine version 1.2.1.0.R5 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.1.0.R5.	September 2, 2023
Engine version 1.2.0.2.R5	As of 2023-08-16, AWS is deploying engine version 1.2.0.2.R5 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.0.2.R5.	August 16, 2023
Engine version 1.2.1.0.R4	As of 2023-08-10, AWS is deploying engine version 1.2.1.0.R4 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.1.0.R4.	August 10, 2023
Blog post about Neptune engine release 1.2.1.0	See Exploring the feature packed 1.2.1.0 release for Amazon Neptune by Joy Wang, Kevin Phillips, Andrea Nassisi, and Navtanay Sinha.	August 4, 2023

Blog post about building a multimodal database solution with Neptune	See Design a use case-driven, highly scalable multimodel database solution using Amazon Neptune by Mike Havey.	July 18, 2023
Blog post about Neptune Serverless use cases and best practices	See Use cases and best practices to optimize cost and performance with Amazon Neptune Serverless by Kevin Phillips and Ankit Gupta.	June 28, 2023
Engine version 1.2.1.0.R3	As of 2023-06-13, AWS is deploying engine version 1.2.1.0.R3 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.1.0.R3.	June 13, 2023
Blog post about generating leisure suggestions in real time	See Generate suggestions for leisure activities in real time with Amazon Neptune by Michael Meidlinger and Nils Müller.	June 6, 2023
Blog post about molecular modeling with Neptune and RDKit	See Model molecular SMILES data with Amazon Neptune and RDKit by Graham Kutchek.	June 1, 2023

Blog post about using caching to accelerate Neptune performance (Part 3)	See Accelerate graph query performance with caching in Amazon Neptune, Part 3: Neptune cluster-wide caching architectures with Amazon ElastiCache by Taylor Riggan, Abhishek Mishra, Melissa Kwok, and Kelvin Lawrence.	May 26, 2023
Blog post about using caching to accelerate Neptune performance (Part 2)	See Accelerate graph query performance with caching in Amazon Neptune, Part 2: Additional Neptune caching features by Taylor Riggan, Abhishek Mishra, Melissa Kwok, and Kelvin Lawrence.	May 26, 2023
Blog post about using caching to accelerate Neptune performance (Part 1)	See Accelerate graph query performance with caching in Amazon Neptune, Part 1: Queries and buffer pool caching by Taylor Riggan, Abhishek Mishra, Melissa Kwok, and Kelvin Lawrence.	May 26, 2023
Blog post about supply-chain analysis using Neptune	See Supply chain data analysis and visualization using Amazon Neptune and the Neptune workbench by Dhiraj Thakur and Rajdip Chaudhur.	May 10, 2023

Engine version 1.2.0.2.R4	As of 2023-05-08, AWS is deploying engine version 1.2.0.2.R4 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.0.2.R4.	May 8, 2023
Neptune launched in the Middle East (UAE) region	Amazon Neptune is now available in the Middle East (UAE) (`me-central-1`) region.	May 2, 2023
Engine version 1.2.1.0.R2	As of 2023-05-02, AWS is deploying engine version 1.2.1.0.R2 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.1.0.R2.	May 2, 2023
Blog post about building a knowledge graph on Neptune with AI-powered video analysis using Media2Cloud	See Build a knowledge graph on Amazon Neptune with AI-powered video analysis using Media2Cloud by Mike Havey.	May 2, 2023
Blog post about how DevOcean built a vulnerability remediation platform using Neptune	See How DevOcean built a vulnerability remediation management platform for cloud-native applications using Amazon Neptune by Gil Makmel and Charles Ivie.	April 25, 2023

Blog post about how Getir build a fraud detection system using Neptune	See How Getir build a comprehensive fraud detection system using Amazon Neptune and Amazon DynamoDB by Berkay Berkman, Mahmut Turan, Mutlu Polatcan, Umut Cemal Kıraç, Yağız Yanıkoğlu, and Esra Kayabali.	April 6, 2023
Blog post about how Wiz reimagines cloud security using Neptune	See The World is a graph: How Wiz reimagines cloud security using a graph in Amazon Neptune by Ami Luttwak and Brad Bebee.	March 31, 2023
Engine version 1.2.0.2.R3	As of 2023-03-27, AWS is deploying engine version 1.2.0.2.R3 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.0.2.R3.	March 27, 2023
Blog post about how CSC Generation powers product discovery using Neptune	See How CSC Generation powers product discovery with knowledge graphs using Amazon Neptune by Bobber Cheng, Ronit Rudra, and Melissa Kwok.	March 21, 2023

Engine version 1.2.1.0	As of 2023-03-08, AWS is deploying engine version 1.2.1.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.2.1.0.	March 8, 2023
Blog post about exploring new features of TinkerPop 3.6.x in Neptune	See Exploring new features of Apache TinkerPop 3.6.x in Amazon Neptune by Stephen Mallette.	March 8, 2023
Blog post about using semantic reasoning to infer new facts from your RDF graph	See Use semantic reasoning to infer new facts from your RDF graph by integrating RDFox with Amazon Neptune by Charles Ivie and Diana Marks.	February 20, 2023
Blog post about analyzing healthcare FHIR data with Neptune	See Analyze healthcare FHIR data with Amazon Neptune by Alena Schmickl.	February 13, 2023
Blog post about building a real-time fraud detection solution using Neptune ML	See Build a real-time fraud detection solution using Amazon Neptune ML by Hua Shu and Soji Adeshina.	February 8, 2023

Engine version 1.1.1.0.R7	As of 2023-01-23, AWS is deploying engine version 1.1.1.0.R7 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.1.1.0.R7.	January 23, 2023
Graph-explorer released	Graph-explorer is an open-source front-end web application tool for visualizing graph data. See https://github.com/aws/graph-explorer.	January 3, 2023
Maintenance release version 1.1.0.0.R3	As of 2022-12-23, AWS is deploying engine version 1.1.0.0.R3 maintenance release to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.1.0.0.R3.	December 23, 2022
Neptune workbench now operates on Amazon Linux 2 and JupyterLab 3.	Neptune graph notebooks now run in an Amazon Linux 2 environment with JupyterLab 3. See Migrating your Neptune notebooks from Jupyter to JupyterLab 3 for information about how to move to this new environment.	December 21, 2022

Blog post about power recommendations and search using an IMDb knowledge graph (part 3)	See Power recommendations and search using an IMDb knowledge graph – Part 3 by Divya Bhargavi, Soji Adeshina, Gaurav Rele, Karan Sindwani, Vidya Sagar Ravipati, and Matthew Rhodes.	December 20, 2022
Blog post about power recommendations and search using an IMDb knowledge graph (part 2)	See Power recommendations and search using an IMDb knowledge graph – Part 2 by Matthew Rhodes, Soji Adeshina, Divya Bhargavi, Gaurav Rele, Karan Sindwani, and Vidya Sagar Ravipati.	December 20, 2022
Blog post about power recommendations and search using an IMDb knowledge graph (part 1)	See Power recommend ations and search using an IMDb knowledge graph – Part 1 by Gaurav Rele, Soji Adeshina, Divya Bhargavi, Karan Sindwani, Vidya Sagar Ravipati, and Matthew Rhodes.	December 20, 2022
Neptune Serverless is now available in new AWS regions	As of 2022-12-16, Neptune Serverless has launched in the following new AWS regions: Canada (Central), Europe (Stockholm), Europe (Frankfur t), Asia Pacific (Singapore), and Asia Pacific (Sydney). See Amazon Neptune Serverless constraints for all the regions where Neptune Serverless is available.	December 16, 2022

Engine version 1.2.0.2.R2	As of 2022-12-15, AWS is deploying engine version 1.2.0.2.R2 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.0.2.R2.	December 15, 2022
Engine version 1.2.0.0.R3	As of 2022-12-15, AWS is deploying engine version 1.2.0.0.R3 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.0.0.R3.	December 15, 2022
Engine version 1.2.0.1.R2	As of 2022-12-13, AWS is deploying engine version 1.2.0.1.R2 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.0.1.R2.	December 13, 2022
Blog post about designing an educational big data analysis architecture with AWS	See Designing an educational big data analysis architecture with AWS by Lavanya Sood.	December 13, 2022
Blog post about how graph databases can enhance learning	See How graph databases can enhance learning by Lavanya Sood.	December 8, 2022

Blog post about loading RDF data into Neptune using AWS Glue	See Load RDF data into Amazon Neptune with AWS Glue by Mike Havey and Fabrizio Napolitano.	November 23, 2022
Engine version 1.2.0.2	As of 2022-11-20, AWS is deploying engine version 1.2.0.2 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.2.0.2.	November 20, 2022
Engine version 1.2.0.1	As of 2022-10-26, AWS is deploying engine version 1.2.0.1 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.2.0.1.	October 26, 2022
Blog post about fraud detection using Neptune	See Empowering fraud detection at Delivery Hero with Amazon Neptune by Wilson Tang, Amr Elnaggar, Matias Pons, Mohammad Azzam, Saurabh Deshpande, and Luis Rodrigues Soares.	October 26, 2022

Blog post about Neptune Serverless	See Introducing Amazon Neptune Serverless – A Fully Managed Graph Database that Adjusts Capacity for Your Workloads by Danilo Poccia.	October 26, 2022
Blog post about event-driven RDF imports to Neptune using Lambda and SPARQL UPDATE LOAD	See How NXP performs event-driven RDF imports to Amazon Neptune using AWS Lambda and SPARQL UPDATE LOAD by John Walker, Onno Buijs, Charles Ivie, and Javy de Koning.	October 20, 2022
Engine version 1.2.0.0.R2	As of 2022-10-14, AWS is deploying engine version 1.2.0.0.R2 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.2.0.0.R2.	October 14, 2022
Blog post about encoding multi-lingual text properties in Neptune	See Encode multi-lingual text properties in Amazon Neptune to train predictive models by Jiani Zhang.	October 14, 2022
Blog post about automated testing of Neptune data access	See Automated testing of Amazon Neptune data access with Apache TinkerPop Gremlin by Greg Biegel.	September 28, 2022

Engine version 1.1.1.0.R6	As of 2022-09-23, AWS is deploying engine version 1.1.1.0.R6 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.1.1.0.R6.	September 23, 2022
Blog post about how Informatica® uses Neptune	See How Informatica® Cloud Data Governance and Catalog uses Amazon Neptune for knowledge graphs by Tiju Titus John, Deepak Ram, and Farooq Ashraf.	September 20, 2022
Blog post about using Neptune and Tom Sawyer Perspectives to uncover financial fraud	See Uncover financial fraud with Amazon Neptune and Tom Sawyer Perspectives by Janet M. Six, Senior Product Manager at Tom Sawyer Software.	August 30, 2022
Blog post about building a GNN-based real-time fraud detection solution using SageMaker, Neptune, and DGL	See Build a GNN-based real-time fraud detection solution using Amazon SageMaker, Amazon Neptune, and the Deep Graph Library by Jian Zhang, Haozhu Wang, and Mengxin Zhu.	August 11, 2022
Blog post about using resource tags to stop and start Neptune environment resources	See Automate the stopping and starting of Amazon Neptune environment resources using resource tags by Kevin Phillips.	August 1, 2022

Blog post about an artist who contributes to Apache TinkerPop	See Beyond Code: The Artist Who Contributes to Apache TinkerPop by Stephen Mallette and Ketrina Thompson.	August 1, 2022
Blog post about fine-grained access control for Neptune data plane actions	See Fine Grained Access Control for Amazon Neptune data plane actions by Abhishek Mishra and Ankit Gupta.	July 29, 2022
Blog post about Neptune global databases	See Introducing Amazon Neptune Global Database by Navtanay Sinha.	July 27, 2022
Engine version 1.2.0.0	As of 2022-07-21, AWS is deploying engine version 1.2.0.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.2.0.0.	July 21, 2022
Engine version 1.1.1.0.R5	As of 2022-07-21, AWS is deploying engine version 1.1.1.0.R5 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.1.1.0.R5.	July 21, 2022

Engine version 1.1.1.0.R4	As of 2022-06-23, AWS is deploying engine version 1.1.1.0.R4 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.1.1.0.R4.	June 23, 2022
Simplified graph analytics and machine learning workflows with Python integration	You can now run graph analytics and machine learning tasks on graph data stored in Amazon Neptune using an open-sour ce Python integration that simplifies data science and ML workflows. See the AWS Data Wrangler documentation for Neptune.	June 7, 2022
Engine version 1.1.1.0.R3	As of 2022-06-07, AWS is deploying engine version 1.1.1.0.R3 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.1.1.0.R3.	June 7, 2022
Blog post about detecting fake news	See Detect social media fake news using graph machine learning with Amazon Neptune ML, by Hasan Shojaei and Sarita Joshi.	May 19, 2022

Blog post about using SQL Server Integration Services (SSIS) with Neptune	See Discover new insights from your data using SQL Server Integration Services (SSIS) and Amazon Neptune by Mesgana Gormley and Melissa Kwok.	May 18, 2022
Maintenance release version 1.1.1.0.R2	As of 2022-05-16, AWS is deploying engine version 1.1.1.0.R2 maintenance release to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.1.1.0.R2.	May 16, 2022
Maintenance release version 1.1.0.0.R2	As of 2022-05-16, AWS is deploying engine version 1.1.0.0.R2 maintenance release to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.1.0.0.R2.	May 16, 2022

Maintenance release version 1.0.5.1.R4	As of 2022-05-16, AWS is deploying engine version 1.0.5.1.R4 maintenance release to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.0.5.1.R4.	May 16, 2022
Maintenance release version 1.0.5.0.R5	As of 2022-05-16, AWS is deploying engine version 1.0.5.0.R5 maintenance release to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.0.5.0.R5.	May 16, 2022
Blog post about openCyphe r general availability in Neptune	See Announcing the General Availability of openCypher support for Amazon Neptune, by Navtanay Sinha and Dave Bechberger.	April 22, 2022
Engine version 1.1.1.0	As of 2022-04-19, AWS is deploying engine version 1.1.1.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.1.1.0.	April 19, 2022

Blog post about data lineage	See Build data lineage for data lakes using AWS Glue, Amazon Neptune, and Spline, by Khoa Nguyen, Krithivasan Balasubramaniyan, and Rahul Shaurya.	April 1, 2022
Upgrades to engine release 1.1.0.0 are re-enabled	As of 2022-02-21, upgrades to engine release 1.1.0.0 were temporarily disabled. They are now re-enabled.	March 22, 2022
Blog post about engineering utility reliability	See Using cloud-based, data-informed, power system models to engineer utility reliability, by Abhineet Parchure.	March 22, 2022
Neptune launched in Africa (Cape Town)	Amazon Neptune is now available in Africa (Cape Town) (`af-south-1`). However, Neptune workbench notebook support is temporarily disabled on the Neptune console in that region.	February 24, 2022
Blog post about model-driven graphs using OWL	See Model-driven graphs using OWL in Amazon Neptune, by Mike Havey.	February 23, 2022
Blog post about exploring semantic knowledge graphs with Rhizomer	See Explore the semantic knowledge graphs without SPARQL using Amazon Neptune with Rhizomer, by Roberto García.	February 22, 2022

Blog post about graphing the utility grid	See Graphing the utility grid on AWS, by Bobby Wilson and Joseph Beer.	February 18, 2022
New Neptune ML text feature encoding options	Neptune now supports FastText and Sentence BERT text encoding for training. See FastText features in Neptune ML and Sentence BERT features in Neptune ML.	February 15, 2022
Blog post about geospatial queries using OpenSearch with Neptune	See Combine Amazon Neptune and Amazon OpenSearch Service for geospatial queries, by Ross Gabay and Abhilash Vinod.	February 1, 2022
Blog post about Financial Crime Discovery using Amazon EKS and Neptune	See Financial Crime Discovery using Amazon EKS and Graph Databases, by Severin Gassauer-Fleissner and Zahi Ben Shabat.	February 1, 2022
A Neptune cluster volume can now grow to 128 tebibytes (TiB) in size	In all supported regions except China and GovCloud, the size limit of a Neptune cluster volume has now increased from 64 TiB to 128 Tib. This applies to all engine releases starting with release 1.0.2.2. See the Amazon Neptune storage page.	February 1, 2022
Neptune full-text search now integrates with all versions of OpenSearch.	See Full text search in Amazon Neptune using Amazon OpenSearch Service.	January 28, 2022

Blog post about using Docker containers to deploy graph notebooks	See Use Docker containers to deploy Graph Notebooks on AWS, by Ganesh Sawhney and Qiang Zhang.	January 22, 2022
Engine version 1.0.5.1.R3	As of 2022-01-13, AWS is deploying engine version 1.0.5.1.R3 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.5.1.R3.	January 13, 2022
Blog post about a graph-based recommendation system using Neptune ML	See Graph-based recommendation system with Neptune ML: An illustration on social network link prediction challenges, by Yanwei Cui and Will Badr.	January 12, 2022
Blog post about autoscaling Neptune	See Auto scale your Amazon Neptune database to meet workload demands, by Navtanay Sinha and Sudhanshu Gupta.	November 29, 2021
Blog post about interactive graph data analytics and visualizations	See Build interactive graph data analytics and visualizations using Amazon Neptune, Amazon Athena Federated Query, and Amazon QuickSight, by Sandeep Veldi and Abhishek Mishra.	November 24, 2021

Blog post about detecting identity fraud using graph-based deep learning	See How Careem is detecting identity fraud using graph-based deep learning and Amazon Neptune, by Kevin O'Brien, Kamran Habib, and Will Badr.	November 23, 2021
Engine version 1.1.0.0	As of 2021-11-19, AWS is deploying engine version 1.1.0.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.1.0.0.	November 19, 2021
Blog post about centraliz ing data protection and compliance	See Centralizing data protection and compliance in Amazon Neptune with AWS Backup, by Brian O'Keefe.	November 8, 2021
Blog post about fighting fraud and improper payments	See Fighting fraud and improper payments in real-time at the scale of federal expenditures by Vladi Royzman and Spencer Smith.	November 2, 2021
Blog post about preventing fake account sign-ups	See Prevent fake account sign-ups in real time with AI using Amazon Fraud Detector, by Anjan Biswas.	October 29, 2021

Engine version 1.0.5.1.R2	As of 2021-10-26, AWS is deploying engine version 1.0.5.1.R2 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.5.1.R2.	October 26, 2021
Blog post about HawkEye 360 predicting vessel risk using the Deep Graph Library	See HawkEye 360 predicts vessel risk using the Deep Graph Library and Amazon Neptune by Tim Pavlick, Ian Avilez, Dan Ford, and Gaurav Rele.	October 15, 2021
Engine version 1.0.5.1	As of 2021-10-01, AWS is deploying engine version 1.0.5.1 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.0.5.1.	October 1, 2021
Blog post about why developers like TinkerPop	See Why developers like Apache TinkerPop, an open source framework for graph computing, by Brad Bebee, Kelvin Lawrence, and Stephen Mallette.	September 27, 2021

Engine version 1.0.5.0.R3	As of 2021-09-15, AWS is deploying engine version 1.0.5.0.R3 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.5.0.R3.	September 15, 2021
Blog post about building a data discovery solution with Amundsen and Neptune	See Building a data discovery solution with Amundsen and Amazon Neptune, by Peter Hanssens and Don Simpson.	September 8, 2021
Neptune has updated the stream-poller to support non-string full-text search queries	Included in this release are many improvements to full-text search, including support for indexing of property values that are not strings. See Non-string OpenSearch indexing in Amazon Neptune.	August 23, 2021
Engine version 1.0.5.0.R2	As of 2021-08-16, AWS is deploying engine version 1.0.5.0.R2 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.5.0.R2.	August 16, 2021

Engine version 1.0.4.2.R5	As of 2021-08-16, AWS is deploying engine version 1.0.4.2.R5 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.4.2.R5.	August 16, 2021
Blog post about Graph Store Protocol support in Neptune	See Introducing Graph Store Protocol support for Amazon Neptune, by Chris Smith.	August 2, 2021
Blog post about new features in Neptune ML	See Discover more insights in your graphs with new features from Amazon Neptune ML, by Soji Adeshina.	July 30, 2021
Blog post about getting predictions faster with Neptune ML	See Get predictions for evolving graph data faster with Amazon Neptune ML, by Soji Adeshina.	July 30, 2021
Blog post about easier and faster graph machine learning with Neptune ML	See Easier and faster graph machine learning with Amazon Neptune ML, by Soji Adeshina.	July 30, 2021
Blog post about Neptune openCypher support	See Announcing openCypher for Amazon Neptune: Building better graph applications with openCypher and Gremlin together, by Brad Bebee.	July 29, 2021

Engine version 1.0.5.0	As of 2021-07-27, AWS is deploying engine version 1.0.5.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.0.5.0.	July 27, 2021
Engine version 1.0.4.2.R4	As of 2021-07-23, AWS is deploying engine version 1.0.4.2.R4 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.4.2.R4.	July 23, 2021
Neptune launched in China (Beijing)	Amazon Neptune is now available in China (Beijing) (`cn-north-1`).	July 21, 2021
Engine version 1.0.4.2.R3	As of 2021-06-28, AWS is deploying engine version 1.0.4.2.R3 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.4.2.R3.	June 28, 2021

Blog post about how Dream11 scaled their social network using Neptune	See Learn how Dream11, the World's largest fantasy sports platform, scale their social network with Amazon Neptune and Amazon ElastiCache.	June 25, 2021
Blog post about transforming data into knowledge with Neptune and PoolParty Semantic Suite	See Transform data into knowledge with PoolParty Semantic Suite and Amazon Neptune, by Ioanna Lytra and Albin Ahmeti.	June 16, 2021
Blog post about using Neptune to explore the UniProt knowledgebase	See Exploring the UniProt protein knowledgebase with AWS Open Data and Amazon Neptune, by Eric Greene, Rafa Xu, and Yuan Shi.	June 10, 2021
Blog post about using Neptune for data-driven risk analysis	See Field Notes: Data-Driven Risk Analysis with Amazon Neptune and Amazon OpenSearch Service, by Adriaan de Jonge and Rohit Satyanarayana.	June 10, 2021
Engine version 1.0.4.2.R2	As of 2021-06-01, AWS is deploying engine version 1.0.4.2.R2 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.4.2.R2.	June 1, 2021

Blog post about using Neptune to visualize your AWS infrastructure	See Visualize your AWS Infrastructure with Amazon Neptune and AWS Config, by Rohan Raizada and Amey Dhavle.	May 25, 2021
Blog post about configuration for using Data Lens with Neptune	See Configure AWS services to build a knowledge graph in Amazon Neptune using Data Lens, by Russell Waterson.	May 5, 2021
Blog post about building a knowledge graph in Neptune using Data Lens	See Build a knowledge graph in Amazon Neptune using Data Lens, by Russell Waterson.	May 5, 2021
Engine versions 1.0.1.0, 1.0.1.1, and 1.0.1.2 are now deprecated	Beginning now, no new DB instance will be created using any of these engine versions, or any patches related to them.	April 26, 2021
English translation of case study about Japan's Ministry of Economy, Trade and Industry using Neptune	See Japan's Ministry of Economy, Trade and Industry Powers gBizINFO Corporate Information Search Database with AWS.	March 31, 2021
Blog post about using Neptune with Amazon Comprehend and Lex	See Supercharge your knowledge graph using Amazon Neptune, Amazon Comprehend, and Amazon Lex, by Dave Bechberger.	March 31, 2021
Blog post about using Lambda functions with Neptune	See Use AWS Lambda functions with Amazon Neptune, by Ian Robinson.	March 26, 2021

Engine version 1.0.4.1.R1.1	As of 2021-03-22, AWS is deploying engine version 1.0.4.1.R1.1 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.4.1.R1.1.	March 22, 2021
Engine version 1.0.4.1.R2.1	As of 2021-03-11, AWS is deploying engine version 1.0.4.1.R2.1 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.4.1.R2.1.	March 11, 2021
Blog post about using Neptune's open source graph notebook for graph visualization	See Getting started with open source graph notebook for graph visualization, by Joy Wang, Ora Lassila, and Stephen Mallette.	March 10, 2021
Tutorial about integrating Neptune with the Amundsen data discovery and metadata engine	See How to use Amundsen with Amazon Neptune, by Andrew Ciambrone.	March 2, 2021

Engine version 1.0.4.1.R2	As of 2021-02-24, AWS is deploying engine version 1.0.4.1.R2 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.4.1.R2.	February 24, 2021
Engine version 1.0.4.0.R2	As of 2021-02-24, AWS is deploying engine version 1.0.4.0.R2 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.4.0.R2.	February 24, 2021
Engine version 1.0.3.0.R3	As of 2021-02-19, AWS is deploying engine version 1.0.3.0.R3 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.3.0.R3.	February 19, 2021

Engine version 1.0.2.2.R6	As of 2021-02-19, AWS is deploying engine version 1.0.2.2.R6 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.2.2.R6.	February 19, 2021
Blog post about building a knowledge graph using Amazon Comprehend events	See Building a knowledge graph in Amazon Neptune using Amazon Comprehend Events, by Brian O'Keefe, Graham Horwood, and Navtanay Sinha.	January 19, 2021
Blog post about enabling low code graph data apps	See Enabling low code graph data apps with Amazon Neptune and Graphistry, by Leo Meyerovich, Dave Bechberger, and Taylor Riggan.	January 18, 2021
Added notebook documentation for getting started with graph data.	Added a section integrating with the Neptune workbench that helps you get started creating graph data and developing graph applications without having to spin up a Neptune cluster until you're ready.	January 15, 2021
Blog post about resetting your Neptune graph data in seconds	See Resetting your graph data in Amazon Neptune in seconds, by Niraj Jetly and Navtanay Sinha.	December 17, 2020

Blog post about how Novartis AG uses SageMaker and Neptune with BERT	See Novartis AG uses Amazon SageMaker and Amazon Neptune to build and enrich a knowledge graph using BERT, by Othmane Hamzaoui, Fatema Alkhanaizi, and Viktor Malesevic.	December 14, 2020
Blog post about building a knowledge graph with topic networks	See Building a knowledge graph with topic networks in Amazon Neptune, by Edward Brown, Head of AI Projects, Eduardo Piairo, Architect, Marcia Oliveira, Lead Data Scientist, and Jack Hampson, CEO at Deeper Insights.	December 14, 2020
Engine version 1.0.4.1	As of 2020-12-08, AWS is deploying engine version 1.0.4.1 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.0.4.1.	December 8, 2020
Blog post about getting started with Neptune ML	See How to get started with Neptune ML, by George Karypis, Dave Bechberger, and Karthik Bharathy.	December 8, 2020
Neptune now has a fast reset API	Using the fast-reset API, you can quickly and easily delete all the data in a DB cluster. See Fast reset API.	December 4, 2020

Blog post about building a biological knowledge graph at Pendulum	See Building a biological knowledge graph at Pendulum using Amazon Neptune, by Connor Skennerton.	November 26, 2020
Blog post about TinkerPop 3.4.8's new features in Neptune	See Exploring Apache TinkerPop 3.4.8's new features in Amazon Neptune, by Stephen Mallette.	November 18, 2020
Blog post about using the Amazon Kendra search service with Neptune	See Incorporating your enterprise knowledge graph into Amazon Kendra, by Yazdan Shirvany, Mohit Mehta, and Dipto Chakravarty.	November 17, 2020
Event notifications now available	Neptune now supports event notifications that you can use to more easily monitor DB clusters. See Using Neptune Event Notification, by.	October 29, 2020
Customs endpoints now available	Neptune now supports custom endpoints for greater control connecting to DB instances. See Connecting to Amazon Neptune Endpoints.	October 29, 2020
Blog post about using AWS Database Migration Service (DMS) to populate your Neptune graph	See Populating your graph in Amazon Neptune from a relational database using AWS Database Migration Service (DMS) – Part 4: Putting it all together, by Chris Smith.	October 22, 2020

Blog post about using AWS Database Migration Service (DMS) to populate your Neptune graph	See Populating your graph in Amazon Neptune from a relational database using AWS Database Migration Service (DMS) – Part 3: Designing the RDF Model, by Chris Smith.	October 22, 2020
Blog post about using AWS Database Migration Service (DMS) to populate your Neptune graph	See Populating your graph in Amazon Neptune from a relational database using AWS Database Migration Service (DMS) – Part 2: Designing the property graph model, by Chris Smith.	October 22, 2020
Blog post about using AWS Database Migration Service (DMS) to populate your Neptune graph	See Populating your graph in Amazon Neptune from a relational database using AWS Database Migration Service (DMS) – Part 1: Setting the stage, by Chris Smith.	October 22, 2020
Engine version 1.0.4.0	As of 2020-10-12, AWS is deploying engine version 1.0.4.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.0.4.0.	October 12, 2020

Engine version 1.0.3.0.R2	As of 2020-10-12, AWS is deploying engine version 1.0.3.0.R2 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.3.0.R2.	October 12, 2020
Engine version 1.0.2.2.R5	As of 2020-10-12, AWS is deploying engine version 1.0.2.2.R5 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.2.2.R5.	October 12, 2020
Blog post about configuring your VPC for SPARQL federated queries	See Configure Amazon VPC for SPARQL 1.1 Federated Query with Amazon Neptune, by Charles Ivie.	October 12, 2020
Blog post about writing a SPARQL cascading delete	See Write a cascading delete in SPARQL, by Ora Lassila.	October 5, 2020
Blog post about graphing AWS resources using Neptune	See Graph your AWS resources with Amazon Neptune, by Dave Bechberger.	September 28, 2020

Blog post about building MedDRA terminology mapping for pharmacov igilance and adverse event reporting using Neptune	See Building Amazon Neptune based MedDRA terminology mapping for pharmacovigilance and adverse event reporting, by Vaijayanti Joshi, Deven Atnoor, Ph.D, and Sudhanshu Malhotra.	September 24, 2020
Blog post about building a knowledge graph out of a data warehouse using Neptune, to complement commercial intelligence	See Complement Commercia l Intelligence by Building a Knowledge Graph out of a Data Warehouse with Amazon Neptune, by Shahria Hossain and Mikael Graindorge.	September 23, 2020
Blog post about load balancing using the Neptune Gremlin client	See Load balance graph queries using the Amazon Neptune Gremlin Client, by Ian Robinson.	September 16, 2020
Blog post about digital personalization using an identity graph at Cox Automotive	See Cox Automotive scales digital personalization using an identity graph powered by Amazon Neptune, by Carlos Rendon and Niraj Jetly.	September 16, 2020
Blog post about collaborative filtering on Yelp data	See Using collaborative filtering on Yelp data to build a recommendation system in Amazon Neptune, by Chad Tindel.	September 8, 2020
Blog post about query-res ult visualization in Amazon Neptune	See Visualize query results using the Amazon Neptune workbench, by Kelvin Lawrence.	September 2, 2020

Neptune released graph visualization	Amazon Neptune now provides extensive graph visualization capabilities in Jupyter notebooks in the Neptune workbench, along with a number of new features that make notebooks easier to use. See Graph visualization.	August 12, 2020
Neptune launched in South America (São Paulo)	Amazon Neptune is now available in South America (São Paulo) (`sa-east-1`).	August 6, 2020
Neptune launched in Asia Pacific (Hong Kong)	Amazon Neptune is now available in Asia Pacific (Hong Kong) (`ap-east-1`).	August 6, 2020
Engine version 1.0.3.0	As of 2020-08-03, AWS is deploying engine version 1.0.3.0 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.0.3.0.	August 3, 2020
Engine version 1.0.2.2.R4	As of 2020-07-23, AWS is deploying engine version 1.0.2.2.R4 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.2.2.R4.	July 23, 2020

Blog post about Zerobase's automated contact tracing using Amazon Neptune	See Zerobase creates private, secure, and automated contact tracing using Amazon Neptune, by David Harris and Aron Szanto.	July 13, 2020
Neptune launched in US West (N. California)	Amazon Neptune is now available in US West (N. California) (`us-west-1`).	July 9, 2020
Amazon Neptune supports tag-based access control	You can now use AWS tags in IAM policies to control access to your Neptune database. See Tag-based access control in Amazon Neptune.	July 7, 2020
A Java stream poller is now available	Amazon Neptune now supports a Java version of the lambda stream poller for Neptune streams as well as the Python one. See Add details about the Neptune streams consumer stack you are creating.	July 6, 2020
Blog post about the AWS COVID-19 knowledge graph	See Building and querying the AWS COVID-19 knowledge graph, by Ninad Kulkarni, Colby Wise, George Price, and Miguel Romero.	July 1, 2020

Engine version 1.0.1.1	As of 2020-06-26, AWS is deploying engine version 1.0.1.1 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.0.1.1.	June 26, 2020
Blog post about migrating from Blazegraph to Amazon Neptune	See Moving to the cloud: Migrating Blazegraph to Amazon Neptune, by Dave Bechberger.	June 25, 2020
Blog post about changing data capture from Neo4j to Amazon Neptune	See Change data capture from Neo4j to Amazon Neptune using Amazon Managed Streaming for Apache Kafka, by Sanjeet Sahay.	June 22, 2020
Blog post about how Waves is using Amazon Neptune	See How Waves runs user queries and recommend ations at scale with Amazon Neptune, by Pavel Vasilyev.	June 16, 2020
Engine version 1.0.1.2	As of 2020-06-10, AWS is deploying engine version 1.0.1.2 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.0.1.2.	June 10, 2020

Blog post about building a customer knowledge repository	See Building a customer 360 knowledge repository with Amazon Neptune and Amazon Redshift, by Ram Bhandarkar.	June 9, 2020
Blog post about how Gunosy is using Amazon Neptune	See How Gunosy built a comment feature in News Pass using Amazon Neptune by Yosuke Uchiyama.	June 8, 2020
Blog post about the AWS COVID-19 knowledge graph	See Building and querying the AWS COVID-19 knowledge graph by Ninad Kulkarni, Colby Wise, George Price, and Miguel Romero.	June 2, 2020
Blog post about exploring COVID-19 research using Amazon Neptune	See Exploring scientific research on COVID-19 with Amazon Neptune, Amazon Comprehend Medical, and the Tom Sawyer Graph Database Browser by George Price, Colby Wise, Miguel Romero, and Ninad Kulkarni.	June 2, 2020
You can now load data into Neptune using AWS DMS	See Using AWS Database Migration Service to load data into Amazon Neptune from a different data store.	June 1, 2020

Engine version 1.0.2.0 is being deprecated	Amazon Neptune engine version 1.0.2.0 is now deprecated. Clusters running on this engine version will be upgraded to version 1.0.2.1 automatically during the first maintenance window following June 1, 2020.	May 19, 2020
Blog post about building a customer identity graph using Neptune	See Building a customer identity graph with Amazon Neptune by Rajesh Wunnava and Taylor Riggan.	May 12, 2020
Engine version 1.0.2.0.R3	As of 2020-05-05, AWS is deploying engine version 1.0.2.0.R3 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.2.0.R3.	May 5, 2020
Engine version 1.0.2.1.R6	As of 2020-04-22, AWS is deploying engine version 1.0.2.1.R6 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.2.1.R6.	April 22, 2020

Blog post about migrating data from Neo4j to Neptune	See Migrating a Neo4j graph database to Amazon Neptune with a fully automated utility by Sanjeet Sahay.	April 13, 2020
Blog post about lowering the cost of building graph applications with Neptune	See Lower the cost of building graph apps by up to 76% with Amazon Neptune T3 instances by Karthik Bharathy and Brad Bebee.	April 9, 2020
Neptune offers a T3 burstable instance class	You can now create an Amazon Neptune T3 burstable instance for cost-effective developme nt and testing purposes. See Neptune T3 Burstable Instance Class.	April 8, 2020
Engine version 1.0.2.2.R2	As of 2020-04-02, AWS is deploying engine version 1.0.2.2.R2 to all Regions. It takes several days for a new release to become available in every Region. For more information about this engine version, see Neptune Engine Release 1.0.2.2.R2.	April 2, 2020
Blog post about graphing investment dependency at EDGAR	See Graphing investment dependency with Amazon Neptune by Lawrence Verdi.	March 17, 2020
Neptune launched in Europe (Paris)	Amazon Neptune is now available in Europe (Paris) (eu-west-3).	March 11, 2020

Engine version 1.0.2.2	As of 2020-03-09, AWS is deploying engine version 1.0.2.2 to all Regions. It takes several days for a new release to become available in every Region. For more informati on about this engine version, see Neptune Engine Release 1.0.2.2.	March 9, 2020
Stopping and Restarting a DB Cluser	You can now stop a DB Cluster for 7 days using the Neptune console, and later restart it when you need it again. While your DB cluster is stopped, you are charged only for cluster storage, manual snapshots, and automated backup storage, but not for any DB instance hours. See Stopping and Starting an Amazon Neptune DB Cluster.	February 19, 2020
Video about a social graph at Nike	Listen in as Todd Escalona of AWS talks with Marc Wangenheim, Senior Engineering Manager at Nike, about how the company powers a number of applicati ons via a social graph built on Amazon Neptune. See Nike: A Social Graph at Scale with Amazon Neptune.	February 11, 2020

Neptune clusters can now be configured to require SSL connections	In regions that still support HTTP connections, SSL is now turned on by default in all new parameter groups. There are no changes to existing parameter groups, but you can force clients to use SSL by changing the `neptune_enforce_ssl` parameter to 1. See Encryption in Transit: Connecting to Neptune Using SSL/HTTPS for information about how to enable HTTP connections for a cluster in a region that still supports them. See Parameters That You Can Use to Configure Amazon Neptune for a description of cluster and instance parameters.	February 10, 2020
You can now specify engine version and deletion protection in Neptune's CloudFormation template	Amazon Neptune has updated its CloudFormation template to include an `AWS::Neptune::DBCluster.EngineVersion` parameter that lets you specify a particular engine version for your new DB cluster, and an `AWS::Neptune::DBCluster.DeletionProtection` parameter that lets you turn on deletion protection for it.	February 9, 2020

Deletion protection	Amazon Neptune has delivered deletion protection for DB clusters and instances . As long as deletion protectio n is enabled on a DB cluster or instance, you cannot delete it. See You cannot delete a DB Instance if Deletion Protectio n is enabled.	January 20, 2020
Neptune launched in China (Ningxia)	Amazon Neptune is now available in China (Ningxia) (cn-northwest-1).	January 15, 2020
Engine version 1.0.2.1.R4	Patch R4 for engine version 1.0.2.1 is generally available . For more information, see Neptune Engine Release 1.0.2.1.R4.	December 20, 2019
Engine version 1.0.2.1.R3	Patch R3 for engine version 1.0.2.1 is generally available . For more information, see Neptune Engine Release 1.0.2.1.R3.	December 12, 2019
Blog post about using Neptune to analyze social media feeds	See Analyzing social media feeds using Amazon Neptune.	November 27, 2019
Engine version 1.0.2.1.R2	Patch R2 for engine version 1.0.2.1 is generally available . For more information, see Neptune Engine Release 1.0.2.1.R2.	November 25, 2019

Engine version 1.0.2.1.R1	Amazon Neptune engine version 1.0.2.1.R1 is generally available. For more information, see Neptune Engine Release 1.0.2.1.	November 22, 2019
Engine version 1.0.2.0.R2	Patch R2 for engine version 1.0.2.0 is generally available. For more information, see Neptune Engine Release 1.0.2.0.R2.	November 21, 2019
Blog post about Neptune sessions and workshops at re:Invent 2019	See Your guide to Amazon Neptune sessions, workshops, and chalk talks at AWS re:Invent 2019.	November 20, 2019
Engine version 1.0.2.0.R1	Amazon Neptune engine version 1.0.2.0.R1 is generally available. For more information, see Neptune Engine Release 1.0.2.0.	November 8, 2019
Blog post about capturing graph changes using Neptune Streams	See Capture Graph Changes using Neptune Streams.	November 6, 2019
Engine version 1.0.1.0.200502.0	Amazon Neptune engine version 1.0.1.0.200502.0 is generally available. For more information, see Update 1.0.1.0.200502.0.	October 31, 2019
Neptune launched in Middle East (Bahrain)	Amazon Neptune is now available in Middle East (Bahrain) (me-south-1).	October 30, 2019

Neptune launched in Canada (Central)	Amazon Neptune is now available in Canada (Central) (ca-central-1).	October 30, 2019
Blog post about Neptune's new SPARQL Streams feature and SPARQL federated query support	See Amazon Neptune releases Streams, SPARQL federated query for graphs and more.	October 17, 2019
Engine version 1.0.1.0.200463.0	Amazon Neptune engine version 1.0.1.0.200463.0 is generally available. For more information, see Update 1.0.1.0.200463.0.	October 15, 2019
Engine version 1.0.1.0.200457.0	Amazon Neptune engine version 1.0.1.0.200457.0 is generally available. For more information, see Update 1.0.1.0.200457.0.	September 19, 2019
Blog post about Neptune's new SPARQL explain feature	See Using SPARQL explain to understand query execution in Amazon Neptune.	September 17, 2019
Blog post about Neptune support for TinkerPop 3.4	See Amazon Neptune now supports TinkerPop 3.4 features.	September 6, 2019
Blog post about using Neptune with PyTorch on Amazon SageMaker	See A personalized 'shop-by-style' experience using PyTorch on Amazon SageMaker and Amazon Neptune.	August 22, 2019

Blog post about using Neptune with AWS AppSync and Amazon Elasticache	See Integrating alternative data sources with AWS AppSync: Amazon Neptune and Amazon ElastiCache.	August 22, 2019
Neptune launched in AWS GovCloud (US-East)	Amazon Neptune is now available in AWS GovCloud (US-East) (us-gov-east-1).	August 21, 2019
Neptune launched in AWS GovCloud (US-West)	Amazon Neptune is now available in AWS GovCloud (US-West) (us-gov-west-1).	August 14, 2019
Engine version 1.0.1.0.200369.0	Amazon Neptune engine version 1.0.1.0.200369.0 is generally available. For more information, see Update 1.0.1.0.200369.0.	August 13, 2019
Engine version 1.0.1.0.200366.0	Amazon Neptune engine version 1.0.1.0.200366.0 is generally available. For more information, see Update 1.0.1.0.200366.0.	July 26, 2019
Blog post about using Neptune with PyTorch on Amazon SageMaker	See A personalized 'shop-by-style' experience using PyTorch on Amazon SageMaker and Amazon Neptune.	July 3, 2019
Engine version 1.0.1.0.200348.0	Amazon Neptune engine version 1.0.1.0.200348.0 is generally available. For more information, see Update 1.0.1.0.200348.0.	July 2, 2019

Neptune launched in Europe (Stockholm)	Amazon Neptune is now available in Europe (Stockholm) (eu-north-1).	June 27, 2019
Neptune can now publish audit logs to CloudWatch Logs	For more information, see Publishing Neptune Logs to Amazon CloudWatch Logs.	June 18, 2019
Engine version 1.0.1.0.200310.0	Amazon Neptune engine version 1.0.1.0.200310.0 is generally available. For more information, see Update 1.0.1.0.200310.0.	June 12, 2019
Blog post about LifeOmic's JupiterOne	See How LifeOmic's JupiterOne simplifies security and compliance operations with Amazon Neptune.	May 2, 2019
Neptune launched in Asia Pacific (Seoul)	Amazon Neptune is now available in Asia Pacific (Seoul) (ap-northeast-2).	May 1, 2019
Engine version 1.0.1.0.200296.0	Amazon Neptune engine version 1.0.1.0.200296.0 is generally available. For more information, see Update 1.0.1.0.200296.0.	May 1, 2019
Neptune launched in Asia Pacific (Mumbai)	Amazon Neptune is now available in Asia Pacific (Mumbai) (ap-south-1).	March 6, 2019
Blog post about Gremlin query hints	See Introducing Gremlin query hints for Amazon Neptune.	February 26, 2019

Neptune launched in Asia Pacific (Tokyo)	Amazon Neptune is now available in Asia Pacific (Tokyo) (ap-northeast-1).	January 23, 2019
CloudFormation template for creating an AWS Lambda function to access Neptune	Updated the getting started section and added an CloudFormation template to create a Lambda function to use with Neptune. For more information, see Getting Started with Neptune.	January 23, 2019
Engine version 1.0.1.0.200267.0	Amazon Neptune engine version 1.0.1.0.200267.0 is generally available. For more information, see Update 1.0.1.0.200267.0.	January 21, 2019
Neptune launched in Asia Pacific (Sydney)	Amazon Neptune is now available in Asia Pacific (Sydney) (ap-southeast-2).	January 9, 2019
Blog post about using Metaphactory	See Exploring Knowledge Graphs on Amazon Neptune Using Metaphactory.	January 9, 2019
Neptune launched in Asia Pacific (Singapore)	Amazon Neptune is now available in Asia Pacific (Singapore) (ap-southeast-1).	December 13, 2018
Engine version 1.0.1.0.200264.0	Amazon Neptune engine version 1.0.1.0.200264.0 is generally available. For more information, see Update 1.0.1.0.200264.0.	November 19, 2018
Amazon Neptune SSL Support	Neptune now supports SSL connections.	November 19, 2018

Consolidated Error topics	All error message and code information are now in a single topic.	November 15, 2018
Updated Getting Started Topic	Updated Getting Started topic with additional links and reorganized documentation.	November 14, 2018
Engine version 1.0.1.0.200258.0	Amazon Neptune engine version 1.0.1.0.200258.0 is generally available. For more information, see Update 1.0.1.0.200258.0.	November 8, 2018
Neptune launched in Europe (Frankfurt)	Amazon Neptune is now available in Europe (Frankfurt) (eu-central-1).	November 7, 2018
Blog post #1 in a series	See Let Me Graph That For You – Part 1 – Air Routes.	November 7, 2018
Blog post about using Amazon SageMaker Jupyter Notebooks	See Analyze Amazon Neptune Graphs using Amazon SageMaker Jupyter Notebooks.	November 1, 2018
Engine version 1.0.1.0.200255.0	Amazon Neptune engine version 1.0.1.0.200255.0 is generally available. For more information, see Update 1.0.1.0.200255.0.	October 29, 2018
Neptune launched in Europe (London)	Amazon Neptune is now available in Europe (London) (eu-west-2).	October 3, 2018

Engine version 1.0.1.0.200237.0	Amazon Neptune engine version 1.0.1.0.200237.0 is generally available. For more information, see Update 1.0.1.0.200237.0.	September 6, 2018
Engine version 1.0.1.0.200236.0	Amazon Neptune engine version 1.0.1.0.200236.0 is generally available. For more information, see Update 1.0.1.0.200236.0.	July 24, 2018
Engine version 1.0.1.0.200233.0	Amazon Neptune engine version 1.0.1.0.200233.0 is generally available. For more information, see Update 1.0.1.0.200233.0.	June 22, 2018
New Neptune Quick Start	Updated quick start with CloudFormation and the Gremlin Console tutorial. For more information, see Amazon Neptune Quick Start Using CloudFormation.	June 19, 2018
Amazon Neptune initial release	This is the initial release of the Neptune User Guide. See also the release blog post, Amazon Neptune Generally Available.	May 30, 2018
Introductory Neptune Blog Post	See Amazon Neptune – A Fully Managed Graph Database Service.	November 29, 2017

Getting started with Amazon Neptune

Amazon Neptune is a fully managed graph database service that scales to handle billions of relationships and lets you query them with milliseconds latency, at a low cost for that kind of capacity.

If you're looking for more detailed information about Neptune, see Overview of Amazon Neptune features.

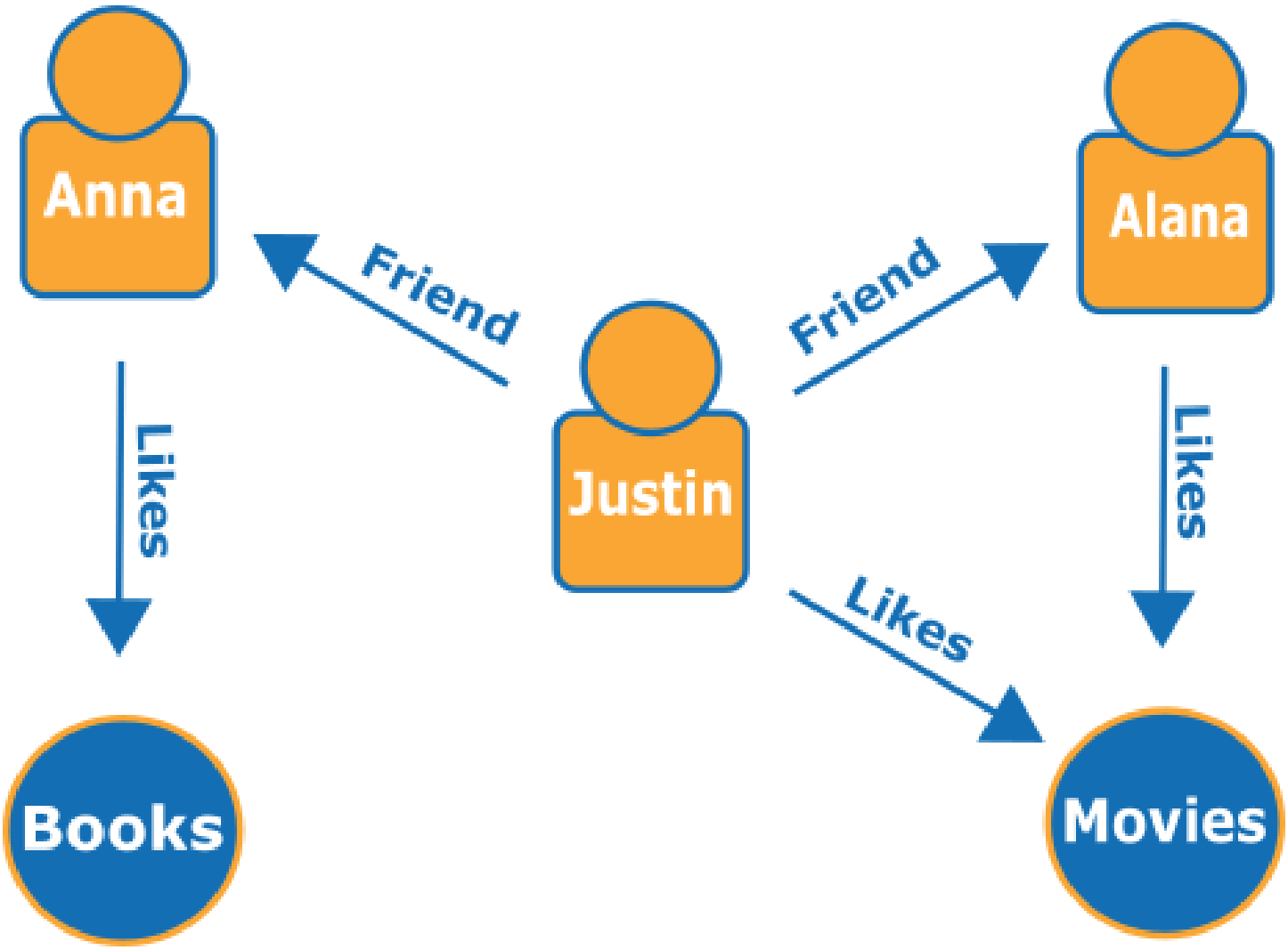
Otherwise, you might want to know a little more about graph databases before you start.

Graph database key concepts

Graph databases are optimized to store and query the *relationships* between data items.

They store data items themselves as *vertices* of the graph, and the relationships between them as *edges*. Each edge has a type, and is directed from one vertex (the start) to another (the end). Relationships can be called *predicates* as well as edges, and vertices are also sometimes referred to as *nodes*. In so-called property graphs, both vertices and edges can have additional *properties* associated with them too.

Here is a small graph representing friends and hobbies in a social network:



The edges are shown as named arrows, and the vertices represent specific people and hobbies that they connect.

A simple traversal of this graph can tell you what Justin's friends like.

Why use a graph database?

Whenever connections or relationships between entities are at the core of the data that you're trying to model, a graph database is your natural choice.

For one thing, it's easy to model data interconnections as a graph, and then write complex queries that extract real-world information from the graph.

Building an equivalent application using a relational database requires you to create many tables with multiple foreign keys and then write nested SQL queries and complex joins. Not only does

that approach quickly become unwieldy from a coding perspective, its performance degrades quickly as the amount of data increases.

By contrast, a graph database like Neptune can query relationships between billions of vertices without bogging down.

What can you do with a graph database?

Graphs can represent the interrelationships of real-world entities in many ways, in terms of actions, ownership, parentage, purchase choices, personal connections, family ties, and so on.

Here are some of the most common areas where graph databases are used:

- **Knowledge graphs** – Knowledge graphs let you organize and query all kinds of connected information to answer general questions. Using a knowledge graph, you can add topical information to product catalogs, and model diverse information such as is contained in Wikidata.

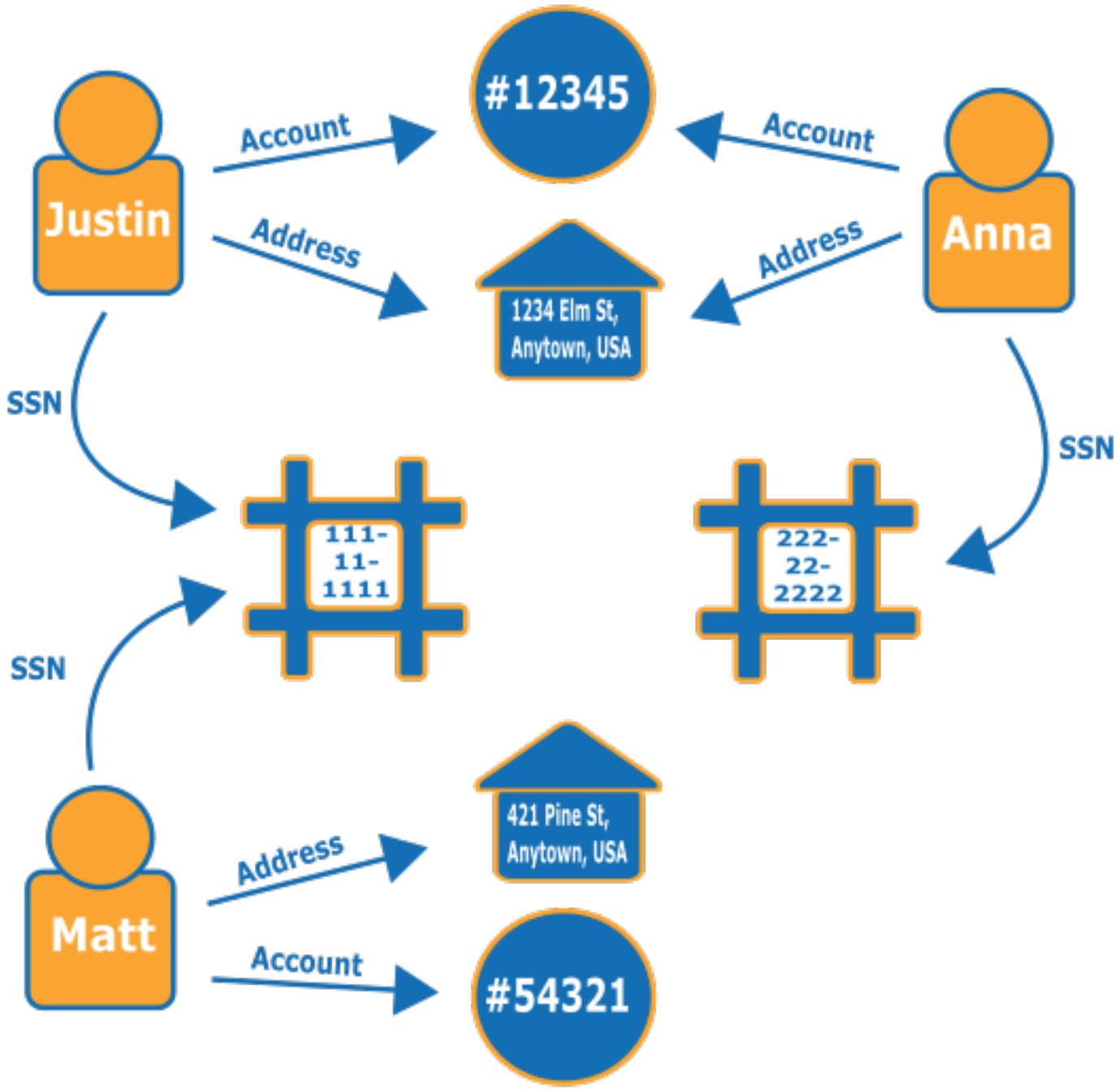
  To learn more about how knowledge graphs work and where they are being used, see Knowledge Graphs on AWS.
- **Identity graphs** – In a graph database, you can store relationships between information categories such as customer interests, friends, and purchase history, and then query that data to make recommendations which are personalized and relevant.

  For example, you can use a graph database to make product recommendations to a user based on which products are purchased by others who follow the same sport and have a similar purchase history. Or, you can identify people who have a friend in common but don't yet know each other, and make a friendship recommendation.

  Graphs of this kind are known as identity graphs, and are widely used for personalizing interactions with users. To find out more, see Identity Graphs on AWS. To get started building your own identity graph, you can begin with the Identity Graph Using Amazon Neptune sample.
- **Fraud graphs** – This is a common use for graph databases. They can help you track credit card purchases and purchase locations to detect uncharacteristic use, or to detect a purchaser is trying to use the same email address and credit card as was used in a known fraud case. They can let you check for multiple people associated with a personal email address, or multiple people in different physical locations who share the same IP address.

  Consider the following graph. It shows the relationship of three people and their identity-related information. Each person has an address, a bank account, and a social security number. However,

we can see that Matt and Justin share the same social security number, which is irregular and indicates possible fraud by one of them. A query to a fraud graph can reveal connections of this kind so that they can be reviewed.



To learn out more about fraud graphs and where they are being used, see Fraud Graphs on AWS.

- **Social networking** – One of the first and most common areas where graph databases are used is in social networking applications.

  For example, suppose that you want to build a social feed into a web site. You can easily use a graph database on the back end to deliver results to users that reflect the latest updates from their families, their friends, from people whose updates they "like," and from people who live close to them.
- **Driving directions** – A graph can help find the best route from a starting point to a destination, given current traffic and typical traffic patterns.
- **Logistics** – Graphs can help identify the most efficient way to use available shipping and distribution resources to meet customer requirements.
- **Diagnostics** – Graphs can represent complex diagnostic trees that can be queried to identify the source of observed problems and failures.
- **Scientific research** – With a graph database, you can build applications that store and navigate scientific data and even sensitive medical information using encryption at rest. For example, you can store models of disease and gene interactions. You can search for graph patterns within protein pathways to find other genes that might be associated with a disease. You can model chemical compounds as a graph and query for patterns in molecular structures. You can correlate patient data from medical records in different systems. You can topically organize research publications to find relevant information quickly.
- **Regulatory rules** – You can store complex regulatory requirements as graphs, and query them to detect situations where they might apply to your day-to-day business operations.
- **Network topology and events** – A graph database can help you manage and protect an IT network. When you store the network topology as a graph, you can also store and process many different kinds of events on the network. You can answer questions such as how many hosts are running a given application. You can query for patterns that might show that a given host has been compromised by a malicious program, and query for connection data that can help trace the program to the original host that downloaded it.

How do you query a graph?

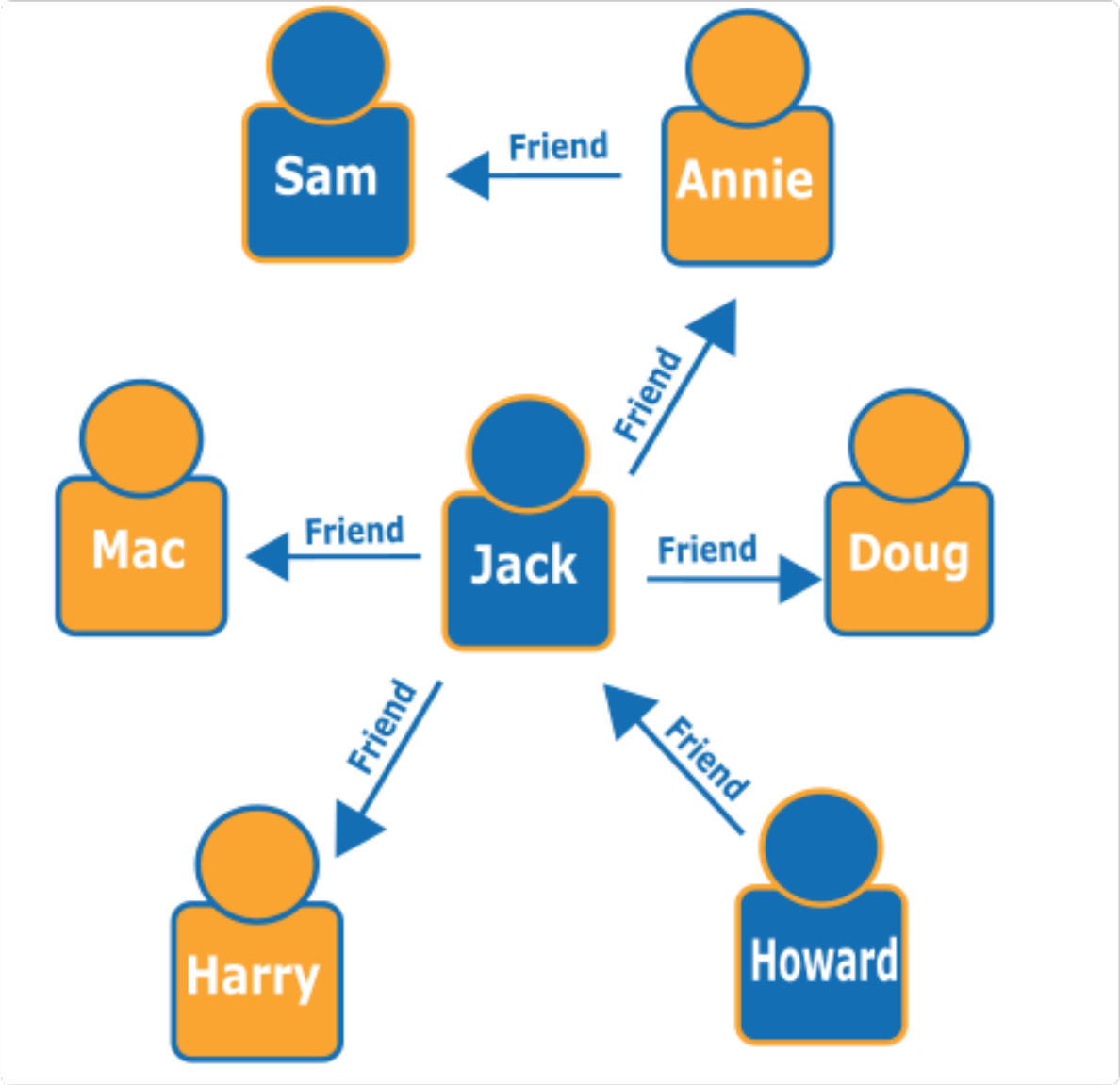
Neptune supports three special-purpose query languages designed for querying graph data of different kinds. You can use these languages to add, modify, delete and query data in a Neptune graph database:

- Gremlin is a graph traversal language for property graphs. A query in Gremlin is a traversal made up of discrete steps, each of which follows an edge to a node. See Gremlin documentation at Apache TinkerPop for more information.

  The Neptune implementation of Gremlin has some differences from other implementations, especially when you are using Gremlin-Groovy (Gremlin queries sent as serialized text). For more information, see Gremlin standards compliance in Amazon Neptune.
- openCypher – openCypher is a declarative query language for property graphs that was originally developed by Neo4j, then open-sourced in 2015, and contributed to the openCypher project under an Apache 2 open-source license. See the Cypher Query Language Reference (Version 9) for the language specification, as well as the Cypher Style Guide for additional information.
- SPARQL is a declarative query language for RDF data, based on the graph pattern matching that is standardized by the World Wide Web Consortium (W3C) and described in SPARQL 1.1 Overview) and the SPARQL 1.1 Query Language specification. See SPARQL standards compliance in Amazon Neptune for specific details about the Neptune implementation of SPARQL.

Examples of matching Gremlin and SPARQL queries

Given the following graph of people (nodes) and their relationships (edges), you can find out who the "friends of friends" of a particular person are— for example, the friends of Howard's friends.



Looking at the graph, you can see that Howard has one friend, Jack, and Jack has four friends: Annie, Harry, Doug, and Mac. This is a simple example with a simple graph, but these types of queries can scale in complexity, dataset size, and result size.

Here is a Gremlin traversal query that returns the names of the friends of Howard's friends:

```
g.V().has('name', 'Howard').out('friend').out('friend').values('name')
```

Here is a SPARQL query that returns the names of the friends of Howard's friends:

```
prefix : <#>

  select ?names where {
    ?howard :name "Howard" .
    ?howard :friend/:friend/:name ?names .
  }
```

> **Note**
>
> Each part of any Resource Description Framework (RDF) triple has a URI associated with it. In the example above, the URI prefix is intentionally short.

Take an online course in using Amazon Neptune

If you like learning with videos, AWS offers online courses in the AWS Online Tech Talks to help you get going. One that introduces graph databases is:

Graph Database introduction, deep-dive and demo with Amazon Neptune.

Digging deeper into graph reference architecture

As you think about what problems a graph database could solve for you, and how to approach them, one of the best places to start is the Neptune graph reference architectures GitHub project.

There you can find detailed descriptions of graph workload types, and three sections to help you design an effective graph database:

- Data Models and Query Languages – This section walks you through the differences between Gremlin and SPARQL and how to choose between them.
- Graph Data Modeling – This is a thorough discussion of how to make graph data modeling decisions, including detailed walkthroughs of property graph modeling using Gremlin and RDF modeling using SPARQL.
- Converting Other Data Models to a Graph Model – Here you can find out how to go about translating a relational data model into a graph model.

There are also three sections that walk you through specific steps for using Neptune:

- Connecting to Amazon Neptune from Clients Outside the Neptune VPC – This section shows you several options for connecting to Neptune from outside the VPC where your DB cluster is located.
- Accessing Amazon Neptune from AWS Lambda Functions – Here you'll find out how to connect reliably to Neptune from Lambda functions.
- Writing to Amazon Neptune from an Amazon Kinesis Data Stream – This section can help you handle high write throughput scenarios with Neptune.

Quick start using AWS CloudShell

This guide helps you get started quickly by using AWS CloudShell to connect to and query your Neptune cluster directly from the AWS Management Console.

Time to complete: Approximately 2 minutes

Cost: CloudShell is available at no additional charge. Creating a Neptune cluster incurs charges. For pricing details, see Amazon Neptune pricing.

Prerequisites

- An AWS account. If you don't have one, see Setting up your AWS account.
- A Neptune cluster with status **Available**. If you don't already have one, see Launching a Neptune DB cluster using the AWS Management Console to create one.

Step 1: Connect to your Neptune cluster

Connect to your Neptune cluster using AWS CloudShell. CloudShell automatically provisions a shell environment connected to your cluster's VPC. You don't need to configure security groups or set up notebooks to interact with the database.

To connect to your cluster

1. In the Neptune console, under **Clusters**, locate the cluster you created.
2. Select the check box next to your cluster.
3. Choose **Connect to cluster**.

> **Note**
>
> This button becomes active after you select the check box next to your cluster and the cluster status shows as **Available**.

The CloudShell **Run command** screen appears.

4. Choose **Run** to connect to the cluster. Provisioning the VPC environment takes a few seconds.

After you connect, the `graphsh` console appears, providing an interactive shell for running graph queries.

Step 2: Insert and query data

Now that you are connected to your cluster, run a few queries to get familiar with Amazon Neptune. The following examples use openCypher, one of the query languages supported by Neptune.

To insert nodes and relationships

Run the following query to create two person nodes and a relationship:

```
CREATE (a:Person {name: 'Alice', age: 30}), (b:Person {name: 'Bob', age: 40}), (a)-
[:KNOWS]->(b)
```

The output looks similar to the following:

```
##########
# result #
##########
# []     #
##########
```

To find nodes

Run the following query to return all person names in alphabetical order:

```
MATCH (p:Person) RETURN p.name AS name ORDER BY name
```

The output looks similar to the following:

```
#########
# name  #
#########
# Alice #
# Bob   #
#########
```

Next steps

You have completed this quick start. To continue learning, explore the following features:

- Overview of Amazon Neptune features – Learn about the key capabilities of Amazon Neptune.
- Amazon Neptune Serverless – Automatically scale your cluster capacity based on workload.
- Managing Your Amazon Neptune Database – Manage your clusters, instances, and configurations.

To avoid ongoing charges, delete the Neptune cluster if you created it only for this quick start. For instructions, see Managing Your Amazon Neptune Database.

Troubleshooting Amazon Neptune notebook issues

When working with Neptune notebooks in JupyterLab, you may occasionally run into issues related to browser behavior, JupyterLab itself, or Amazon SageMaker AI integration. The following sections describe the most common problems and how to resolve them.

Widget state desynchronization

Graph-notebook widgets can sometimes become desynchronized between the JupyterLab UI and the kernel backend. When this happens, you may see widget display errors or an error message like the following:

```
Model class 'OutputModel' from module '@jupyter-widgets/output' is loaded but cannot be
 instantiated
```

This typically occurs after long idle periods, when switching between notebook tabs, after clearing the browser cache, during unstable network connectivity, or when reopening a notebook that has unsaved widget outputs.

To resolve this, choose **Restart Kernel and Clear All Outputs** from the **Kernel** menu, and then re-run your notebook cells.

JupyterLab 4.x initial loading screen delay

When opening a notebook for the first time after JupyterLab starts, you may see a persistent loading screen with the message: `Loading... The loading screen is taking a long time. Would you like to clear the workspace or keep waiting?`

This primarily affects Firefox and only occurs on the first notebook open in a session. You can resolve it by choosing **Clear Workspace**, refreshing the browser page, or simply reopening the notebook.

If you're using Firefox and encounter this repeatedly, try clearing the cookies and site data for your SageMaker AI domain in Firefox's **Privacy & Security** settings, and then refresh the JupyterLab page.

Graph Explorer access issues

After notebook restarts or when using cached credentials, you may have trouble accessing Graph Explorer. This can show up as authentication errors, access denied messages, or Graph Explorer failing to load entirely.

To work around this, try clearing your browser cache and cookies. You can also open JupyterLab or Jupyter Classic first to force a credential refresh, and then open Graph Explorer. If the problem persists, try using an incognito or private browser window.

General tips

Many notebook issues are browser-related. To minimize problems, consider the following:

- Regularly clear your browser cache when working with Neptune notebooks.
- Always use the most recent notebook session rather than reopening stale ones.
- Use an incognito or private browser window if you encounter persistent issues.
- Keep your JupyterLab and graph-notebook extensions up to date.

Setting up Amazon Neptune

Amazon Neptune is a fully managed graph database service that makes it easy to build and run applications that work with highly connected datasets. To help you get started with Neptune, this section of the documentation covers a wide range of topics, from choosing the right instance types and storage configurations to securely connecting to your Neptune cluster, loading data, and monitoring performance. The key areas covered in this section are:

- Instance Types: The documentation outlines the various instance types available for Neptune, along with their specifications and considerations to help you select the most appropriate option for your workload.
- Storage Types: It also covers the different storage types supported by Neptune and guidance on how to choose the right storage configuration.
- Cluster Creation: The documentation provides step-by-step instructions for creating a new Neptune cluster, including how to configure the required VPC settings.
- Connectivity and Security: It covers how to securely connect to your Neptune cluster, set up VPC access, and configure the necessary security measures to protect your data.
- Data Loading: The documentation offers guidance on loading data into your Neptune cluster, including options for bulk loading and streaming data using various tools and APIs.
- Monitoring and Troubleshooting: It also includes information on monitoring your Neptune environment, accessing logs, and troubleshooting common issues that may arise during operation.

> **Note**
>
> For AWS graph database reference architectures and reference deployment architectures, See Amazon Neptune Resources. These resources can help inform your choices about graph data models and query languages, and accelerate your development process.

Choosing instance types for Amazon Neptune

Amazon Neptune offers a number of different instance sizes and families. that offer different capabilities suited to different graph workloads. This section is meant to help you choose the best instance type for your needs.

For the pricing of each instance-type in these families, please see the Neptune pricing page.

Overview of instance resource allocation

Each Amazon EC2 instance type and size used in Neptune offers a defined amount of compute (vCPUs) and system memory. The primary storage for Neptune is external to the DB instances in a cluster, which lets compute and storage capacity scale independently of each other.

This section focuses on how the compute resources can be scaled, and on the differences between each of the various instance families.

In all instance families, vCPU resources are allocated to support two (2) query execution threads per vCPU. This support is dictated by the instance size. When determining the proper size of a given Neptune DB instance, you need to consider the possible concurrency of your application and the average latency of your queries. You can estimate the number of vCPUs needed as follows, where latency is measured as the average query latency in seconds and concurrency is measured as the target number of queries per second:

```
vCPUs = (latency x concurrency) / 2
```

> **Note**
>
> SPARQL queries, openCypher queries, and Gremlin read queries that use the DFE query engine can, under certain circumstances, use more than one execution thread per query. When initially sizing your DB cluster, start with the assumption that each query will consume a single execution thread per execution and scale up if you observe back pressure into query queue. This can be observed by using the `/gremlin/status`, `/oc/status`, or `/sparql/status` APIs, or it can also be observed using the `MainRequestsPendingRequestsQueue` CloudWatch metric.

System memory on each instance is divided into two primary allocations: buffer pool cache and query execution thread memory.

Approximately two thirds of the available memory in an instance is allocated for buffer-pool cache. Buffer-pool cache is used to cache the most recently used components of the graph for faster access on queries that repeatedly access those components. Instances with a larger amount of system memory have larger buffer pool caches that can store more of the graph locally. A user can

tune for the appropriate amount of buffer-pool cache by monitoring the buffer cache hit and miss metrics available in CloudWatch.

You may want to increase the size of your instance if the cache hit rate drops below 99.9% for a consistent period of time. This suggests that the buffer pool is not big enough, and the engine is having to fetch data from the underlying storage volume more often than is efficient.

The remaining third of system memory is distributed evenly across query execution threads, with some memory remaining for the operating system and a small dynamic pool for threads to use as needed. The memory available for each thread increases slightly from one instance size to the next up to an `8xl` instance type, at which size the memory allocated per thread reaches a maximum.

The time to add more thread memory is when you encounter an `OutOfMemoryException` (OOM). OOM exceptions occur when one thread needs more than the maximum memory allocated to it (this is not the same as the entire instance running out of memory).

t3 and t4g instance types

The `t3` and `t4g` family of instances offers a low-cost option for getting started using a graph database and also for initial development and testing. These instances are eligible for the Neptune free-tier offer, which lets new customers use Neptune at no cost for the first 750 instance hours used within a standalone AWS account or rolled up underneath an AWS Organization with Consolidated Billing (Payer Account).

The `t3` and `t4g` instances are only offered in the medium size configuration (`t3.medium` and `t4g.medium`).

They are not intended for use in a production environment.

Because these instances have very constrained resources, they are not recommended for testing query execution time or overall database performance. To assess query performance, upgrade to one of the other instance families.

r4 family of instance types

DEPRECATED – The `r4` family was offered when Neptune was launched in 2018, but now newer instance types offer much better price/performance. As of engine version 1.1.0.0, Neptune no longer supports `r4` instance types.

r5 family of instance types

The r5 family contains memory-optimized instance types that work well for most graph use cases. The r5 family contains instance types from r5.large up to r5.24xlarge. They scale linearly in compute performance as you increase in size. For example, an r5.xlarge (4 vCPUs and 32GiB of memory) has twice the vCPUs and memory of an r5.large (2 vCPUs and 16GiB of memory), and an r5.2xlarge (8 vCPUs and 64GiB of memory) has twice the vCPUs and memory of an r5.xlarge. You can expect query performance to scale directly with compute capacity up to the r5.12xlarge instance type.

The r5 instance family has a 2-socket Intel CPU architecture. The r5.12xlarge and smaller types use a single socket and the system memory owned by that single-socket processor. The r5.16xlarge and r5.24xlarge types use both sockets and available memory. Because there's some memory-management overhead required between two physical processors in a 2-socket architecture, the performance gains scaling up from a r5.12xlarge to a r5.16xlarge or r5.24xlarge instance type are not as linear as you get scaling up at the smaller sizes.

r5d family of instance types

Neptune has a lookup-cache feature that can be used to improve the performance of queries which need to fetch and return large numbers of property values and literals. This feature is used primarily by customers with queries that need to return many attributes. The lookup cache boosts performance of these queries by fetching these attribute values locally rather than looking up each one over and over in Neptune indexed storage.

The lookup cache is implemented using a NVMe-attached EBS volume on an r5d instance type. It is enabled using a cluster's parameter group. As data is fetched from Neptune indexed storage, property values and RDF literals are cached within this NVMe volume.

If you don't need the lookup cache feature, use a standard r5 instance type rather than an r5d, to avoid the higher cost of the r5d.

The r5d family has instance types in the same sizes as the r5 family, from r5d.large to r5d.24xlarge.

r6g family of instance types

AWS has developed its own ARM-based processor called Graviton, that delivers better price/performance than the Intel and AMD equivalents. The r6g family uses the Graviton2 processor.

In our testing, the Graviton2 processor offers 10-20% better performance for OLTP-style (constrained) graph queries. Larger, OLAP-ish queries, however, may be slightly less performant with the Graviton2 processors than with Intel ones owing to slightly less performant memory-paging performance.

It's also important to note that the `r6g` family has a single-socket architecture, which means that performance scales linearly with compute capacity from an `r6g.large` to an `r6g.16xlarge` (the largest type in the family).

`r6i` family of instance types

Amazon R6i instances are powered by 3rd-generation Intel Xeon Scalable processors (code named Ice Lake) and are an ideal fit for memory-intensive workloads. As a general rule they offer up to 15% better compute price performance and up to 20% higher memory bandwidth per vCPU than comparable R5 instance types.

x2g family of instance types

Some graph use cases see better performance when instances have larger buffer-pool caches. The `x2g` family was launched to better support those use cases. The `x2g` family has a larger memory-to-vCPU ratio than the `r5` or `r6g` family. The `x2g` instances also use the Graviton2 processor, and have many of the same performance characteristics as `r6g` instance types, as well as a larger buffer-pool cache.

If you're using `r5` or `r6g` instance types with low CPU utilization and a high buffer-pool cache miss rate, try using the `x2g` family instead. That way, you'll be getting the additional memory you need without paying for more CPU capacity.

`x2iezn` family of instance types

The `x2iezn` family provides memory-optimized instances powered by Intel Xeon Scalable processors with high-frequency performance. These instances offer a high memory-to-vCPU ratio (32 GiB per vCPU), making them well-suited for memory-intensive graph workloads that benefit from high single-threaded performance.

Key features include up to 4.5 GHz all-core turbo frequency and availability in sizes from 2xlarge to 12xlarge.

x2iedn family of instance types

The x2iedn family provides memory-optimized instances with local NVMe SSD storage. These instances combine high memory capacity (32 GiB per vCPU) with fast local storage, making them ideal for graph workloads that benefit from both large in-memory caches and high-performance local disk caching.

Powered by 3rd generation Intel Xeon Scalable processors, these instances are available in sizes from xlarge to 32xlarge and are optimized for large-scale graph databases requiring both memory and storage performance.

r8g family of instance types

The r8g family contains memory-optimized instance types powered by AWS Graviton4 processors. These instances offer significant performance improvements over previous generations, making them well-suited for memory-intensive graph workloads. The r8g instances provide approximately 15-20% better performance for graph queries compared to r7g instances.

The r8g family uses a dual-socket platform. Instance types from r8g.large to r8g.24xlarge run on a single socket, which means that performance scales linearly with compute capacity across that range. The r8g.48xlarge uses both sockets and is the largest instance type in the family; as with other dual-socket families, performance gains when scaling from r8g.24xlarge to r8g.48xlarge may not be perfectly linear due to cross-socket memory management overhead.

Key features of the r8g family include:

- Powered by AWS Graviton4 ARM-based processors
- Higher memory bandwidth per vCPU compared to previous generations
- Excellent price/performance ratio for both OLTP-style (constrained) graph queries and OLAP-style analytical workloads
- Improved memory management capabilities that benefit complex graph traversals

The r8g family is ideal for production workloads that require high memory capacity and consistent performance. They're particularly effective for applications with high query concurrency requirements.

r7g family of instance types

The r7g family uses the AWS Graviton3 processor, which delivers better price/performance than previous Graviton2-based instances. In testing, the Graviton3 processor offers 25-30% better performance for OLTP-style graph queries compared to r6g instances.

Like the r6g family, the r7g family has a single-socket architecture, which means that performance scales linearly with compute capacity from an r7g.large to an r7g.16xlarge (the largest type in the family).

Key features of the r7g family include:

- Powered by AWS Graviton3 ARM-based processors
- Improved memory-paging performance compared to r6g, benefiting both OLTP and OLAP workloads
- Enhanced buffer-pool cache efficiency
- Lower latency for memory-intensive operations

The r7g family is well-suited for production environments with varied query patterns and is particularly effective for workloads that benefit from improved memory bandwidth.

r7i family of instance types

The r7i family is powered by 4th-generation Intel Xeon Scalable processors (code named Sapphire Rapids) and offers significant improvements over r6i instances. These instances provide approximately 15% better compute price/performance and up to 20% higher memory bandwidth per vCPU than comparable r6i instance types.

The r7i instance family has a 2-socket Intel CPU architecture, similar to the r5 family. The r7i.12xlarge and smaller types use a single socket and the system memory owned by that single-socket processor. The r7i.16xlarge and r7i.24xlarge types use both sockets and available memory. Because there's some memory-management overhead required between two physical processors in a 2-socket architecture, the performance gains scaling up from a r7i.12xlarge to a r7i.16xlarge or r7i.24xlarge instance type are not as linear as you get scaling up at the smaller sizes.

Key features of the r7i family include:

- Powered by 4th-generation Intel Xeon Scalable processors

- Performance scales linearly with compute capacity up to r7i.12xlarge
- Enhanced memory management between physical processors in the 2-socket architecture
- Improved performance for memory-intensive graph operations

For all of these instance families, you can estimate the number of vCPUs needed using the same formula mentioned previously:

```
vCPUs = (latency x concurrency) / 2
```

Where latency is measured as the average query latency in seconds and concurrency is measured as the target number of queries per second.

`serverless` instance type

The Neptune Serverless feature can scale instance size dynamically based on a workload's resource needs. Instead of calculating how many vCPUs are needed for your application, Neptune Serverless lets you set lower and upper limits on compute capacity (measured in Neptune Capacity Units) for the instances in your DB cluster. Workloads with varying utilization can be cost-optimized by using serverless rather than provsioned instances.

You can set up both provisioned and serverless instances in the same DB cluster to achieve an optimal cost-performance configuration.

Choosing storage types for Amazon Neptune

Neptune offers two types of storage with a different pricing model:

- **Standard storage** – Standard storage provides cost-effective database storage for applications with moderate to low I/O usage.
- **I/O–Optimized storage** – With I/O–Optimized storage, available from engine version 1.3.0.0, you pay only for the storage and instances you are using. The storage costs are higher than for standard storage, and also the instance costs are higher than for standard instances. You pay nothing for the I/O that you use. If your I/O usage is high, provisioned IOPs storage can reduce costs significantly.

  I/O–Optimized storage is designed to meet the needs of I/O–intensive graph workloads at a predictable cost. You can only switch between I/O–Optimized and Standard storage types once per 30 days.

For pricing information about I/O–Optimized storage, see the Neptune pricing page. The following section describes how to set up I/O–Optimized storage for a Neptune DB cluster.

Choosing I/O–Optimized storage for a Neptune DB cluster

By default, Neptune DB clusters use standard storage.You can enable I/O–Optimized storage on a DB cluster at the time you create it, like this:

Here is an example of how you can enable I/O–Optimized storage when you create a cluster using the AWS CLI:

```
aws neptune create-db-cluster \
  --db-cluster-identifier (an ID for the cluster) \
  --engine neptune \
  --engine-version (the Neptune engine version) \
  --storage-type iopt1
```

Then, any instance you create automatically has I/O–Optimized storage enabled:

```
aws neptune create-db-instance \
  --db-cluster-identifier (the ID of the new cluster) \
  --db-instance-identifier (an ID for the new instance) \
  --engine neptune \
  --db-instance-class db.r5.large
```

You can also modify an existing DB cluster to enable I/O–Optimized storage on it, like this:

```
aws neptune modify-db-cluster \
  --db-cluster-identifier (the ID of a cluster without I/O-Optimized storage) \
  --storage-type iopt1 \
  --apply-immediately
```

You can restore a backup snapshot to a DB cluster with I/O–Optimized storage enabled:

```
aws neptune restore-db-cluster-from-snapshot \
  --db-cluster-identifier (an ID for the restored cluster) \
  --snapshot-identifier (the ID of the snapshot to restore from) \
  --engine neptune \
  --engine-version (the Neptune engine version) \
```

```
  --storage-type iopt1
```

You can determine whether a cluster is using I/O–Optimized storage using any `describe-` call. If the I/O–Optimized storage is enabled, the call returns a storage-type field set to `iop1`.

Creating an Amazon Neptune cluster

The easiest way to create a new Amazon Neptune DB cluster is to use an CloudFormation template that creates all the required resources for you, without having to do everything by hand. The CloudFormation template performs much of the setup for you, including creating an Amazon Elastic Compute Cloud (Amazon EC2) instance:

To launch a new Neptune DB cluster using an CloudFormation template

1. Create a new IAM user with the permissions you will need for working with your Neptune DB cluster, as explained in IAM user permissions.
2. Set up additional prerequisites needed to use the CloudFormation template, as explained in Prerequisites for setting up Amazon Neptune using AWS CloudFormation.
3. Invoke the CloudFormation stack, as described in Creating an Amazon Neptune cluster using AWS CloudFormation.

You can also create a Neptune global database that spans multiple AWS Regions, enabling low-latency global reads and providing fast recovery in the rare case where an outage affects an entire AWS Region.

For information about creating an Amazon Neptune cluster manually using the AWS Management Console, see Launching a Neptune DB cluster using the AWS Management Console.

You can also use an CloudFormation template to create a Lambda function to use with Neptune (see Using CloudFormation to Create a Lambda Function to Use in Neptune).

For general information about managing clusters and instances in Neptune, see Managing Your Amazon Neptune Database.

Prerequisites for setting up Amazon Neptune using AWS CloudFormation

Before you create an Amazon Neptune cluster using an CloudFormation template, you need to have the following:

- An Amazon EC2 key pair.
- The permissions required for using CloudFormation.

Create an Amazon EC2 Key Pair to use for launching a Neptune cluster using CloudFormation

In order to launch a Neptune DB cluster using an CloudFormation template, you must have an Amazon EC2key pair (and its associated PEM file) available in the region where you create the CloudFormation stack.

If you need to create the key pair, see either Creating a Key Pair Using Amazon EC2 in the Amazon EC2 User Guide, or Creating a Key Pair Using Amazon EC2 in the Amazon EC2 User Guide for instructions.

Add IAM policies to grant permissions needed to use the CloudFormation template

First, you need to have an IAM user set up with permissions needed for working with Neptune, as described in Creating an IAM user with permissions for Neptune.

Then you need to add the AWS managed policy, `AWSCloudFormationReadOnlyAccess`, to that user.

Finally, you need to create the following customer-managed policy and add it to that user:

> **Important**
>
> You must create this as a *customer-managed policy*, not as an inline policy. The total size of all inline policies for an IAM user cannot exceed 2,048 non-whitespace characters, and this policy exceeds that limit. For more information, see IAM and STS character limits in the *IAM User Guide*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws-eusc:iam::111122223333:role/*",
            "Condition": {
                "StringEquals": {
                    "iam:passedToService": "rds.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws-eusc:iam::*:role/aws-service-role/
rds.amazonaws.com/AWSServiceRoleForRDS",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": "rds.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "sns:ListTopics",
                "sns:ListSubscriptions",
                "sns:Publish"
            ],
            "Resource": "arn:aws-eusc:sns:*:111122223333:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "kms:ListRetirableGrants",
                "kms:ListKeys",
                "kms:ListAliases",
                "kms:ListKeyPolicies"
            ],
```

```
            "Resource": "arn:aws-eusc:kms:*:111122223333:key/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "cloudwatch:GetMetricStatistics",
                "cloudwatch:ListMetrics"
            ],
            "Resource": "arn:aws-eusc:cloudwatch:*:111122223333:service/*-*",
            "Condition": {
                "StringLike": {
                    "cloudwatch:namespace": "AWS/Neptune"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeAvailabilityZones",
                "ec2:DescribeVpcs",
                "ec2:DescribeAccountAttributes",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcAttribute"
            ],
            "Resource": [
                "arn:aws-eusc:ec2:*:111122223333:vpc/*",
                "arn:aws-eusc:ec2:*:111122223333:subnet/*",
                "arn:aws-eusc:ec2:*:111122223333:security-group/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "rds:CreateDBCluster",
                "rds:CreateDBInstance",
                "rds:AddTagsToResource",
                "rds:ListTagsForResource",
                "rds:RemoveTagsFromResource",
                "rds:RemoveRoleFromDBCluster",
                "rds:ResetDBParameterGroup",
                "rds:CreateDBSubnetGroup",
                "rds:ModifyDBParameterGroup",
                "rds:DownloadDBLogFilePortion",
```

```
                "rds:CopyDBParameterGroup",
                "rds:AddRoleToDBCluster",
                "rds:ModifyDBInstance",
                "rds:ModifyDBClusterParameterGroup",
                "rds:ModifyDBClusterSnapshotAttribute",
                "rds:DeleteDBInstance",
                "rds:CopyDBClusterParameterGroup",
                "rds:CreateDBParameterGroup",
                "rds:DescribeDBSecurityGroups",
                "rds:DeleteDBSubnetGroup",
                "rds:DescribeValidDBInstanceModifications",
                "rds:ModifyDBCluster",
                "rds:CreateDBClusterSnapshot",
                "rds:DeleteDBParameterGroup",
                "rds:CreateDBClusterParameterGroup",
                "rds:RemoveTagsFromResource",
                "rds:PromoteReadReplicaDBCluster",
                "rds:RestoreDBClusterFromSnapshot",
                "rds:DescribeDBSubnetGroups",
                "rds:DescribePendingMaintenanceActions",
                "rds:DescribeDBParameterGroups",
                "rds:FailoverDBCluster",
                "rds:DescribeDBInstances",
                "rds:DescribeDBParameters",
                "rds:DeleteDBCluster",
                "rds:ResetDBClusterParameterGroup",
                "rds:RestoreDBClusterToPointInTime",
                "rds:DescribeDBClusterSnapshotAttributes",
                "rds:AddTagsToResource",
                "rds:DescribeDBClusterParameters",
                "rds:CopyDBClusterSnapshot",
                "rds:DescribeDBLogFiles",
                "rds:DeleteDBClusterSnapshot",
                "rds:ListTagsForResource",
                "rds:RebootDBInstance",
                "rds:DescribeDBClusterSnapshots",
                "rds:DeleteDBClusterParameterGroup",
                "rds:ApplyPendingMaintenanceAction",
                "rds:DescribeDBClusters",
                "rds:DescribeDBClusterParameterGroups",
                "rds:ModifyDBSubnetGroup"
            ],
            "Resource": [
                "arn:aws-eusc:rds:*:111122223333:cluster-snapshot:*",
```

```
                "arn:aws-eusc:rds:*:111122223333:cluster:*",
                "arn:aws-eusc:rds:*:111122223333:pg:*",
                "arn:aws-eusc:rds:*:111122223333:cluster-pg:*",
                "arn:aws-eusc:rds:*:111122223333:secgrp:*",
                "arn:aws-eusc:rds:*:111122223333:db:*",
                "arn:aws-eusc:rds:*:111122223333:subgrp:*"
            ],
            "Condition": {
                "StringEquals": {
                    "rds:DatabaseEngine": [
                        "graphdb",
                        "neptune"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:GetLogEvents",
                "logs:DescribeLogStreams"
            ],
            "Resource": [
                "arn:aws-eusc:logs:*:111122223333:log-group:*:log-stream:*",
                "arn:aws-eusc:logs:*:111122223333:log-group:*"
            ]
        }
    ]
}
```

Note

The following permissions are only required to delete a stack: `iam:DeleteRole`, `iam:RemoveRoleFromInstanceProfile`, `iam:DeleteRolePolicy`, `iam:DeleteInstanceProfile`, and `ec2:DeleteVpcEndpoints`.
Also note that `ec2:*Vpc` grants `ec2:DeleteVpc` permissions.

Creating an Amazon Neptune cluster using AWS CloudFormation

You can use an CloudFormation template to set up a Neptune DB Cluster.

1. To launch the CloudFormation stack on the CloudFormation console, choose one of the **Launch Stack** buttons in the following table.

Region	View	View in Designer	Launch
US East (N. Virginia)	View	View in Designer	Launch Stack
US East (Ohio)	View	View in Designer	Launch Stack
US West (N. California)	View	View in Designer	Launch Stack
US West (Oregon)	View	View in Designer	Launch Stack
Canada (Central)	View	View in Designer	Launch Stack
Canada West (Calgary)	View	View in Designer	Launch Stack
South America (São Paulo)	View	View in Designer	Launch Stack
Europe (Stockholm)	View	View in Designer	Launch Stack
Europe (Ireland)	View	View in Designer	Launch Stack
Europe (London)	View	View in Designer	Launch Stack
Europe (Paris)	View	View in Designer	Launch Stack

Region	View	View in Designer	Launch
Europe (Spain)	View	View in Designer	Launch Stack
Europe (Frankfurt)	View	View in Designer	Launch Stack
Europe (Zurich)	View	View in Designer	Launch Stack
Middle East (Bahrain)	View	View in Designer	Launch Stack
Middle East (UAE)	View	View in Designer	Launch Stack
Israel (Tel Aviv)	View	View in Designer	Launch Stack
Africa (Cape Town)	View	View in Designer	Launch Stack
Asia Pacific (Hong Kong)	View	View in Designer	Launch Stack
Asia Pacific (Tokyo)	View	View in Designer	Launch Stack
Asia Pacific (Seoul)	View	View in Designer	Launch Stack
Asia Pacific (Singapore)	View	View in Designer	Launch Stack
Asia Pacific (Sydney)	View	View in Designer	Launch Stack
Asia Pacific (Jakarta)	View	View in Designer	Launch Stack
Asia Pacific (Melbourne)	View	View in Designer	Launch Stack

Region	View	View in Designer	Launch
Asia Pacific (Malaysia)	View	View in Designer	Launch Stack
Asia Pacific (Mumbai)	View	View in Designer	Launch Stack
Asia Pacific (Hyderabad)	View	View in Designer	Launch Stack
China (Beijing)	View	View in Designer	Launch Stack
China (Ningxia)	View	View in Designer	Launch Stack
AWS GovCloud (US-West)	View	View in Designer	Launch Stack
AWS GovCloud (US-East)	View	View in Designer	Launch Stack

2. On the **Select Template** page, choose **Next**.
3. On the **Specify Details** page, choose a key pair for the **EC2SSHKeyPairName**.

   This key pair is required to access the EC2 instance. Ensure that you have the PEM file for the key pair that you choose.
4. Choose **Next**.
5. On the **Options** page, choose **Next**.
6. On the **Review** page, select the first check box to acknowledge that CloudFormation will create IAM resources. Select the second check box to acknowledge `CAPABILITY_AUTO_EXPAND` for the new stack.

> **Note**
>
> `CAPABILITY_AUTO_EXPAND` explicitly acknowledges that macros will be expanded when creating the stack, without prior review. Users often create a change set from a processed template so that the changes made by macros can be reviewed before

actually creating the stack. For more information, see the CloudFormation CreateStack API.

Then choose **Create**.

> **Note**
>
> You can also use your CloudFormation template to upgrade your DB cluster's engine version.

Set up the Amazon VPC where your Amazon Neptune DB cluster is located

An Amazon Neptune DB cluster can *only* be created in an Amazon Virtual Private Cloud (Amazon VPC). Its endpoints are accessible within that VPC, and if Neptune public endpoints are enabled, they can also be accessed outside of the VPC and over the internet.

There are a number of different ways to set up the VPC, depending on how you want to access your DB cluster.

Here are some things to keep in mind when configuring the VPC where your Neptune DB cluster is located:

- Your VPC must have at least two subnets. These subnets must be in two different Availability Zones (AZs). By distributing your cluster instances across at least two AZs, Neptune helps ensure that there are always instances available in your DB cluster even in the unlikely event of an availability zone failure. The cluster volume for your Neptune DB cluster always spans three AZs to provide durable storage with extremely low likelihood of data loss.
- The CIDR blocks in each subnet must be large enough to provide IP addresses that Neptune may need during maintenance activities, failover, and scaling.
- The VPC must have a DB subnet group that contains subnets that you have created. Neptune chooses one of the subnets in the subnet group and an IP address within that subnet to associate with each DB instance in the DB cluster. The DB instance is then located in the same AZ as the subnet.

- The VPC should have DNS enabled (both DNS hostnames and DNS resolution).
- The VPC must have a VPC security group to allows access to your DB cluster.
- Tenancy in a Neptune VPC should be set to **Default**.

Adding subnets to the VPC where your Neptune DB cluster is located

A subnet is a range of IP addresses in your VPC. You can launch resources such as a Neptune DB cluster or an EC2 instance into a specific subnet. When you create a subnet, you specify the IPv4 CIDR block for the subnet, which is a subset of the VPC CIDR block. Each subnet must reside entirely within one Availability Zone (AZ) and cannot span zones. By launching instances in separate Availability Zones, you can protect your applications from a failure in one of the zones. See VPC subnet documentation for more information.

A Neptune DB cluster requires at least two VPC subnets.

To add subnets to a VPC

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://eusc-de-east-1.console.amazonaws-eusc.eu/vpc/.
2. In the navigation pane, choose **Subnets**.
3. In the **VPC Dashboard** choose **Subnets**, and then choose **Create subnet**.
4. On the **Create subnet** page, choose the VPC where you want to create the subnet.
5. Under **Subnet settings**, make the following choices:
   a. Enter a name for the new subnet under **Subnet name**.
   b. Choose an Availability Zone (AZ) for the subnet, or leave the choice at **No preference**.
   c. Enter the subnet's IP address block under **IPv4 CIDR block**.
   d. Add tags to the subnet if you need to.
   e. Choose
6. If you want to create another subnet at the same time, choose **Add new subnet**.
7. Choose **Create subnet** to create the new subnet(s).

Configuring VPC in Amazon Neptune

Create a subnet group.

To create a Neptune subnet group

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. Choose **Subnet groups**, and then choose **Create DB Subnet Group**.
3. Enter a name and description for the new subnet group (the description is required).
4. Under **VPC**, choose the VPC where you want this subnet group to be located.
5. Under **Avalability zone**, choose the AZ where you want this subnet group to be located.
6. Under **Subnet**, add one or more of the subnets in this AZ to this subnet group.
7. Choose **Create** to create the new subnet group.

Create a security group using the VPC console

Security groups provide access to your Neptune DB cluster in the VPC. They act as a firewall for the associated DB cluster, controlling both inbound and outbound traffic at the instance level. By default, a DB instance is created with a firewall and a default security group that prevents any access to it. To enable access, you must have a VPC security group with additional rules.

The following procedure shows you how to add a custom TCP rule that specifies the port range and IP addresses for the Amazon EC2 instance to use to access your Neptune DB cluster. You can use the VPC security group assigned to the EC2 instance rather than its IP address.

To create a VPC security group for Neptune on the console

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://eusc-de-east-1.console.amazonaws-eusc.eu/vpc/.
2. From the **Region** selector, choose the AWS Region where you want to create a VPC security group for Neptune. In the list of Amazon VPC resources for that region, it should show that you have at least one VPC and several subnets. If it does not, you don't have a default VPC in that Region.
3. In the navigation pane under **Security**, choose **Security Groups**.
4. Choose **Create security group**. In the **Create security group** window, enter the **Security group name**, a **Description**, and the identifier of the VPC where your Neptune DB cluster will reside.
5. Add an inbound rule for the security group of an Amazon EC2 instance that you want connected to your Neptune DB cluster:

   a. In the **Inbound rules** area, choose **Add rule**.
   b. In the **Type** list, leave **Custom TCP** selected.
   c. In the **Port range** box, enter **8182**, the default port value for Neptune.
   d. Under **Source**, enter the IP address range (CIDR value) from which you will access Neptune, or choose an existing security group name.
   e. If you need to add more IP addresses or different port ranges, choose **Add rule** again.
6. In the Outbound rules area, you can also add one or more outbound rules if you need to.
7. When you finish, choose **Create security group**.

You can use this new VPC security group when you create a new Neptune DB cluster.

If you use a default VPC, a default subnet group spanning all of the VPC's subnets is already created for you. When you choose the **Create database** in the Neptune console, the default VPC is used unless you specify a different one.

Make sure that you have DNS support in your VPC

Domain Name System (DNS) is a standard by which names used on the internet are resolved to their corresponding IP addresses. A DNS hostname uniquely names a computer and consists of a host name and a domain name. DNS servers resolve DNS hostnames to their corresponding IP addresses.

Check to make sure that DNS hostnames and DNS resolution are both enabled in your VPC. The VPC network attributes `enableDnsHostnames` and `enableDnsSupport` must be set to `true`. To view and modify these attributes, go to the VPC console at https://eusc-de-east-1.console.amazonaws-eusc.eu/vpc/.

For more information, see Using DNS with your VPC.

> **Note**
>
> If you are using Route 53, confirm that your configuration does not override DNS network attributes in your VPC.

Working with a Neptune DB cluster in dual-stack mode

When you create or modify a Neptune DB cluster, you can specify *dual-stack mode* to allow your resources to communicate with the DB cluster over IPv4, IPv6, or both. A DB cluster in dual-stack mode supports both Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6) addressing.

Overview of dual-stack mode

A Neptune DB cluster runs in dual-stack mode when it can communicate over both the IPv4 and IPv6 addressing protocols. Resources can then communicate with the DB cluster using either IPv4, IPv6, or both protocols. Private dual-stack mode DB clusters have IPv6 endpoints that are restricted to VPC access only, ensuring your IPv6 endpoints remain private. Public dual-stack mode DB clusters provide both IPv4 and IPv6 endpoints that you can access from the internet.

Dual-stack mode and DB subnet groups

To use dual-stack mode, each subnet in the DB subnet group must have an IPv6 CIDR block associated with it.

Because Neptune does not allow changing the DB subnet group after cluster creation, you must either create the cluster with a dual-stack-capable subnet group, or add IPv6 CIDR blocks to the existing subnets before modifying the cluster to dual-stack mode.

After a DB cluster is in dual-stack mode, clients can connect to it normally. Ensure that client security firewalls and Neptune DB cluster security groups allow traffic over IPv6. To connect, clients use the DB cluster's endpoint. The DB cluster detects the client's preferred network protocol and uses that protocol for the connection.

If a DB subnet group stops supporting dual-stack mode, there's a risk of an incompatible network state for associated DB clusters. This can happen because of subnet deletion or CIDR disassociation. Also, you can't use that DB subnet group to create a new dual-stack mode DB cluster.

Creating and modifying dual-stack mode DB clusters

When you create or modify a DB cluster, you can specify dual-stack mode to allow your resources to communicate with your DB cluster over IPv4, IPv6, or both.

When you use the AWS Management Console to create or modify a DB cluster, you can specify dual-stack mode in the **Network type** section.

When you use the AWS Command Line Interface (AWS CLI) to create a DB cluster, set the `--network-type` option to `DUAL` to use dual-stack mode:

```
aws neptune create-db-cluster \
    --db-cluster-identifier my-dual-stack-cluster \
    --engine neptune \
    --db-subnet-group-name my-dual-stack-subnet-group \
    --network-type DUAL
```

When you use the AWS CLI to modify an existing DB cluster to use dual-stack mode:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier my-cluster \
    --network-type DUAL \
    --apply-immediately
```

When you use the Neptune API to create or modify a DB cluster, set the `NetworkType` parameter to `DUAL` to use dual-stack mode.

Modifying IPv4-only DB clusters to use dual-stack mode

You can modify an IPv4-only DB cluster to use dual-stack mode. To do so, change the network type of the DB cluster.

Before modifying a DB cluster to use dual-stack mode, ensure that its DB subnet group supports dual-stack mode. Because you cannot change the DB subnet group after the cluster is created, you must add IPv6 CIDR blocks to the existing subnets in the cluster's subnet group.

To modify an IPv4-only DB cluster to use dual-stack mode

1. Add IPv6 support to the existing DB subnet group's subnets:
    a. Associate an IPv6 CIDR block with your VPC. For instructions, see Add an IPv6 CIDR block to your VPC in the *Amazon VPC User Guide*.
    b. Attach the IPv6 CIDR block to all of the subnets in your DB subnet group. For instructions, see Add an IPv6 CIDR block to your subnet in the *Amazon VPC User Guide*.
    c. Confirm that the DB subnet group supports dual-stack mode. Using the AWS CLI, run the `describe-db-subnet-groups` command and check `SupportedNetworkTypes` in the output.

2. Modify the DB cluster to use dual-stack mode. Using the AWS CLI:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier my-cluster \
    --network-type DUAL \
    --apply-immediately
```

3. Confirm the DB cluster is in dual-stack mode by checking the `NetworkType` in the output of `describe-db-clusters`.

If you can't connect to the DB cluster after the change, ensure that the client and database security firewalls and route tables allow traffic to the database on the selected network (either IPv4 or IPv6).

Limitations for dual-stack network DB clusters

The following limitations apply to dual-stack network Neptune DB clusters:

- The network type is set at the DB cluster level. All DB instances in the cluster inherit the cluster's network type. You cannot set a different network type for individual instances.
- DB clusters can't use the network type `IPV6` exclusively. They can use `IPV4` exclusively or dual-stack mode (`DUAL`).
- Neptune doesn't support native IPv6 subnets (IPv6-only subnets). DB cluster subnets must support both IPv4 and IPv6 to use dual-stack mode.

Connecting to an Amazon Neptune cluster

After creating a Neptune cluster, you must set up network access so that your applications can reach the cluster endpoint. The following sections describe the network connectivity options. For more information about querying your data after you connect, see Accessing graph data.

Network connectivity options

An Amazon Neptune DB cluster can *only* be created in an Amazon Virtual Private Cloud (Amazon VPC). Its endpoints are accessible only within that VPC unless you enable and set up Neptune public endpoints for the DB cluster.

You can set up network access to your Neptune DB cluster in its VPC in several ways:

- Connecting from an Amazon EC2 instance in the same VPC
- Connecting from an Amazon EC2 instance in another VPC
- Connecting from a private network
- Connecting from a public endpoint

Verify your connection

After you set up network access, you can verify that your connection works by calling the instance status endpoint. A successful response confirms that your client can reach the Neptune cluster.

AWS CLI

Run the following command:

```
aws neptunedata get-engine-status \
  --endpoint-url https://your-neptune-endpoint:port \
  --region us-east-1
```

For more information, see get-engine-status in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=10, retries={'total_max_attempts': 1})
)

response = client.get_engine_status()

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/status \
  --region us-east-1 \
```

```
  --service neptune-db
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

Run the following command:

```
curl -G https://your-neptune-endpoint:port/status
```

> **Note**
>
> This command works only when IAM authentication is not enabled on your cluster. If IAM authentication is enabled, use **awscurl** or the AWS CLI instead.

A healthy cluster returns a JSON response that includes `"status": "healthy"`. For example:

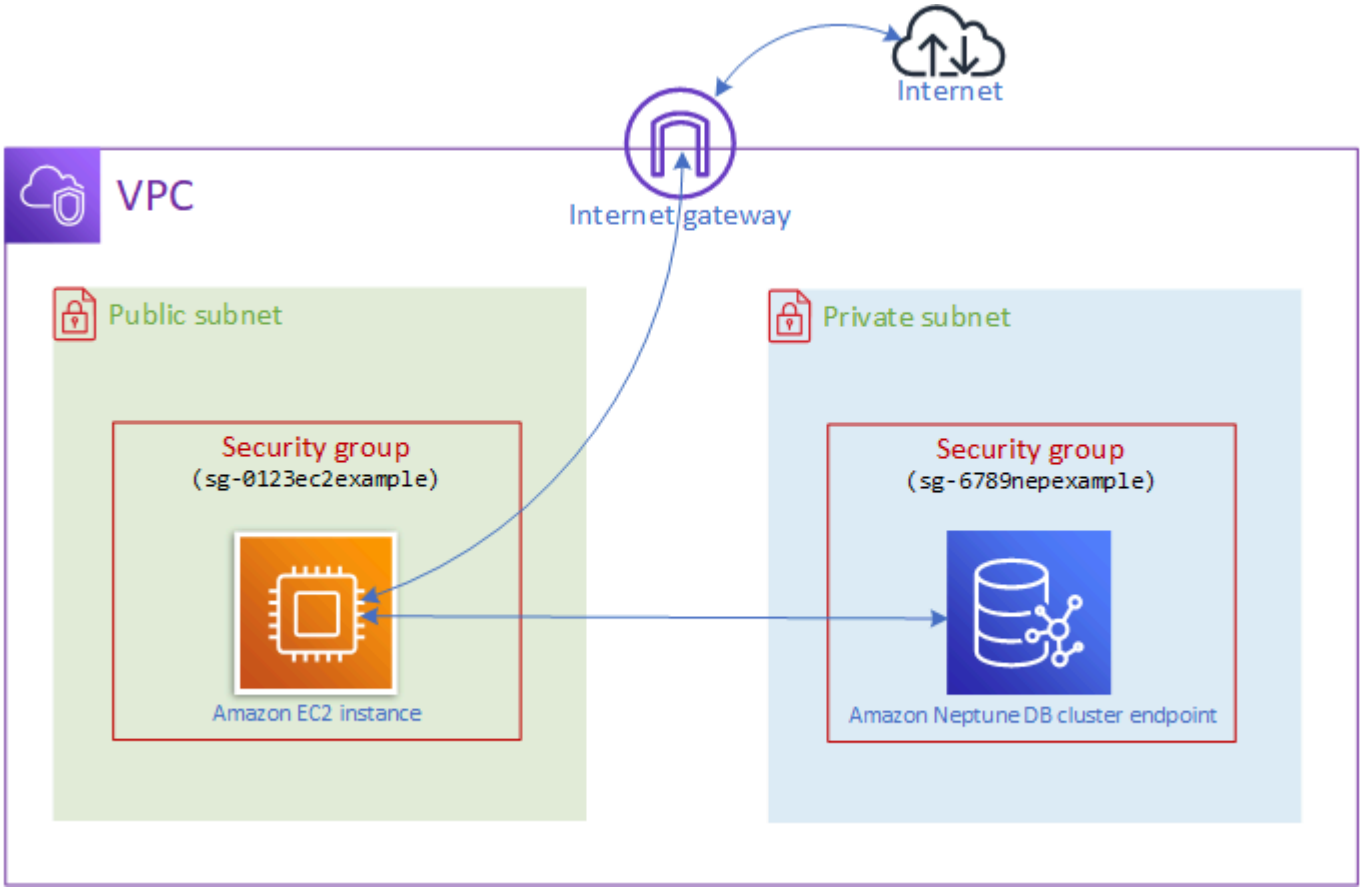
```
{
    "status": "healthy",
    "startTime": "Thu Aug 24 21:07:13 UTC 2023",
    "dbEngineVersion": "1.3.1.0.R1",
    "role": "writer",
    "dfeQueryEngine": "viaQueryHint",
    "gremlin": { "version": "tinkerpop-3.7.2" },
    "sparql": { "version": "sparql-1.1" },
    "opencypher": { "version": "Neptune-9.0.20190305-1.0" },
    "labMode": { ... },
    "features": { ... },
    "settings": { ... }
}
```

If you cannot reach the endpoint, verify that your network configuration allows traffic on port 8182 (or your configured port) and that your security group rules permit inbound connections from

your client. For more information, see Securing access to Neptune. For help setting up the AWS CLI, SDKs, or `curl`, see Using command-line tools.

Connecting an Amazon EC2 instance to Amazon Neptune cluster in the same VPC

One of the most common ways to connect to a Neptune database is from an Amazon EC2 instance in the same VPC as your Neptune DB cluster. For example, the EC2 instance might be running a web server that connects with the internet. In this case, only the EC2 instance has access to the Neptune DB cluster, and the internet only has access to the EC2 instance:



To enable this configuration, you need to have the right VPC security groups and subnet groups set up. The web server is hosted in a public subnet, so that it can reach the public internet, and your Neptune cluster instance is hosted in a private subnet to keep it secure. See Set up the Amazon VPC where your Amazon Neptune DB cluster is located.

In order for the Amazon EC2 instance to connect to your Neptune endpoint on, for example, port 8182, you will need to set up a security group to do that. If your Amazon EC2 instance is using a security group named, for example, `ec2-sg1`, you need to create another Amazon EC2 security

group (let's say `db-sg1`) that has inbound rules for port 8182 and has `ec2-sg1` as its source. Then, add `db-sg1` to your Neptune cluster to allow the connection.

After creating the Amazon EC2 instance, you can log into it using SSH and connect to your Neptune DB cluster. For information about connecting to an EC2 instance using SSH, see Connect to your Linux instance in the *Amazon EC2 User Guide*.

If you are using a Linux or macOS command line to connect to the EC2 instance, you can paste the SSH command from the **SSHAccess** item in the **Outputs** section of the CloudFormation stack. You must have the PEM file in the current directory and the PEM file permissions must be set to 400 (`chmod 400 keypair.pem`).

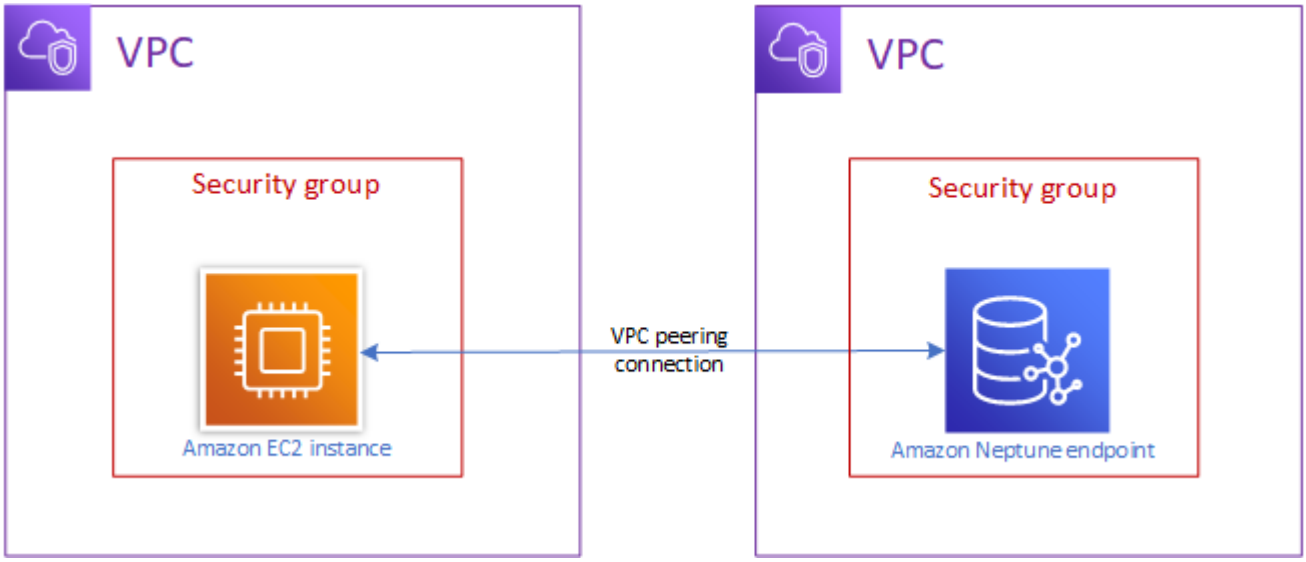
To create a VPC with both private and public subnets

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://eusc-de-east-1.console.amazonaws-eusc.eu/vpc/.
2. From the **Region** selector, choose the AWS Region to create your VPC in.
3. In the **VPC Dashboard**, choose **Launch VPC Wizard**.
4. Complete the **VPC Settings** area of the **Create VPC** page:
   a. Under **Resources to create**, choose **VPC, subnets, etc.**.
   b. Leave the default name tag as is, or enter a name of your choosing, or un-check the **Auto-generate** check box to disable name tag generation.
   c. Leave the IPv4 CIDR block value at `10.0.0.0/16`.
   d. Leave the IPv6 CIDR block value at **No IPv6 CIDR block**.
   e. Leave the **Tenancy** at **Default**.
   f. Leave the number of **Availability Zones (AZs)** at **2**.
   g. Leave **NAT gateways ($)** at **None**, unless you will be needing one or more NAT gateways.
   h. Set **VPC endpoints** to **None**, unless you will be using Amazon S3.
   i. Both **Enable DNS hostnames** and **Enable DNS resolution** should be checked.
5. Choose **Create VPC**.

Connecting an Amazon EC2 instance to an Amazon Neptune cluster in a different VPC

An Amazon Neptune DB cluster can *only* be created in an Amazon Virtual Private Cloud (Amazon VPC), and its endpoints are accessible within that VPC, usually from an Amazon Elastic Compute Cloud (Amazon EC2) instance running in that VPC. Alternatively, it can be accessed using a public endpoint. For more information on public endpoints, see Neptune public endpoints.

When your DB cluster is in a different VPC from the EC2 instance you are using to access it, you can use VPC peering to make the connection:



A VPC peering connection is a networking connection between two VPCs that routes traffic between them privately, so that instances in either VPC can communicate as if they are within the same network. You can create a VPC peering connection between VPCs in your account, between a VPC in your AWS account and a VPC in another AWS account, or with a VPC in a different AWS Region.

AWS uses the existing infrastructure of a VPC to create a VPC peering connection. It is neither a gateway nor an AWS Site-to-Site VPN connection, and it does not rely on a separate piece of physical hardware. It has no single point of failure for communication and no bandwidth bottleneck.

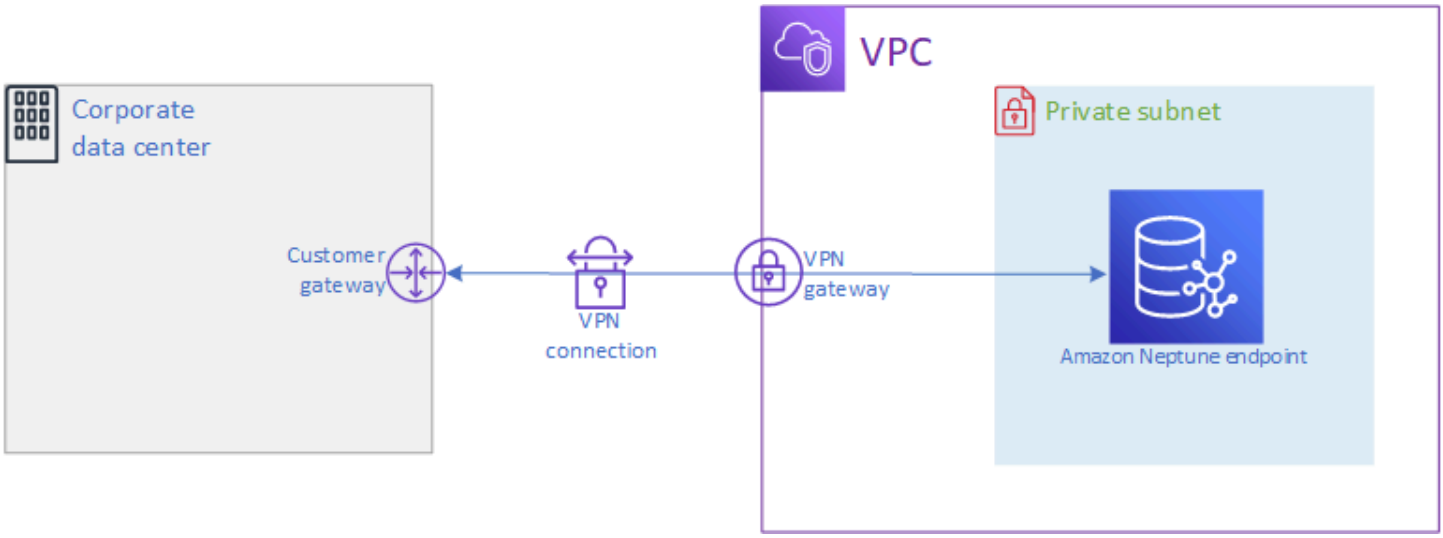
See the Amazon VPC Peering Guide for more information about how use VPC peering.

Connecting to an Amazon Neptune cluster over a private network

You can access a Neptune DB cluster from a private network in two different ways:

- Using an AWS Site-to-Site VPN connection.
- Using an AWS Direct Connect connection.

The links above have information about these connection methods and how to set them up. The configuration of an AWS Site-to-Site connection might look like this:



Neptune Public Endpoints

Overview

Amazon Neptune clusters are typically deployed within your VPC and can only be accessed from within that VPC. This requires configuring applications and development environments within the VPC or using proxy services to connect to the VPC, which increases setup time and costs.

Public endpoints simplify this experience by allowing direct connections to Neptune over the internet, making it easier to get started with graph databases without specialized networking knowledge.

When to use public endpoints

Consider using public endpoints in the following scenarios:

- You want to quickly test Neptune in a development or test environment without complex network configuration
- You don't have specialized AWS networking knowledge
- Your application's security posture doesn't require a private VPC

- You need to connect to Neptune from your local development environment

Security considerations

When using public endpoints, keep these security considerations in mind:

- IAM authentication is required for clusters with public endpoints enabled.
- Access to the database is controlled by the security group it uses.
- You can restrict which IP addresses can connect to your cluster.
- You can use IAM policies to control who can create or modify clusters with public access. See: Restricting public access creation

Enabling public endpoints

By default, new Neptune databases are created with public endpoints disabled. You must explicitly enable public access when creating or modifying a cluster.

Public endpoints are supported from Neptune engine release version 1.4.6.x. You need to upgrade existing clusters to at least this version to use this feature.

Public endpoint setting is available on the Neptune instance and not the Neptune cluster. Hence, a Neptune cluster can exist with some instances with public endpoints and some with not. However, we don't recommend having such a setting. For more information on this refer to: How public endpoints work

Prerequisites

IAM authentication setting on the Neptune cluster

Before enabling public endpoints on a Neptune instance, ensure your cluster supports IAM authentication. If not, enable it using the following command:

```
aws neptune modify-db-cluster \
  --region us-west-2 \
  --engine graphdb \
  --engine-version 1.4.6.x \
  --db-cluster-identifier neptune-public-endpoint \
  --enable-iam-database-authentication
```

Network settings

1. Ensure your VPC has subnets that enable public routing (has an entry for an internet gateway in the route table of the subnets). If you don't provide a `db-subnet-group-name` parameter while creating the cluster, the default subnet group is picked for the cluster creation.
2. Make sure the security group attached to the cluster allows inbound traffic for the allowed IP ranges and the allowed ports. For example, if you want to allow TCP traffic from all IPs to connect to the Neptune instance running on port 8182, the inbound rule should have:
   a. Type: All TCP
   b. Protocol: TCP
   c. Port range: 8182
   d. CIDR block: 0.0.0.0/0

> **Note**
>
> Although you can set the CIDR block range to 0.0.0.0/0, we recommend reducing this to a specific IP range of your client application for a better security posture.

Creating a new instance with public endpoints

You can create a new Neptune instance with public endpoints using the AWS Management Console, AWS CLI, or AWS SDK.

Using the AWS CLI:

```
aws neptune create-db-instance \
  --region us-west-2 \
  --engine graphdb \
  --engine-version 1.4.6.x \
  --db-cluster-identifier neptune-public-endpoint \
  --publicly-accessible
```

Modifying an existing instance for public access

To modify an existing Neptune instance to enable public access:

```
aws neptune modify-db-instance \
```

```
  --region us-west-2 \
  --engine graphdb \
  --engine-version 1.4.6.x \
  --db-instance-identifier neptune-public-endpoint \
  --publicly-accessible
```

> **Note**
>
> Public access is enabled at the instance level, not the cluster level. To ensure your cluster is always accessible via public endpoints, all instances in the cluster must have public access enabled.

Using the public endpoints

To check if your database is reachable, check the status using the AWS CLI `NeptuneData` API:

```
aws neptunedata get-engine-status \
  --endpoint-url https://my-cluster-name.cluster-abcdefgh1234.us-
east-1.neptune.amazonaws.com:8182
```

If the database is accessible, the response is like:

```
{
    "status": "healthy",
    "startTime": "Sun Aug 10 06:54:15 UTC 2025",
    "dbEngineVersion": "1.4.6.0.R1",
    "role": "writer",
    "dfeQueryEngine": "viaQueryHint",
    "gremlin": {
        "version": "tinkerpop-3.7.1"
    },
    "sparql": {
        "version": "sparql-1.1"
    },
    "opencypher": {
        "version": "Neptune-9.0.20190305-1.0"
    },
    "labMode": {
        "ObjectIndex": "disabled",
        "ReadWriteConflictDetection": "enabled"
```

```
    },
    "features": {
        "SlowQueryLogs": "disabled",
        "InlineServerGeneratedEdgeId": "disabled",
        "ResultCache": {
            "status": "disabled"
        },
        "IAMAuthentication": "disabled",
        "Streams": "disabled",
        "AuditLog": "disabled"
    },
    "settings": {
        "StrictTimeoutValidation": "true",
        "clusterQueryTimeoutInMs": "120000",
        "SlowQueryLogsThreshold": "5000"
    }
}
```

Examples of how to query the database

AWS CLI

```
aws neptunedata execute-open-cypher-query \
--open-cypher-query "MATCH (n) RETURN n LIMIT 10" \
--endpoint-url https://my-cluster-name.cluster-abcdefgh1234.us-
east-1.neptune.amazonaws.com:8182
```

Python

```
import boto3
import json
from botocore.config import Config

# Configuration - Replace with your actual Neptune cluster details
cluster_endpoint = "my-cluster-name.cluster-abcdefgh1234.my-
region.neptune.amazonaws.com"
port = 8182
region = "my-region"

# Configure Neptune client
# This disables retries and sets the client timeout to infinite
#     (relying on Neptune's query timeout)
endpoint_url = f"https://{cluster_endpoint}:{port}"
```

```
config = Config(
    region_name=region,
    retries={'max_attempts': 1},
    read_timeout=None
)

client = boto3.client("neptunedata", config=config, endpoint_url=endpoint_url)

cypher_query = "MATCH (n) RETURN n LIMIT 5"
try:
    response = client.execute_open_cypher_query(openCypherQuery=cypher_query)
    print("openCypher Results:")
    for item in response.get('results', []):
        print(f"  {item}")
except Exception as e:
    print(f"openCypher query failed: {e}")
```

JavaScript

```
import {
    NeptunedataClient,
    GetPropertygraphSummaryCommand
} from "@aws-sdk/client-neptunedata";
import { inspect } from "util";
import { NodeHttpHandler } from "@smithy/node-http-handler";

/**
 * Main execution function
 */
async function main() {
    // Configuration - Replace with your actual Neptune cluster details
    const clusterEndpoint = 'my-cluster-name.cluster-abcdefgh1234.my-
region.neptune.amazonaws.com';
    const port = 8182;
    const region = 'my-region';

    // Configure Neptune client
    // This disables retries and sets the client timeout to infinite
    //     (relying on Neptune's query timeout)
    const endpoint = `https://${clusterEndpoint}:${port}`;
    const clientConfig = {
        endpoint: endpoint,
        sslEnabled: true,
```

```
        region: region,
        maxAttempts: 1,  // do not retry
        requestHandler: new NodeHttpHandler({
            requestTimeout: 0  // no client timeout
        })
    };

    const client = new NeptunedataClient(clientConfig);
    try {
        try {
            const command = new GetPropertygraphSummaryCommand({ mode: "basic" });
            const response = await client.send(command);
            console.log("Graph Summary:", inspect(response.payload, { depth: null }));
        } catch (error) {
            console.log("Property graph summary failed:", error.message);
        }
    } catch (error) {
        console.error("Error in main execution:", error);
    }
}

// Run the main function
main().catch(console.error);
```

Go

```
package main
import (
    "context"
    "fmt"
    "github.com/aws/aws-sdk-go-v2/aws"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/neptunedata"
    "os"
    "encoding/json"
    "net/http"
)

func main() {
    // Configuration - Replace with your actual Neptune cluster details
    clusterEndpoint := "my-cluster-name.cluster-abcdefgh1234.my-
region.neptune.amazonaws.com"
    port := 8182
```

```
    region := "my-region"

    // Configure Neptune client
    // Configure HTTP client with no timeout
    //    (relying on Neptune's query timeout)
    endpoint := fmt.Sprintf("https://%s:%d", clusterEndpoint, port)
    // Load AWS SDK configuration
    sdkConfig, _ := config.LoadDefaultConfig(
        context.TODO(),
        config.WithRegion(region),
        config.WithHTTPClient(&http.Client{Timeout: 0}),
    )

    // Create Neptune client with custom endpoint
    client := neptunedata.NewFromConfig(sdkConfig, func(o *neptunedata.Options) {
        o.BaseEndpoint = aws.String(endpoint)
        o.Retryer = aws.NopRetryer{} // Do not retry calls if they fail
    })

    gremlinQuery :=
 "g.addV('person').property('name','charlie').property(id,'charlie-1')"
    serializer := "application/vnd.gremlin-v1.0+json;types=false"

    gremlinInput := &neptunedata.ExecuteGremlinQueryInput{
        GremlinQuery: &gremlinQuery,
        Serializer:   &serializer,
    }
    gremlinResult, err := client.ExecuteGremlinQuery(context.TODO(), gremlinInput)
    if err != nil {
        fmt.Printf("Gremlin query failed: %v\n", err)
    } else {
        var resultMap map[string]interface{}
        err = gremlinResult.Result.UnmarshalSmithyDocument(&resultMap)
        if err != nil {
            fmt.Printf("Error unmarshaling Gremlin result: %v\n", err)
        } else {
            resultJSON, _ := json.MarshalIndent(resultMap, "", "  ")
            fmt.Printf("Gremlin Result: %s\n", string(resultJSON))
        }
    }
}
```

How public endpoints work

When a Neptune instance is publicly accessible:

- Its DNS endpoint resolves to the private IP address from within the DB cluster's VPC.
- It resolves to the public IP address from outside of the cluster's VPC.
- Access is controlled by the security group assigned to the cluster.
- Only instances that are publicly accessible can be accessed via the internet.

Reader endpoint behavior

- If all reader instances are publicly accessible, the reader endpoint will always resolve over the public internet.
- If only some reader instances are publicly accessible, the reader endpoint will resolve publicly only if it selects a publicly accessible instance to serve the read query.

Cluster endpoint behavior

- The DB cluster endpoint always resolves to the instance endpoint of the writer.
- If public endpoint is enabled on the writer instance, the cluster endpoint will be publicly accessible, otherwise it won't be.

Behavior after cluster failover

- A Neptune cluster can have instances on different public accessible setting.
- If a cluster has a public writer and a non-public reader, post a cluster failover, the new writer (previous reader) becomes non-public and the new reader (previous writer) becomes public.

Network configuration requirements

For public endpoints to work properly:

1. The Neptune instances must be in public subnets within your VPC.
2. The route tables associated with these subnets must have a route to an internet gateway for 0.0.0.0/0.

3. The security group must allow access from the public IP addresses or CIDR ranges you want to grant access to.

Restricting public access creation

You can use IAM policies to restrict who can create or modify Neptune clusters with public access. The following example policy denies the creation of Neptune instances with public access:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "rds:CreateDBInstance",
        "rds:ModifyDBInstance",
        "rds:RestoreDBInstanceFromDBSnapshot",
        "rds:RestoreDBInstanceToPointInTime"
      ],
      "Resource": "*",
      "Condition": {
        "Bool": {
          "rds:PubliclyAccessible": true
        }
      }
    }
  ]
}
```

More about the `rds:PublicAccessEnabled` IAM condition key: Amazon RDS Service Authorization Reference

CloudFormation support

You can use CloudFormation to launch Neptune clusters with public endpoints enabled by specifying the `PubliclyAccessible` parameter in your CloudFormation template.

Compatibility with Neptune features

A cluster with public endpoints enabled supports all Neptune features that a VPC-only cluster supports, including:

- Neptune workbench
- Full text search integration
- Neptune Streams
- Custom endpoints
- Neptune Serverless
- Graph Explorer

Pricing

Public endpoints are available at no additional cost beyond standard Neptune pricing. However, connecting from your local environment to Neptune over a public IP might incur increased data transfer costs.

Securing access to an Amazon Neptune cluster

There are multiple ways for you to secure your Amazon Neptune clusters.

Using IAM policies to restrict access to a Neptune DB cluster

To control who can perform Neptune management actions on Neptune DB clusters and DB instances, use AWS Identity and Access Management (IAM).

When you use an IAM account to access the Neptune console, you must first sign in to the AWS Management Console using your IAM account before opening the Neptune console at https://console.aws.amazon.com/neptune/home.

When you connect to AWS using IAM credentials, your IAM account must have IAM policies that grant the permissions required to perform Neptune management operations. For more information, see Using different kinds of IAM policies for controlling access to Neptune.

Using VPC security groups to restrict access to a Neptune DB cluster

Neptune DB clusters must be created in an Amazon Virtual Private Cloud (Amazon VPC). To control which devices and EC2 instances can open connections to the endpoint and port of the DB instance for Neptune DB clusters in a VPC, you use a VPC security group. For more information about VPCs, see Create a security group using the VPC console.

> **Note**
>
> To connect to your Neptune cluster you must expose the cluster's Database port (default of 8182) for both the inbound and outbound rules to allow for proper connectivity.

Using IAM authentication to restrict access to a Neptune DB cluster

If you enable AWS Identity and Access Management (IAM) authentication in a Neptune DB cluster, anyone accessing the DB cluster must first be authenticated. See Authenticating your Amazon Neptune database with AWS Identity and Access Management for information about setting up IAM authentication.

For information about using temporary credentials to authenticate, including examples for the AWS CLI, AWS Lambda, and Amazon EC2, see the section called “Temporary credentials”.

The following links provide additional information about connecting to Neptune using IAM authentication with the individual query languages:

Using Gremlin with IAM authentication

- the section called “Connecting from Gremlin Console”
- the section called “Connecting using Gremlin Java”
- the section called “Gremlin Python”
- the section called “Gremlin Javascript”
- the section called “Gremlin Go”
- the section called “Gremlin .NET”

Using openCypher with IAM authentication

- the section called “Java IAM authentication Bolt example”

- the section called “Python IAM authentication Bolt example”
- the section called “Node.js with IAM authentication”
- the section called “.NET IAM authentication Bolt example”
- the section called “Golang IAM authentication Bolt example”

Using SPARQL with IAM authentication

- the section called “Connecting with SPARQL Java”
- the section called “Conncting with SPARQL and Node.js”
- the section called “Connecting with Python”

Accessing graph data in Amazon Neptune

You can interact with a Amazon Neptune DB cluster after establishing network connectivity. If you have not set up access to your cluster yet, see Connect to a cluster. The following sections describe the tools and query languages you can use to load data, run queries, and manage your graph.

Tools for accessing Neptune

Neptune supports several tools for submitting queries and managing your graph data:

- **AWS CLI** – Use the `aws neptunedata` commands to run Gremlin and openCypher queries, check engine status, manage bulk loads, and more. For more information, see neptunedata in the AWS CLI Command Reference.
- **AWS SDKs** – Use the Neptune Data API through the AWS SDKs to run queries programmatically. SDKs are available for Gremlin and openCypher.
- **`curl` and `awscurl`** – Use `curl` to submit HTTP requests directly to the Neptune endpoints. If IAM authentication is enabled, use awscurl or `curl` 7.75.0+ with the `--aws-sigv4` option to sign requests. For more information, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.
- **Neptune notebooks** – Use Neptune notebooks to run interactive queries in a Jupyter environment with built-in visualizations. For more information, see ???.
- **Drivers** – Connect using language-specific drivers for each query language. For Gremlin, use TinkerPop-compliant drivers available from the Apache TinkerPop project. For openCypher, use

Bolt protocol drivers as described in Using the Bolt protocol. For SPARQL, use compatible tools such as RDF4J for Java as described in Java.

Using a query language to access graph data in your Neptune DB cluster

Once you are connected, you can use the Gremlin and openCypher query languages to create and query a property graph, or the SPARQL query language to create and query a graph containing RDF data.

Graph query languages supported by Neptune

- Gremlin is a graph traversal language for property graphs. A query in Gremlin is a traversal made up of discrete steps, each of which follows an edge to a node. See Gremlin documentation at Apache TinkerPop for more information.

  The Neptune implementation of Gremlin has some differences from other implementations, especially when you are using Gremlin-Groovy (Gremlin queries sent as serialized text). For more information, see Gremlin standards compliance in Amazon Neptune.

  To get started, see Using Gremlin.
- openCypher is a declarative query language for property graphs that was originally developed by Neo4j, then open-sourced in 2015, and contributed to the openCypher project under an Apache 2 open-source license. Its syntax is documented in the Cypher Query Language Reference, Version 9.

  To get started, see Using openCypher.
- SPARQL is a declarative query language for RDF data, based on the graph pattern matching that is standardized by the World Wide Web Consortium (W3C) and described in SPARQL 1.1 Overview and the SPARQL 1.1 Query Language specification.

  To get started, see Using SPARQL.

> **Note**
>
> You can access property graph data in Neptune using both Gremlin and openCypher, but not using SPARQL. Similarly, you can only access RDF data using SPARQL, not Gremlin or openCypher.

Using command-line tools to access Amazon Neptune

You can use the AWS CLI, AWS SDKs, and HTTP tools such as `curl` and `awscurl` to submit queries to your Neptune DB cluster. The following sections show how to set up each tool and run basic queries. You can run Gremlin and openCypher queries with all of these tools. You submit SPARQL queries to the HTTP `/sparql` endpoint using `curl` or `awscurl`.

Using the AWS CLI

The `aws neptunedata` commands let you run Gremlin and openCypher queries, check engine status, manage bulk loads, and more. For the full command reference, see neptunedata in the AWS CLI Command Reference.

The following examples show how to run a basic query:

Gremlin

```
aws neptunedata execute-gremlin-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --region us-east-1 \
  --gremlin-query "g.V().limit(1)"
```

For more information, see execute-gremlin-query in the AWS CLI Command Reference.

openCypher

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --region us-east-1 \
  --open-cypher-query "MATCH (n) RETURN n LIMIT 1"
```

For more information, see execute-open-cypher-query in the AWS CLI Command Reference.

> **Note**
>
> The `aws neptunedata` commands don't include a SPARQL query operation. To run SPARQL queries, submit them to the HTTP `/sparql` endpoint using `curl` or `awscurl`. For more information, see Using `curl` and `awscurl`.

Using the AWS SDK

You can use the Neptune Data API through the AWS SDKs to run queries programmatically. The following Python examples show how to run a basic query:

Gremlin

```
import boto3
import json
from botocore.config import Config

# Disable the client-side read timeout and retries so that
# Neptune's server-side neptune_query_timeout controls query duration.
client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_gremlin_query(
    gremlinQuery='g.V().limit(1)',
    serializer='application/vnd.gremlin-v3.0+json;types=false'
)

print(json.dumps(response['result'], indent=2))
```

For AWS SDK examples in other languages, see AWS SDK.

openCypher

```
import boto3
import json
from botocore.config import Config

# Disable the client-side read timeout and retries so that
```

```
# Neptune's server-side neptune_query_timeout controls query duration.
client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_query(
    openCypherQuery='MATCH (n) RETURN n LIMIT 1'
)

print(json.dumps(response['results'], indent=2))
```

For AWS SDK examples in other languages, see AWS SDK.

> **Note**
>
> The Neptune Data API doesn't include a SPARQL query operation. To run SPARQL queries, submit them to the HTTP `/sparql` endpoint using `curl` or `awscurl`. For more information, see Using `curl` and `awscurl`.

Using `curl` and `awscurl`

The curl command-line tool submits HTTP requests directly to the Neptune endpoints. If IAM authentication is enabled, use awscurl or `curl` 7.75.0+ with the `--aws-sigv4` option to sign requests. For more information, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

Setting up `curl` for HTTPS

To connect using HTTPS (as Neptune requires in most Regions), `curl` needs access to appropriate certificates. For information about how to obtain certificates and format them into a certificate authority (CA) store, see SSL Certificate Verification in the `curl` documentation.

You can specify the location of this CA certificate store using the `CURL_CA_BUNDLE` environment variable. On Windows, `curl` automatically looks for it in a file named `curl-ca-bundle.crt`. It looks first in the same directory as `curl.exe` and then elsewhere on the path. For more information, see SSL Certificate Verification.

As long as `curl` can locate the appropriate certificates, it handles HTTPS connections just like HTTP connections, without extra parameters. Examples in this documentation are based on that scenario.

For more information about the tool, see the curl man page, and the book *Everything curl*.

Query examples

The following examples show how to run a basic query using `curl` and `awscurl`:

Gremlin (awscurl)

```
awscurl https://your-neptune-endpoint:port/gremlin \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d '{"gremlin":"g.V().limit(1)"}'
```

Gremlin (curl)

```
curl -X POST \
  -d '{"gremlin":"g.V().limit(1)"}' \
  https://your-neptune-endpoint:port/gremlin
```

> **Note**
>
> Plain `curl` works only when IAM authentication is not enabled on your cluster.

openCypher (awscurl)

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=MATCH (n) RETURN n LIMIT 1"
```

openCypher (curl)

```
curl -X POST \
```

```
  -d "query=MATCH (n) RETURN n LIMIT 1" \
  https://your-neptune-endpoint:port/openCypher
```

> **Note**
>
> Plain `curl` works only when IAM authentication is not enabled on your cluster.

SPARQL (awscurl)

```
awscurl https://your-neptune-endpoint:port/sparql \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=SELECT ?s ?p ?o WHERE { ?s ?p ?o } LIMIT 1"
```

SPARQL (curl)

```
curl -X POST \
  -d "query=SELECT ?s ?p ?o WHERE { ?s ?p ?o } LIMIT 1" \
  https://your-neptune-endpoint:port/sparql
```

> **Note**
>
> Plain `curl` works only when IAM authentication is not enabled on your cluster.

For more Gremlin HTTP examples, see HTTPS REST. For more openCypher HTTP examples, see HTTPS endpoint. For more SPARQL HTTP examples, see SPARQL HTTP API.

Using Gremlin to access graph data in Amazon Neptune

You can use the Gremlin Console to experiment with TinkerPop graphs and queries in a REPL (read-eval-print loop) environment.

The following tutorial walks you through using the Gremlin console to add vertices, edges, properties, and more to a Neptune graph, highlights some differences in the Neptune-specific Gremlin implementation.

> **Note**
>
> This example assumes that you have completed the following:
>
> - You have connected using SSH to an Amazon EC2 instance.
> - You have created a Neptune cluster as detailed in Create Neptune cluster.
> - You have installed the Gremlin console as described in Installing the Gremlin console.

Using the Gremlin Console

1. Change directories into the folder where the Gremlin console files are unzipped.

```
cd apache-tinkerpop-gremlin-console-3.8.0
```

2. Enter the following command to run the Gremlin Console.

```
bin/gremlin.sh
```

You should see the following output:

```
         \,,,/
         (o o)
-----oOOo-(3)-oOOo-----
plugin activated: tinkerpop.server
plugin activated: tinkerpop.utilities
plugin activated: tinkerpop.tinkergraph
gremlin>
```

You are now at the `gremlin>` prompt. You enter the remaining steps at this prompt.

3. At the `gremlin>` prompt, enter the following to connect to the Neptune DB instance.

```
:remote connect tinkerpop.server conf/neptune-remote.yaml
```

4. At the `gremlin>` prompt, enter the following to switch to remote mode. This sends all Gremlin queries to the remote connection.

```
:remote console
```

5. **Add vertex with label and property.**

```
g.addV('person').property('name', 'justin')
```

The vertex is assigned a `string` ID containing a GUID. All vertex IDs are strings in Neptune.

6. **Add a vertex with custom id.**

```
g.addV('person').property(id, '1').property('name', 'martin')
```

The `id` property is not quoted. It is a keyword for the ID of the vertex. The vertex ID here is a string with the number 1 in it.

Normal property names must be contained in quotation marks.

7. **Change property or add property if it doesn't exist.**

```
g.V('1').property(single, 'name', 'marko')
```

Here you are changing the `name` property for the vertex from the previous step. This removes all existing values from the `name` property.

If you didn't specify `single`, it instead appends the value to the `name` property if it hasn't done so already.

8. **Add property, but append property if property already has a value.**

```
g.V('1').property('age', 29)
```

Neptune uses set cardinality as the default action.

This command adds the `age` property with the value 29, but it does not replace any existing values.

If the `age` property already had a value, this command appends 29 to the property. For example, if the `age` property was 27, the new value would be `[ 27, 29 ]`.

9. **Add multiple vertices.**

```
g.addV('person').property(id, '2').property('name', 'vadas').property('age',
 27).iterate()
```

```
g.addV('software').property(id, '3').property('name', 'lop').property('lang',
 'java').iterate()
g.addV('person').property(id, '4').property('name', 'josh').property('age',
 32).iterate()
g.addV('software').property(id, '5').property('name', 'ripple').property('lang',
 'java').iterate()
g.addV('person').property(id, '6').property('name', 'peter').property('age', 35)
```

You can send multiple statements at the same time to Neptune.

Statements can be separated by newline (`'\n'`), spaces (`'  '`), semicolon (`';  '`), or nothing (for example: `g.addV(‘person’).iterate()g.V()` is valid).

> **Note**
>
> The Gremlin Console sends a separate command at every newline (`'\n'`), so they are each a separate transaction in that case. This example has all the commands on separate lines for readability. Remove the newline (`'\n'`) characters to send it as a single command via the Gremlin Console.

All statements other than the last statement must end in a terminating step, such as `.next()` or `.iterate()`, or they will not run. The Gremlin Console does not require these terminating steps. Use `.iterate` whenever you don't need the results to be serialized.

All statements that are sent together are included in a single transaction and succeed or fail together.

10. **Add edges.**

```
g.V('1').addE('knows').to(__.V('2')).property('weight', 0.5).iterate()
g.addE('knows').from(__.V('1')).to(__.V('4')).property('weight', 1.0)
```

Here are two different ways to add an edge.

11. **Add the rest of the Modern graph.**

```
g.V('1').addE('created').to(__.V('3')).property('weight', 0.4).iterate()
g.V('4').addE('created').to(__.V('5')).property('weight', 1.0).iterate()
g.V('4').addE('knows').to(__.V('3')).property('weight', 0.4).iterate()
```

```
g.V('6').addE('created').to(__.V('3')).property('weight', 0.2)
```

12. **Delete a vertex.**

```
g.V().has('name', 'justin').drop()
```

Removes the vertex with the `name` property equal to `justin`.

> **Important**
>
> *Stop here, and you have the full Apache TinkerPop Modern graph. The examples in the Traversal section of the TinkerPop documentation use the Modern graph.*

13. **Run a traversal.**

```
g.V().hasLabel('person')
```

Returns all `person` vertices.

14. **Run a Traversal with values (valueMap()).**

```
g.V().has('name', 'marko').out('knows').valueMap()
```

Returns key, value pairs for all vertices that `marko` “knows.”

15. **Specify multiple labels.**

```
g.addV("Label1::Label2::Label3")
```

Neptune supports multiple labels for a vertex. When you create a label, you can specify multiple labels by separating them with `::`.

This example adds a vertex with three different labels.

The `hasLabel` step matches this vertex with any of those three labels: `hasLabel("Label1")`, `hasLabel("Label2")`, and `hasLabel("Label3")`.

The `::` delimiter is reserved for this use only.

You cannot specify multiple labels in the `hasLabel` step. For example, `hasLabel("Label1::Label2")` does not match anything.

16. **Specify Time/date**.

```
g.V().property(single, 'lastUpdate', datetime('2018-01-01T00:00:00'))
```

For Gremlin queries sent as text strings (Gremlin Console, HTTP REST, or query strings submitted through a driver), use the `datetime()` function — Neptune doesn't accept raw `java.util.Date` values in string queries. `datetime()` accepts an ISO-8601-compliant `datetime` string.

It supports the following formats: `YYYY-MM-DD`, `YYYY-MM-DDTHH:mm`, `YYYY-MM-DDTHH:mm:SS`, and `YYYY-MM-DDTHH:mm:SSZ`.

> **Neptune 1.5.0.0: GLV datetime values**
>
> Starting with TinkerPop 3.8.0, the Java GLV returns datetime values as `java.time.OffsetDateTime` instead of `java.util.Date`. Other GLVs continue to use their language-native datetime type. The `datetime()` function shown above is for string-mode queries and is unaffected.
> If you are using a GLV, use the language-native datetime type rather than `java.util.Date`. For example:
>
> - Java: `java.time.OffsetDateTime`
> - Python: `datetime.datetime`
> - JavaScript: `Date`
> - .NET: `System.DateTime`
> - Go: `time.Time`
>
> For more information, see Using Native Date and Time for GLV Time Data.

17. **Delete vertices, properties, or edges.**

```
g.V().hasLabel('person').properties('age').drop().iterate()
g.V('1').drop().iterate()
```

```
g.V().outE().hasLabel('created').drop()
```

Here are several drop examples.

> **Note**
>
> The `.next()` step does not work with `.drop()`. Use `.iterate()` instead.

18. When you are finished, enter the following to exit the Gremlin Console.

```
:exit
```

> **Note**
>
> Use a semicolon (`;`) or a newline character (\n) to separate each statement.
> Each traversal preceding the final traversal must end in `iterate()` to be executed. Only the data from the final traversal is returned.

Using openCypher to access graph data in Amazon Neptune

Using SPARQL to access graph data in Amazon Neptune

SPARQL is a query language for the Resource Description Framework (RDF), which is a graph data format designed for the web. Amazon Neptune is compatible with SPARQL 1.1. This means that you can connect to a Neptune DB instance and query the graph using the query language described in the SPARQL 1.1 Query Language specification.

A query in SPARQL consists of a `SELECT` clause to specify the variables to return and a `WHERE` clause to specify which data to match in the graph. If you are unfamiliar with SPARQL queries, see Writing Simple Queries in the SPARQL 1.1 Query Language.

The HTTP endpoint for SPARQL queries to a Neptune DB instance is `https://`*`your-neptune-endpoint`*`:`*`port`*`/sparql`.

To connect to SPARQL

1. You can get the SPARQL endpoint for your Neptune cluster from the **SparqlEndpoint** item in the **Outputs** section of the CloudFormation stack.
2. Enter the following to submit a SPARQL **UPDATE** using HTTP POST and the **curl** command.

```
curl -X POST --data-binary 'update=INSERT DATA { <https://test.com/s> <https://
test.com/p> <https://test.com/o> . }' https://your-neptune-endpoint:port/sparql
```

   The preceding example inserts the following triple into the SPARQL default graph: `<https://test.com/s> <https://test.com/p> <https://test.com/o>`

3. Enter the following to submit a SPARQL **QUERY** using HTTP POST and the **curl** command.

```
curl -X POST --data-binary 'query=select ?s ?p ?o where {?s ?p ?o} limit 10'
 https://your-neptune-endpoint:port/sparql
```

   The preceding example returns up to 10 of the triples (subject-predicate-object) in the graph by using the `?s ?p ?o` query with a limit of 10. To query for something else, replace it with another SPARQL query.

> **Note**
>
> The default MIME type of a response is `application/sparql-results+json` for `SELECT` and `ASK` queries.
> The default MIME type of a response is `application/n-quads` for `CONSTRUCT` and `DESCRIBE` queries.
> For a list of all available MIME types, see SPARQL HTTP API.

Loading Data into an Amazon Neptune cluster

Amazon Neptune provides a process for loading data from external files directly into a Neptune DB instance. You can use this process instead of executing a large number of `INSERT` statements, `addV` and `addE` steps, or other API calls.

The following are links to additional loading information.

- **Methods for loading data** – *Loading data*

- **Data formats supported by the bulk loader** – the section called "Data Formats"
- **Loading example** – the section called "Loading Example"

Monitoring an Amazon Neptune cluster

Amazon Neptune supports the following monitoring methods.

- **Amazon CloudWatch** – Amazon Neptune automatically sends metrics to CloudWatch and also supports CloudWatch Alarms. For more information, see the section called "Using CloudWatch".
- **AWS CloudTrail** – Amazon Neptune supports API logging using CloudTrail. For more information, see the section called "Logging Neptune API Calls with AWS CloudTrail".
- **Tagging** – Use tags to add metadata to your Neptune resources and track usage based on tags. For more information, see the section called "Tagging Neptune resources".
- **Audit log files** – View, download, or watch database log files using the Neptune console. For more information, see the section called "Audit Logs with Neptune".
- **Instance status** – Check the health of a Neptune instance's graph database engine, find out what version of the engine is installed, and obtain other engine status information using the instance status API.

Troubleshooting an Amazon Neptune cluster

The following links might be helpful for resolving issues with Amazon Neptune.

- **Best practices** – For solutions to common issues and performance suggestions, see *Best practices*.
- **Service errors** – For a list of errors for both management APIs and graph database connections, see *Neptune Errors*.
- **Service limits** – For information about Neptune limits, see *Neptune Limits*.
- **Engine releases** – For information about graph engine releases, including release notes, see *Engine releases*.
- **Support forums** – To join a discussion about Neptune, see the Amazon Neptune Forum.
- **Pricing** – For information about the costs of using Amazon Neptune, see Amazon Neptune Pricing.
- **AWS Support** – For help and guidance from the experts, see AWS Support.

Amazon Neptune Serverless

Amazon Neptune Serverless is an on-demand autoscaling configuration that is architected to scale your DB cluster as needed to meet even very large increases in processing demand, and then scale down again when the demand decreases. It helps to automate the processes of monitoring workload and adjusting capacity for your Neptune database. Because capacity is adjusted automatically based on application demand, you're charged only for the resources that your application actually needs.

Use cases for Neptune Serverless

Neptune Serverless supports many types of workloads. It is suitable for demanding, highly variable workloads and can be very helpful if your database usage is typically heavy for short periods of time, followed by long periods of light activity or no activity at all. Neptune Serverless is especially useful for the following use cases:

- **Variable workloads** – Workloads that have sudden and unpredictable increases in CPU activity. With Neptune Serverless, your graph database automatically scales capacity to meet the needs of the workload and scales back down when the surge of activity is over. You no longer have to provision for peak or average capacity. You can specify an upper capacity limit to handle peak workloads, and that capacity isn't used unless it's needed.

  The granularity of scaling provided by Neptune Serverless helps you match capacity closely to your workload's needs. Neptune Serverless can add or remove capacity in fine grained increments based on what is needed. It can add as little as half a Neptune Capacity Unit (NCU) when only a little more capacity is required.
- **Multi-tenant applications** – By taking advantage of Neptune Serverless, you can create a separate DB cluster for each of the applications you need to run without having to manage those tenant clusters individually. Each of the tenant clusters may have different busy and idle periods depending on multiple factors, but Neptune Serverless can scale them efficiently without your intervention.
- **New applications** – When you deploy a new application, you're often unsure how much database capacity it will need. Using Neptune Serverless, you can set up a DB cluster that can scale automatically to meet the new application's capacity requirements as they develop.
- **Capacity planning** – Suppose you usually adjust your database capacity, or verify the optimal database capacity for your workload, by modifying the DB instance classes of all the DB

instances in a cluster. With Neptune Serverless, you can avoid this administrative overhead. Instead, you can modify existing DB instances from provisioned to serverless or from serverless to provisioned without having to create a new DB cluster or instance.

- **Development and testing** – Neptune Serverless is also perfect for development and testing environments. With Neptune Serverless you can create DB instances with a high enough maximum capacity to test your most demanding application, and a low minimum capacity for all the other times when the system may be idle between tests.

Neptune Serverless only scales compute capacity. Your storage volume remains the same, and is not affected by serverless scaling.

> **Note**
>
> You can also use Neptune auto-scaling with Neptune Serverless to handle different kinds of workload variations.

Amazon Neptune Serverless constraints

- Neptune Serverless is only available in the following regions:
  - US East (N. Virginia): `us-east-1`
  - US East (Ohio): `us-east-2`
  - US West (N. California): `us-west-1`
  - US West (Oregon): `us-west-2`
  - Canada (Central): `ca-central-1`
  - Europe (Stockholm): `eu-north-1`
  - Europe (Spain): `eu-south-2`
  - Europe (Ireland): `eu-west-1`
  - Europe (London): `eu-west-2`
  - Europe (Paris): `eu-west-3`
  - Europe (Frankfurt): `eu-central-1`
  - Europe (Zurich): `eu-central-2`
  - Asia Pacific (Tokyo): `ap-northeast-1`

- Asia Pacific (Seoul): `ap-northeast-2`
- Asia Pacific (Singapore): `ap-southeast-1`
- Asia Pacific (Sydney): `ap-southeast-2`
- Asia Pacific (Jakarta): `ap-southeast-3`
- Asia Pacific (Hong Kong): `ap-east-1`
- Asia Pacific (Mumbai): `ap-south-1`
- Asia Pacific (Hyderabad): `ap-south-2`
- South America (São Paulo): `sa-east-1`

- **Not available in early engine versions** – Neptune Serverless is only available in engine releases 1.2.0.1 or later.
- **Not compatible with the Neptune lookup cache** – The lookup cache does not work with serverless DB instances.
- **Maximum memory in a serverless instance is 256 GB** – Setting `MaxCapacity` to 128 NCUs (the highest supported setting) allows a Neptune Serverless instance to scale to 256 GB of memory, which is equivalent to that of an `R6g.8XL` provisioned instance type.

Capacity scaling in a Neptune Serverless DB cluster

Setting up a Neptune Serverless DB cluster is similar to setting up a normal provisioned cluster, with additional configuration for minimum and maximum units for scaling, and with the instance type set to `db.serverless`. The scaling configuration is defined in Neptune Capacity Units (NCUs), each of which consists of 2 GiB (gibibyte) of memory (RAM) along with associated virtual processor capacity (vCPU) and networking. It is set as a part of a `ServerlessV2ScalingConfiguration` object, represented in JSON like this:

```
"ServerlessV2ScalingConfiguration": {
  "MinCapacity": (minimum NCUs, a floating-point number such as   1.0),
  "MaxCapacity": (maximum NCUs, a floating-point number such as 128.0)
}
```

At any moment in time, each Neptune writer or reader instance has a capacity measured by a floating-point number that represents the number of NCUs currently being used by that instance. You can use the CloudWatch ServerlessDatabaseCapacity metric at an instance level to find out you how many NCUs a given DB instance is currently using, and the NCUUtilization metric to find

out what percentage of its maximum capacity the instance is using. Both of these metrics are also available at a DB cluster level to show average resource utilization for the DB cluster as a whole.

When you create a Neptune Serverless DB cluster, you set both the minimum and the maximum number of **Neptune capacity units** (NCUs) for all the serverless instances.

The minimum NCU value that you specify sets the smallest size to which a serverless instance in your DB cluster can shrink, and likewise, the maximum NCU value establishes the largest size to which a serverless instance can grow. The highest maximum NCU value you can set is 128.0 NCUs, and the lowest minimum is 1.0 NCUs.

Neptune continuously tracks the load on each Neptune Serverless instance by monitoring its utilization of resources such as CPU, memory, and network. The load is generated by your application's database operations, by background processing for the server, and by other administrative tasks.

When the load on a serverless instance reaches the limit of current capacity, or when Neptune detects any other performance issues, the instance scales up automatically. When the load on the instance declines, the capacity scales down towards the configured minimum capacity units, with CPU capacity being released before memory. This architecture allows releasing of resources in a controlled step-down manner and handles demand fluctuations effectively.

You can make a reader instance scale together with the writer instance or scale independently by setting its promotion tier. Reader instances in promotion tiers 0 and 1 scale at the same time as the writer, which keeps them sized at the right capacity to take over the workload from the writer rapidly in case of failover. Readers in promotion tiers 2 through 15 scale independently of the writer instance, and of each other.

If you've created your Neptune DB cluster as a Multi-AZ cluster to ensure high availability, Neptune Serverless scales instances in all AZs up and down with your database load. You can set the promotion tier of a reader instance in a secondary AZ to 0 or 1 so that it scales up and down along with the capacity of the writer instance in the primary AZ so that it's ready to take over the current workload at any time.

> **Note**
>
> Storage for a Neptune DB cluster consists of six copies of all your data, spread across three AZs, regardless of whether you created the cluster as a Multi-AZ cluster or not. Storage replication is handled by the storage subsystem and is not affected by Neptune Serverless.

Choosing a minimum capacity value for a Neptune Serverless DB cluster

The smallest value you can set for the minimum capacity is `1.0` NCUs.

Be sure not to set the minimum value lower than what your application requires to operate efficiently. Setting it too low can result in a higher rate of timeouts in certain memory-intensive workloads.

Setting the minimum value as low as possible can save money, since your cluster will use minimal resources when demand is low. However, if your workload tends to fluctuate dramatically, from very low to very high, you may want to set the minimum higher, because a higher minimum lets your Neptune Serverless instances scale up faster.

The reason for this is that Neptune chooses scaling increments based on current capacity. If current capacity is low, Neptune will initially scale up slowly. If the minimum is higher, Neptune starts with a larger scaling increment, and can therefore scale up faster to meet a large sudden increase in workload.

Choosing a maximum capacity value for a Neptune Serverless DB cluster

The largest value you can set for the maximum capacity is `128.0` NCUs, and the smallest value you can set for the maximum capacity is `2.5` NCUs. Whatever maximum capacity value you set must be at least as large as the minimum capacity value you set.

As a general rule, set the maximum value high enough to handle the peak load that your application is likely to encounter. Setting it too low can result in a higher rate of timeouts in certain memory-intensive workloads.

Setting the maximum value as high as possible has the advantage that your application is likely to be able to handle even the most unexpected workloads. The disadvantage is that you lose some ability to predict and control resource costs. An unexpected spike in demand can end up costing much more than your budget has anticipated.

The benefit of a carefully targeted maximum value is that it lets you meet peak demand while also putting a cap on Neptune compute costs.

> **Note**
>
> Changing the capacity range of a Neptune Serverless DB cluster causes changes to the default values of some configuration parameters. Neptune can apply some of those new defaults immediately, but some of the dynamic parameter changes take effect only after a reboot. A `pending-reboot` status indicates that you need a reboot to apply some parameter changes.

Use your existing configuration to estimate serverless requirements

If you typically modify the DB instance class of your provisioned DB instances to meet exceptionally high or low workload, you can use that experience to make a rough estimate of the equivalent Neptune Serverless capacity range.

Estimate the best minimum capacity setting

You can apply what you know about your existing Neptune DB cluster to estimate the serverless minimum capacity setting that will work best.

For example, if your provisioned workload has memory requirements that are too high for small DB instance classes such as T3 or T4g, choose a minimum NCU setting that provides memory comparable to an R5 or R6g DB instance class.

Or, suppose that you use the `db.r6g.xlarge` DB instance class when your cluster has a low workload. That DB instance class has 32 GiB of memory, so you can specify a minimum NCU setting of 16 to create serverless instances that can scale down to approximately that same capacity (each NCU corresponds to about 2 GiB of memory). If your `db.r6g.xlarge` instance is sometimes underutilized, you might be able to specify a lower value.

If your application works most efficiently when your DB instances can hold a given amount of data in memory or the buffer cache, consider specifying a minimum NCU setting large enough to provide enough memory for that. Otherwise, data may be evicted from the buffer cache when the serverless instances scale down, and will have to be read back into the buffer cache over time when instances scale back up. If the amount of I/O to bring data back into the buffer cache is substantial, choosing a higher minimum NCU value could be worthwhile.

If you find that your serverless instances are running most of the time at a particular capacity, it works well to set the minimum capacity just a little lower than that. Neptune Serverless can

efficiently estimate how much and how fast to scale up when the current capacity isn't drastically lower than the required capacity.

In a mixed configuration, with a provisioned writer and Neptune Serverless readers, the readers don't scale along with the writer. Because they scale independently, setting a low minimum capacity for them can result in excessive replication lag. They may not have sufficient capacity to keep up with changes the writer is making when there is a highly write-intensive workload. In this situation, set a minimum capacity that's comparable to the writer capacity. In particular, if you observe replica lag in readers that are in promotion tiers 2–15, increase the minimum capacity setting for your cluster.

Estimate the best maximum capacity setting

You can also apply what you know about your existing Neptune DB cluster to estimate the serverless maximum capacity setting that will work best.

For example, suppose that you use the `db.r6g.4xlarge` DB instance class when your cluster has a high workload. That DB instance class has 128 GiB of memory, so you can specify a maximum NCU setting of 64 to set up equivalent Neptune Serverless instances (each NCU corresponds to about 2 GiB of memory). You could specify a higher value to let the DB instance scale up further in case your `db.r6g.4xlarge` instance can't always handle the workload.

If unexpected spikes in your workload are rare, it may make sense to set your maximum capacity high enough to maintain application performance even during those spikes. On the other hand, you may want to set a lower maximum capacity that can reduce throughput during unusual spikes but that allows Neptune to handle your expected workloads without problem, and that limits costs.

Additional configuration for Neptune Serverless DB clusters and instances

In addition to setting minimum and maximum capacity for your Neptune Serverless DB cluster, there are a few other configuration choices to consider.

Combining serverless and provisioned instances in a DB cluster

A DB cluster doesn't have to be serverless only— you can create a combination of serverless and provisioned instances (a mixed configuration).

For example, suppose that you need more write capacity than is available in a serverless instance. In that case, you can set up the cluster with a very large provisioned writer and still use serverless instances for the readers.

Or, suppose that the write workload on your cluster varies but the read workload is steady. In that case, you could set up your cluster with a serverless writer and one or more provisioned readers.

See Using Amazon Neptune Serverless for information about how to create a mixed-configuration DB cluster.

Setting the promotion tiers for Neptune Serverless instances

For clusters containing multiple serverless instances, or a mixture of provisioned and serverless instances, pay attention to the promotion tier setting for each serverless instance. This setting controls more behavior for serverless instances than for provisioned DB instances.

In the AWS Management Console, you specify this setting using the **Failover priority** under **Additional configuration** on the **Create database**, **Modify instance**, and **Add reader** pages. You see this property for existing instances in the optional **Priority tier** column on the **Databases** page. You can also see this property on the details page for a DB cluster or instance.

For provisioned instances, the choice of tier 0–15 determines only the order in which Neptune chooses which reader instance to promote to the writer during a failover operation. For Neptune Serverless reader instances, the tier number also determines whether the instance scales up to match the capacity of the writer instance or scales independently of it based only on its own workload.

Neptune Serverless reader instances in tier 0 or 1 are kept at a minimum capacity at least as high as the writer instance so that they are ready to take over from the writer in case of failover. If the writer is a provisioned instance, Neptune estimates the equivalent serverless capacity and uses that estimate as the minimum capacity for the serverless reader instance.

Neptune Serverless reader instances in tiers 2–15 don't have the same constraint on their minimum capacity, and scale independently of the writer. When they are idle, they scale down to the minimum NCU value specified in the cluster's capacity range. This can cause problems, however, if the read workload spikes rapidly.

Keeping reader capacity aligned to writer capacity

One important thing to keep in mind is that you want to make sure your reader instances can keep up with your writer instance, to prevent excessive replication lag. This is particularly a concern in two situations, where serverless reader instances do not automatically scale in sync with the writer instance:

- When your writer is provisioned, and your readers are serverless.
- When your writer is serverless, and your serverless readers are in promotion tiers 2-15.

In both of those cases, set the minimum serverless capacity to match that of the expected writer capacity, to ensure that reader operations don't time out and potentially cause restarts. In the case of a provisioned writer instance, set the minimum capacity to match that of the provisioned instance. In the case of a serverless writer, the optimal setting may be harder to predict.

Because the instance capacity range is set at the cluster level, all serverless instances are controlled by the same minimum and maximum capacity settings. Reader instances in tiers 0 and 1 scale in sync with the writer instance, but instances in promotion tiers 2-15 scale independently of each other and of the writer instance, depending on their workload. If you set the minimum capacity too low, idle instances in tiers 2 to 15 can scale down too low to scale back up fast enough to handle a sudden burst in writer activity.

Avoid setting the timeout value too high

It is possible to incur unexpected costs if you set the query timeout value too high on a serverless instance.

Without a reasonable timeout setting, you may inadvertently issue a query that requires a powerful, expensive instance type and that keeps running for a very long time, incurring costs you never anticipated. You can avoid that situation by using a query timeout value which accomodates most of your queries and only causes unexpectedly long-running ones to time out.

This is true both for general query timeout values set using parameters and for per-query timeout values set using query hints.

Optimizing your Neptune Serverless configuration

If your Neptune Serverless DB cluster is not tuned to the workload it is running, you may notice that it doesn't run optimally. You can adjust the minimum and/or maximum capacity setting so that it can scale without encountering memory problems.

- Increase the minimum capacity setting for the cluster. This can correct the situation where an idle instance scales back to a capacity that has less memory than your application and enabled features need.
- Increase the maximum capacity setting for the cluster. This can correct the situation where a busy database can't scale up to a capacity with enough memory to handle the workload and any memory-intensive features that are enabled.
- Change the workload on the instance in question. For example, you can add reader instances to the cluster to spread the read load across more instances.
- Tune your application's queries so that they use fewer resources.
- Try using a provisioned instance that is larger than the maximum NCUs available within Neptune Serverless, to see if it is a better fit for the memory and CPU requirements of the workload.

Using Amazon Neptune Serverless

You can create a new Neptune DB cluster as a serverless one, or in some cases you can convert an existing DB cluster to use serverless. You can also convert DB instances in a serverless DB cluster to and from serverless instances. You can only use Neptune Serverless in one of the AWS Regions where it's supported, with a few other limitations (see Amazon Neptune Serverless constraints).

You can also use the Neptune CloudFormation stack to create a Neptune Serverless DB cluster.

Creating a new DB cluster that uses Serverless

To create a Neptune DB cluster that uses serverless, you can do so using the AWS Management Console the same way you do to create a provisioned cluster. The difference is that under **DB instance size**, you need to set the **DB instance class** to **serverless**. When you do that, you then need to set the serverless capacity range for the cluster.

You can also create a serverless DB cluster using the AWS CLI with commands like this (on Windows, replace '\' with '^'):

```
aws neptune create-db-cluster \
  --region (an AWS Region region that supports serverless) \
  --db-cluster-identifier (ID for the new serverless DB cluster) \
  --engine neptune \
  --engine-version (optional: 1.2.0.1 or above) \
  --serverless-v2-scaling-configuration "MinCapacity=1.0, MaxCapacity=128.0"
```

You could also specify the `serverless-v2-scaling-configuration` parameter like this:

```
  --serverless-v2-scaling-configuration '{"MinCapacity":1.0, "MaxCapacity":128.0}'
```

You can then run the `describe-db-clusters` command for the `ServerlessV2ScalingConfiguration` attribute, which should return the capacity range settings you specified:

```
"ServerlessV2ScalingConfiguration": {
    "MinCapacity": (the specified minimum number of NCUs),
    "MaxCapacity": (the specified maximum number of NCUs)
}
```

Converting an existing DB cluster or instance to Serverless

If you have a Neptune DB cluster that is using engine version 1.2.0.1 or above, you can convert it to be serverless. This process does incur some downtime.

The first step is to add a capacity range to the existing cluster. You can do so using the AWS Management Console, or by using a AWS CLI command like this (on Windows, replace '\' with '^'):

```
aws neptune modify-db-cluster \
  --db-cluster-identifier (your DB cluster ID) \
  --serverless-v2-scaling-configuration \
      MinCapacity=(minimum number of NCUs, such as  2.0), \
      MaxCapacity=(maximum number of NCUs, such as 24.0)
```

The next step is to create a new serverless DB instance to replace the existing primary instance (the writer) in the cluster. Again, you can do this and all the subsequent steps using either the AWS Management Console or the AWS CLI. In either case, specify the DB instance class as serverless. The AWS CLI command would look like this (on Windows, replace '\' with '^'):

```
aws neptune create-db-instance \
  --db-instance-identifier (an instance ID for the new writer instance) \
  --db-cluster-identifier (ID of the DB cluster) \
  --db-instance-class db.serverless \
  --engine neptune
```

When the new writer instance has become available, perform a failover to make it the writer instance for the cluster:

```
aws neptune failover-db-cluster \
  --db-cluster-identifier (ID of the DB cluster) \
  --target-db-instance-identifier (instance ID of the new serverless instance)
```

Next, delete the old writer instance:

```
aws neptune delete-db-instance \
  --db-instance-identifier (instance ID of the old writer instance) \
  --skip-final-snapshot
```

Finally, do the same thing to create a new serverless instance to take the place of each existing provisioned reader instance that you would like to turn into a serverless instance, and delete the existing provisioned instances (no failover is needed for reader instances).

Modifying the capacity range of an existing serverless DB cluster

You can change the capacity range of a Neptune Serverless DB cluster using the AWS CLI like this (on Windows, replace '\' with '^'):

```
aws neptune modify-db-cluster \
  --region (an AWS region that supports serverless) \
  --db-cluster-identifier (ID of the serverless DB cluster) \
  --apply-immediately \
  --serverless-v2-scaling-configuration MinCapacity=4.0, MaxCapacity=32
```

Changing the capacity range causes changes to the default values of some configuration parameters. Neptune can apply some of those new defaults immediately, but some of the dynamic parameter changes take effect only after a reboot. A status of `pending-reboot` indicates that you need a reboot to apply some parameter changes.

Changing a Serverless DB instance to provisioned

All you need to do to convert a Neptune Serverless instance to a provisioned one is to change its instance class to one of the provisioned instance classes. See Modifying a Neptune DB Instance (and Applying Immediately).

Configuring Gremlin clients for Serverless

When using Gremlin WebSocket clients with Neptune Serverless, you need to configure the client's heartbeat interval appropriately to maintain stable connections during scaling events. For detailed configuration instructions for Java, Go, JavaScript/Node.js, and Python clients, see Heartbeat Configuration for Neptune Serverless.

Monitoring serverless capacity with Amazon CloudWatch

You can use CloudWatch to to monitor the capacity and utilization of the Neptune serverless instances in your DB cluster. There are two CloudWatch metrics that let you track current serverless capacity both at the cluster level and at the instance level:

- **`ServerlessDatabaseCapacity`** – As an instance-level metric, `ServerlessDatabaseCapacity` reports the current instance capacity, in NCUs. As a cluster-level metric, it reports the average of all the `ServerlessDatabaseCapacity` values of all the DB instances in the cluster.
- **`NCUUtilization`** – This metric reports the percentage of possible capacity being used. It is calculated as the current `ServerlessDatabaseCapacity` (either at the instance level or at the cluster level) divided by the maximum capacity setting for the DB cluster.

  If this metric approaches 100% at a cluster level, meaning that the cluster has scaled as high as it can, consider increasing the maximum capacity setting.

  If it approaches 100% for a reader instance while the writer instance is not near maximum capacity, consider adding more reader instances to distribute the read workload.

Note that the `CPUUtilization` and `FreeableMemory` metrics have slightly different meanings for serverless instances than for provisioned instances. In a serverless context, `CPUUtilization` is a percentage that's calculated as the amount of CPU currently being used divided by the amount of CPU that would be available at maximum capacity. Similarly, `FreeableMemory` reports the amount of freeable memory that would be available if an instance was at maximum capacity.

The following example shows how to use the AWS CLI on Linux to retrieve the minimum, maximum, and average capacity values for a given DB instance, measured every 10 minutes over one hour. The Linux `date` command specifies the start and end times relative to the current date and time. The `sort_by` function in the `--query` parameter sorts the results chronologically based on the `Timestamp` field:

```
aws cloudwatch get-metric-statistics \
  --metric-name "ServerlessDatabaseCapacity" \
  --start-time "$(date -d '1 hour ago')" \
  --end-time "$(date -d 'now')" \
  --period 600 \
  --namespace "AWS/Neptune"
  --statistics Minimum Maximum Average \
  --dimensions Name=DBInstanceIdentifier,Value=(instance ID) \
  --query 'sort_by(Datapoints[*].
{min:Minimum,max:Maximum,avg:Average,ts:Timestamp},&ts)' \
  --output table
```

Overview of Amazon Neptune features

> **Note**
>
> This section does not cover using the query languages that you can use to access the data in a Neptune graph.
> For information about how to connect to a running Neptune DB cluster with Gremlin, see Accessing a Neptune graph with Gremlin.
> For information about how to connect to a running Neptune DB cluster with openCypher, see Accessing the Neptune Graph with openCypher.
> For information about how to connect to a running Neptune DB cluster with SPARQL, see Accessing the Neptune graph with SPARQL.

This section provides an overview of specific Neptune features, including:

- Neptune compliance with query-language standards.
- Neptune's graph data model.
- An explanation of Neptune transaction semantics.
- An introduction to Neptune clusters and instances.
- Neptune's storage, reliability and availability.
- An explanation of Neptune endpoints.
- Using Neptune's *lab mode* to enable experimental features.
- A description of Neptune's DFE engine.
- Neptune's JDBC connectivity.
- A list of Neptune engine releases and how to update your engine.

Notes on Amazon Neptune Standards Compliance

Amazon Neptune complies with applicable standards in implementing the Gremlin and SPARQL graph query languages in most cases.

These sections describe the standards as well as those areas where Neptune extends or diverges from them.

Topics

- Gremlin standards compliance in Amazon Neptune
- SPARQL standards compliance in Amazon Neptune
- openCypher specification compliance in Amazon Neptune

Gremlin standards compliance in Amazon Neptune

The following sections provide an overview of the Neptune implementation of Gremlin and how it differs from the Apache TinkerPop implementation.

Neptune implements some Gremlin steps natively in its engine, and uses the Apache TinkerPop Gremlin implementation to process others (see Native Gremlin step support in Amazon Neptune).

> **Note**
>
> For some concrete examples of these implementation differences shown in Gremlin Console and Amazon Neptune, see the the section called “Using Gremlin” section of the Quick Start.

Topics

- Applicable Standards for Gremlin
- Variables and parameters in scripts
- Script execution
- Properties on elements
- Meta-properties
- Ordering of vertices, edges, and properties
- Datetime offsets

- Sessions
- Transactions
- Vertex and edge IDs
- User-supplied IDs
- Vertex property IDs
- Cardinality of vertex properties
- Updating a vertex property
- Labels
- Escape characters
- Serialization
- Lambda steps
- Unsupported Gremlin steps
- Gremlin graph features in Neptune

Applicable Standards for Gremlin

- The Gremlin language is defined by Apache TinkerPop Documentation and the Apache TinkerPop implementation of Gremlin rather than by a formal specification.
- For numeric formats, Gremlin follows the IEEE 754 standard (IEEE 754-2019 - IEEE Standard for Floating-Point Arithmetic. For more information, also see the Wikipedia IEEE 754 page).

Variables and parameters in scripts

Where pre-bound variables are concerned, the traversal object `g` is Pre-bound in Neptune, and the `graph` object is not supported.

Although Neptune does not support Gremlin variables or parameterization in scripts, you may often encounter sample scripts for Gremlin Server on the Internet that contain variable declarations, such as:

```
String query = "x = 1; g.V(x)";
List<Result> results = client.submit(query).all().get();
```

There are also many examples that make use of parameterization (or bindings) when submitting queries, such as:

```
Map<String,Object> params = new HashMap<>();
params.put("x",1);
String query = "g.V(x)";
List<Result> results = client.submit(query).all().get();
```

The parameter examples are usually associated with warnings about performance penalties for not parameterizing when possible. There are a great many such examples for TinkerPop that you may encounter, and they all sound quite convincing about the need to parameterize.

However, both the variables declarations feature and the parameterization feature (along with the warnings) only apply to TinkerPop's Gremlin Server when it is using the `GremlinGroovyScriptEngine`. They do not apply when Gremlin Server uses Gremlin's `gremlin-language` ANTLR grammar to parse queries. The ANTLR grammar doesn't support either variable declarations or parameterization, so when using ANTLR, you don't have to worry about failing to parameterize. Because the ANTLR grammar is a newer component of TinkerPop, older content you may encounter on the Internet doesn't generally reflect this distinction.

Neptune uses the ANTLR grammar in its query processing engine rather than the `GremlinGroovyScriptEngine`, so it does not support variables or parameterization or the `bindings` property. As a result, the problems related to failing to parameterize do not apply in Neptune. Using Neptune, it's perfectly safe simply to submit the query as-is where one would normally parameterize. As a result, the previous example can be simplified without any performance penalty as follows:

```
String query = "g.V(1)";
List<Result> results = client.submit(query).all().get();
```

Script execution

Neptune's Gremlin engine parses queries using TinkerPop's `gremlin-language` ANTLR grammar. It does *not* run a `GremlinGroovyScriptEngine` (as some TinkerPop-based Gremlin Server deployments do), so scripts submitted to Neptune must contain only the Gremlin language — not arbitrary Groovy or Java code.

Scripts can be sent to Neptune in a variety of ways, such as through the Gremlin REST endpoint, the Gremlin Console, or via TinkerPop language drivers (for example, the Java driver's script client). The constraints described in this section apply to any of these text-string submission paths.

It's important not to confuse the Gremlin language itself with the syntactic sugar or general-purpose functions of whatever programming language you may have seen wrapping Gremlin examples elsewhere. Where such code appears in TinkerPop tutorials or online samples, it depends on a Groovy or Java runtime that Neptune does not provide.

> **Important**
>
> Everything in this section applies to text-string Gremlin submissions. GLV (Gremlin Language Variant) bytecode submissions built in a host language such as Java, Python, or .NET are not subject to these constraints, because the host-language traversal builder produces bytecode that Neptune's engine consumes directly.

What a script may contain

- All queries must begin with g, the traversal object.
- Multiple traversals can be issued in a single submission separated by a semicolon (`;`) or a newline character (\n). Every statement other than the last must end with an `.iterate()` step to be executed; only the final traversal's data is returned.

Referencing TinkerPop enumeration values

Where a TinkerPop enumeration value is expected as a step argument (for example, a cardinality on `property()` or an order on `by()`), use the short-form values recognized by the ANTLR grammar. Neptune does not resolve fully qualified Java class names in this position — for example, `org.apache.tinkerpop.gremlin.structure.VertexProperty.Cardinality.single` is not accepted; use `single` instead.

The following table lists the allowed short-form values and the underlying TinkerPop class each one belongs to.

Allowed Values	Class
`id`, `key`, `label`, `value`	org.apache.tinkerpop.gremlin.structure.T
`T.id`, `T.key`, `T.label`, `T.value`	org.apache.tinkerpop.gremlin.structure.T

`set`, `single`	org.apache.tinkerpop.gremlin.structure.VertexProperty.Cardinality
`asc`, `desc`, `shuffle`	org.apache.tinkerpop.gremlin.process.traversal.Order
`Order.asc` , `Order.desc` , `Order.shuffle`	org.apache.tinkerpop.gremlin.process.traversal.Order
`global`, `local`	org.apache.tinkerpop.gremlin.process.traversal.Scope
`Scope.global` , `Scope.local`	org.apache.tinkerpop.gremlin.process.traversal.Scope
`all`, `first`, `last`, `mixed`	org.apache.tinkerpop.gremlin.process.traversal.Pop
`normSack`	org.apache.tinkerpop.gremlin.process.traversal.SackFunctions.Barrier
`addAll`, `and`, `assign`, `div`, `max`, `min`, `minus`, `mult`, `or`, `sum`, `sumLong`	org.apache.tinkerpop.gremlin.process.traversal.Operator
`keys`, `values`	org.apache.tinkerpop.gremlin.structure.Column
`BOTH`, `IN`, `OUT`	org.apache.tinkerpop.gremlin.structure.Direction
`any`, `none`, `unproductive`	org.apache.tinkerpop.gremlin.process.traversal.step.TraversalOptionParent.Pick

What a script may not contain

The following are **not** supported in text-string Gremlin queries to Neptune, because they rely on Groovy or Java runtime support that Neptune does not provide:

- **Groovy statements that don't begin with g.** This includes:

  - Arithmetic expressions such as `1 + 1`
  - System calls such as `System.nanoTime()`
  - Variable declarations such as `x = 1; g.V(x)`
- **Java method or library calls other than supported Gremlin APIs.** For example, `java.lang.*`, `Date()`, and `g.V().tryNext().orElseGet(...)` are not allowed.
- **Gremlin methods that take a Java type as an argument.** These are only reachable from a JVM-language host, not from a text-string submission. Examples:
  - `org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversal.progr`
  - `org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversal.sideE`
  - `org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversal.from(`
  - `org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversal.to(or`

  For example, the following traversal cannot be submitted as a text string: `g.V().addE('something').from(__.V().next()).to(__.V().next())`.

Properties on elements

Starting with Amazon Neptune engine release 1.5.0.0, you can use the `materializeProperties` flag, introduced in TinkerPop 3.7.0, to control whether the query returns properties on the elements (vertices and edges) in the result. For more information, see Properties of Elements on the Apache TinkerPop website.

You set the flag on a traversal using the `with()` configuration step:

- `g.with('materializeProperties', 'all')` returns elements with their properties included inline in the result.
- `g.with('materializeProperties', 'tokens')` returns elements as references only (that is, with just their `id` and `label`, and no properties).

When the flag is not set explicitly, the default behavior depends on the serializer that the request uses, not on the transport or the submission method:

- GraphSON requests (such as HTTP and script-based requests) default to returning properties, which is equivalent to `all`.

- GraphBinary requests (such as modern Gremlin Language Variant (GLV) clients like `gremlin-python` and `gremlin-java` over a WebSocket) default to returning references only, which is equivalent to `tokens`.

You can override either default by setting the `materializeProperties` flag explicitly with the `with()` step, as shown earlier.

Neptune does not accept `null`, `Map`, or `UUID` values as vertex or edge property values. `Set`-cardinality property values are supported (see Cardinality of vertex properties), but property values that are themselves sets — a single property whose value is a set of nested elements — are not supported.

Meta-properties

Neptune does not support meta-properties — that is, properties on properties. Attempting to attach a property to another property through TinkerPop's `VertexProperty.property(...)` API does not persist the nested value.

Ordering of vertices, edges, and properties

Neptune does not preserve insertion order for vertices, edges, or properties. Iteration order for `V()`, `E()`, and property enumeration is determined by an internal hash of the element's `(id, key, value)` tuple.

This behavior is not unique to Neptune. The Apache TinkerPop specification does not require providers to enforce insertion order for vertices, edges, or properties, so most graph databases behave the same way. You might encounter the expectation of preserved insertion order from in-memory graphs like `TinkerGraph` that preserve it.

If your application requires a deterministic iteration order, apply an explicit `order()` step with a stable `by()` modulator (for example, `order().by(id)` or `order().by('name')`) rather than relying on insertion order.

Datetime offsets

Neptune internally represents datetime values as instants — UTC-based points in time. When `asDate()` or a related datetime step is applied to a string that carries a time zone offset (for example, `'2024-01-15T10:30:00+05:00'`), the offset is normalized to UTC and the original offset information is not preserved in the round-trip result.

Sessions

Sessions in Neptune are limited to only 10 minutes in duration. See Gremlin script-based sessions and the TinkerPop Session Reference for more information.

Transactions

Neptune opens a new transaction at the beginning of each Gremlin traversal and closes the transaction upon the successful completion of the traversal. The transaction is rolled back when there is an error.

Multiple statements separated by a semicolon (`;`) or a newline character (\n) are included in a single transaction. Every statement other than the last must end with a `next()` step to be executed. Only the final traversal data is returned.

Manual transaction logic using `tx.commit()` and `tx.rollback()` is not supported.

> **Important**
>
> This ***only*** applies to methods where you send the Gremlin query as a ***text string*** (see Gremlin transactions).

Vertex and edge IDs

Neptune Gremlin Vertex and Edge IDs must be of type `String`. These ID strings support Unicode characters, and cannot exceed 55 MB in size.

User-supplied IDs are supported, but they are optional in normal usage. If you don't provide an ID when you add a vertex or an edge, Neptune generates a UUID and converts it to a string, in a form like this: `"48af8178-50ce-971a-fc41-8c9a954cea62"`. These UUIDs do not conform to the RFC standard, so if you need standard UUIDs you should generate them externally and provide them when you add vertices or edges.

> **Note**
>
> The Neptune Load command requires that you provide IDs, using the **~id** field in the Neptune CSV format.

User-supplied IDs

User-supplied IDs are allowed in Neptune Gremlin with the following stipulations.

- Supplied IDs are optional.
- Only vertexes and edges are supported.
- Only type `String` is supported.

To create a new vertex with a custom ID, use the `property` step with the `id` keyword: `g.addV().property(id, 'customid')`.

> **Note**
>
> Do not put quotation marks around the `id` keyword. It refers to `T.id`.

All vertex IDs must be unique, and all edge IDs must be unique. However, Neptune does allow a vertex and an edge to have the same ID.

If you try to create a new vertex using the `g.addV()` and a vertex with that ID already exists, the operation fails. The exception to this is if you specify a new label for the vertex, the operation succeeds but adds the new label and any additional properties specified to the existing vertex. Nothing is overwritten. A new vertex is not created. The vertex ID does not change and remains unique.

For example, the following Gremlin Console commands succeed:

```
gremlin> g.addV('label1').property(id, 'customid')
gremlin> g.addV('label2').property(id, 'customid')
gremlin> g.V('customid').label()
==>label1::label2
```

Vertex property IDs

Vertex property IDs are generated automatically and can show up as positive or negative numbers when queried.

Cardinality of vertex properties

Neptune supports set cardinality and single cardinality. If it isn't specified, set cardinality is selected. This means that if you set a property value, it adds a new value to the property, but only if it doesn't already appear in the set of values. This is the Gremlin enumeration value of Set.

`List` is not supported. For more information about property cardinality, see the Vertex topic in the Gremlin JavaDoc.

Updating a vertex property

To update a property value without adding an additional value to the set of values, specify `single` cardinality in the `property` step.

```
g.V('exampleid01').property(single, 'age', 25)
```

This removes all existing values for the property.

Changing a vertex property's cardinality from `single` to `set` through `property(set, ...)` does not remove the existing single-cardinality value. The single value remains in the property set alongside any new set values. To replace an existing single value, drop the property first with `properties(...).drop()` before applying the new set-cardinality value.

Labels

Neptune supports multiple labels for a vertex. When you create a label, you can specify multiple labels by separating them with `::`. For example, `g.addV("Label1::Label2::Label3")` adds a vertex with three different labels. The `hasLabel` step matches this vertex with any of those three labels: `hasLabel("Label1")`, `hasLabel("Label2")`, and `hasLabel("Label3")`.

> **Important**
>
> The `::` delimiter is reserved for this use only. You cannot specify multiple labels in the `hasLabel` step. For example, `hasLabel("Label1::Label2")` does not match anything.

Escape characters

Neptune resolves all escape characters as described in the Escaping Special Characters section of the Apache Groovy language documentation.

Serialization

Neptune supports the following serializations based on the requested MIME type.

With Neptune, you can use many of the serializers that TinkerPop offers, with support for the various versions and configurations of GraphSON and GraphBinary. See the following table for the currently supported serializers. Despite there being many options present, the guidance for which to use is straightforward:

- If you are using Apache TinkerPop drivers, prefer the default for the driver without specifying one explicitly. Unless you have a very specific reason for doing so, you likely don't need to specify the serializer in your driver initialization. In general, the default used by the drivers is `application/vnd.graphbinary-v1.0`.
- If you are connecting to Neptune over HTTP, prioritize the use of `application/vnd.gremlin-v3.0+json;types=false` as the embedded types in the alternative version of GraphSON 3 make it complicated to work with.
- The `application/vnd.graphbinary-v1.0-stringd` is generally only useful when used in conjunction with Gremlin Console as it converts all results to a string representation for simple display.
- The remaining formats remain present for legacy reasons and should typically not be used with drivers without clear cause.

MIME type	Serialization	Configuration
`application/vnd.gremlin-v1.0+json;types=false`	`GraphSONUntypedMessageSerializerV1`	`ioRegistries: [org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerIoRegistryV1]`
`application/vnd.gremlin-v2.0+json`	`GraphSONMessageSerializerV2`	`ioRegistries: [org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerIoRegistryV2]`

application/vnd.gremlin-v2.0+json;types=false	GraphSONUntypedMessageSerializerV2	ioRegistries: [org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerIoRegistryV2]
application/vnd.gremlin-v3.0+json	GraphSONMessageSerializerV3	ioRegistries: [org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerIoRegistryV3]
application/vnd.gremlin-v3.0+json;types=false	GraphSONUntypedMessageSerializerV3	ioRegistries: [org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerIoRegistryV3]
application/json	GraphSONUntypedMessageSerializerV3	ioRegistries: [org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerIoRegistryV1]
application/vnd.graphbinary-v1.0	GraphBinaryMessageSerializerV1	
application/vnd.graphbinary-v1.0-stringd	GraphBinaryMessageSerializerV1	serializeResultToString: true
application/vnd.gremlin-v1.0+json	GraphSONMessageSerializerGremlinV1	ioRegistries: [org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerIoRegistryV1]

application/vnd.gremlin-v2.0+json	GraphSONMessageSerializerV2 (only works with WebSockets)	ioRegistries: [org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerIoRegistryV2]
application/vnd.gremlin-v3.0+json	GraphSONMessageSerializerV3	
application/json	GraphSONMessageSerializerV3	ioRegistries: [org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerIoRegistryV3]
application/vnd.graphbinary-v1.0	GraphBinaryMessageSerializerV1	

> **Note**
>
> The serializer table shown here refers to naming as of TinkerPop 3.7.0. If you would like to know more about this change, please see the TinkerPop upgrade documentation. Gryo serialization support was deprecated in 3.4.3 and was officially removed in 3.6.0. If you are explicitly using Gryo or on a driver version that uses it by default, then you should switch to GraphBinary or upgrade your driver.

Lambda steps

Neptune does not support Lambda Steps.

Unsupported Gremlin steps

Neptune does not support the following Gremlin steps:

- The Gremlin io( ) Step is only partially supported in Neptune. You can use it in a read context, as in `g.io("https://example.com/data/my-graph.graphml").read()`, but you cannot use it to write. To read a file that you store as an Amazon S3 object, first generate a presigned

URL. Then pass that HTTPS URL to `g.io()`. For more information about presigned URLs, see Download and upload objects with presigned URLs in the *Amazon S3 User Guide*.

Gremlin graph features in Neptune

The Neptune implementation of Gremlin does not expose the `graph` object. The following tables list Gremlin features and indicate whether or not Neptune supports them.

Neptune support for `graph` features

The Neptune graph features, where supported, are the same as would be returned by the `graph.features()` command.

Graph feature	Enabled?
`Transactions`	true
`ThreadedTransactions`	false
`Computer`	false
`Persistence`	true
`ConcurrentAccess`	true

Neptune support for variable features

Variable feature	Enabled?
`Variables`	false
`SerializableValues`	false
`UniformListValues`	false
`BooleanArrayValues`	false
`DoubleArrayValues`	false
`IntegerArrayValues`	false

`StringArrayValues`	false
`BooleanValues`	false
`ByteValues`	false
`DoubleValues`	false
`FloatValues`	false
`IntegerValues`	false
`LongValues`	false
`MapValues`	false
`MixedListValues`	false
`StringValues`	false
`ByteArrayValues`	false
`FloatArrayValues`	false
`LongArrayValues`	false

Neptune support for vertex features

Vertex feature	Enabled?
`MetaProperties`	false
`DuplicateMultiProperties`	false
`AddVertices`	true
`RemoveVertices`	true
`MultiProperties`	true
`UserSuppliedIds`	true

`AddProperty`	true
`RemoveProperty`	true
`NumericIds`	false
`StringIds`	true
`UuidIds`	false
`CustomIds`	false
`AnyIds`	false

Neptune support for vertex property features

Vertex property feature	Enabled?
`UserSuppliedIds`	false
`AddProperty`	true
`RemoveProperty`	true
`NumericIds`	true
`StringIds`	true
`UuidIds`	false
`CustomIds`	false
`AnyIds`	false
`Properties`	true
`SerializableValues`	false
UniformListValues	false
`BooleanArrayValues`	false

DoubleArrayValues	false
IntegerArrayValues	false
StringArrayValues	false
BooleanValues	true
ByteValues	true
DoubleValues	true
FloatValues	true
IntegerValues	true
LongValues	true
MapValues	false
MixedListValues	false
StringValues	true
ByteArrayValues	false
FloatArrayValues	false
LongArrayValues	false

Neptune support for edge features

Edge feature	Enabled?
AddEdges	true
RemoveEdges	true
UserSuppliedIds	true
AddProperty	true

`RemoveProperty`	true
`NumericIds`	false
`StringIds`	true
`UuidIds`	false
`CustomIds`	false
`AnyIds`	false

Neptune support for edge property features

Edge property feature	Enabled?
`Properties`	true
`SerializableValues`	false
`UniformListValues`	false
`BooleanArrayValues`	false
`DoubleArrayValues`	false
`IntegerArrayValues`	false
`StringArrayValues`	false
`BooleanValues`	true
`ByteValues`	true
`DoubleValues`	true
`FloatValues`	true
`IntegerValues`	true
`LongValues`	true

`MapValues`	false
`MixedListValues`	false
`StringValues`	true
`ByteArrayValues`	false
`FloatArrayValues`	false
`LongArrayValues`	false

SPARQL standards compliance in Amazon Neptune

After listing applicable SPARQL standards, the following sections provide specific details about how Neptune's SPARQL implementation extends or diverges from those standards.

Topics

- Applicable Standards for SPARQL
- Default Namespace Prefixes in Neptune SPARQL
- SPARQL Default Graph and Named Graphs
- SPARQL XPath Constructor Functions Supported by Neptune
- Default base IRI for queries and updates
- xsd:dateTime Values in Neptune
- Neptune Handling of Special Floating Point Values
- Neptune Limitation of Arbitrary-Length Values
- Neptune Extends Equals Comparison in SPARQL
- Handling of Out-of-Range Literals in Neptune SPARQL

Amazon Neptune complies with the following standards in implementing the SPARQL graph query language.

Applicable Standards for SPARQL

- SPARQL is defined by the W3C SPARQL 1.1 Query Language recommendation of March 21, 2013.

- The SPARQL Update protocol and query language are defined by the W3C SPARQL 1.1 Update specification.
- For numeric formats, SPARQL follows the W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes specification, which is consistent with the IEEE 754 specification (IEEE 754-2019 - IEEE Standard for Floating-Point Arithmetic. For more information, see also the Wikipedia IEEE 754 page). However, features that were introduced after the `IEEE 754-1985` version are not included in the specification.

Default Namespace Prefixes in Neptune SPARQL

Neptune defines the following prefixes by default for use in SPARQL queries. For more information, see Prefixed Names in the SPARQL specification.

- `rdf` – `http://www.w3.org/1999/02/22-rdf-syntax-ns#`
- `rdfs` – `http://www.w3.org/2000/01/rdf-schema#`
- `owl` – `http://www.w3.org/2002/07/owl#`
- `xsd` – `http://www.w3.org/2001/XMLSchema#`

SPARQL Default Graph and Named Graphs

Amazon Neptune associates every triple with a named graph. The default graph is defined as the union of all named graphs.

Default Graph for Queries

If you submit a SPARQL query without explicitly specifying a graph via the `GRAPH` keyword or constructs such as `FROM NAMED`, Neptune always considers all triples in your DB instance. For example, the following query returns all triples from a Neptune SPARQL endpoint:

```
SELECT * WHERE { ?s ?p ?o }
```

Triples that appear in more than one graph are returned only once.

For information about the default graph specification, see the RDF Dataset section of the SPARQL 1.1 Query Language specification.

Specifying the Named Graph for Loading, Inserts, or Updates

If you don't specify a named graph when loading, inserting, or updating triples, Neptune uses the fallback named graph defined by the URI `http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph`.

When you issue a Neptune `Load` request using a triple-based format, you can specify the named graph to use for all triples by using the `parserConfiguration: namedGraphUri` parameter. For information about the `Load` command syntax, see the section called "Loader Command".

> **Important**
>
> If you don't use this parameter, and you don't specify a named graph, the fallback URI is used: `http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph`.

This fallback named graph is also used if you load triples via `SPARQL UPDATE` without explicitly providing a named graph target.

You can use the quads-based format N-Quads to specify a named graph for each triple in the database.

> **Note**
>
> Using N-Quads allows you to leave the named graph blank. In this case, `http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph` is used.
> You can override the default named graph for N-Quads using the `namedGraphUri` parser configuration option.

SPARQL XPath Constructor Functions Supported by Neptune

The SPARQL standard allows SPARQL engines to support an extensible set of XPath constructor functions. Neptune currently supports the following constructor functions, where the `xsd` prefix is defined as `http://www.w3.org/2001/XMLSchema#`:

- `xsd:boolean`
- `xsd:integer`
- `xsd:double`
- `xsd:float`

- `xsd:decimal`
- `xsd:long`
- `xsd:unsignedLong`

Default base IRI for queries and updates

Because a Neptune cluster has several different endpoints, using the request URL of a query or update as the base IRI could lead to unexpected results when resolving relative IRIs.

As of engine release 1.2.1.0, Neptune uses `http://aws.amazon.com/neptune/default/` as the base IRI if an explicit base IRI is not part of the request.

In the following request, the base IRI is part of the request:

```
BASE <http://example.org/default/>
INSERT DATA { <node1> <id> "n1" }

BASE <http://example.org/default/>
SELECT * { <node1> ?p ?o }
```

And the result would be:

```
?p                                                  ?o
http://example.org/default/id                       n1
```

In this request, however, no base IRI is included:

```
INSERT DATA { <node1> <id> "n1" }

SELECT * { <node1> ?p ?o }
```

In that case, the result would be:

```
?p                                                  ?o
http://aws.amazon.com/neptune/default/id            n1
```

xsd:dateTime Values in Neptune

For performance reasons, Neptune always stores date/time values as Coordinated Universal Time (UTC). This makes direct comparisons very efficient.

This also means that if you enter a `dateTime` value that specifies a particular time zone, Neptune translates the value to UTC and discards that time-zone information. Then, when you retrieve the `dateTime` value later, it is expressed in UTC, not the time of the original time zone, and you can no longer tell what that original time zone was.

Neptune Handling of Special Floating Point Values

Neptune handles special floating-point values in SPARQL as follows.

SPARQL NaN Handling in Neptune

In Neptune, SPARQL can accept a value of `NaN` in a query. No distinction is made between signaling and quiet `NaN` values. Neptune treats all `NaN` values as quiet.

Semantically, no comparison of a `NaN` is possible, because nothing is greater than, less than, or equal to a `NaN`. This means that a value of `NaN` on one side of a comparison in theory never matches *anything* on the other side.

However, the XSD specification does treat two `xsd:double` or `xsd:float` `NaN` values as equal. Neptune follows this for the `IN` filter, for the equal operator in filter expressions, and for exact match semantics (having a `NaN` in the object position of a triple pattern).

SPARQL Infinite Value Handling in Neptune

In Neptune, SPARQL can accept a value of `INF` or `-INF` in a query. `INF` compares as greater than any other numeric value, and `-INF` compares as less than any other numeric value.

Two INF values with matching signs compare as equal to each other regardless of their type (for example, a float `-INF` compares as equal to a double `-INF`).

Of course, no comparison with a `NaN` is possible because nothing is greater than, less than, or equal to a `NaN`.

SPARQL Negative Zero Handling in Neptune

Neptune normalizes a negative zero value to an unsigned zero. You can use negative zero values in a query, but they aren't recorded as such in the database, and they compare as equal to unsigned zeros.

Neptune Limitation of Arbitrary-Length Values

Neptune limits the storage size of XSD integer, floating point, and decimal values in SPARQL to 64 bits. Using larger values results in an `InvalidNumericDataException` error.

Neptune Extends Equals Comparison in SPARQL

The SPARQL standard defines a ternary logic for value expressions, where a value expression can either evaluate to `true`, `false`, or `error`. The default semantics for term equality as defined in the SPARQL 1.1 specification), which applies to = and `!=` comparisons in `FILTER` conditions, produces an `error` when comparing data types that are not explicitly comparable in the operators table in the specification.

This behavior can lead to unintuitive results, as in the following example.

Data:

```
<http://example.com/Server/1> <http://example.com/ip> "127.0.0.1"^^<http://example.com/
datatype/IPAddress>
```

Query 1:

```
SELECT * WHERE {
    <http://example.com/Server/1> <http://example.com/ip> ?o .
    FILTER(?o = "127.0.0.2"^^<http://example.com/datatype/IPAddress>)
}
```

Query 2:

```
SELECT * WHERE {
    <http://example.com/Server/1> <http://example.com/ip> ?o .
    FILTER(?o != "127.0.0.2"^^<http://example.com/datatype/IPAddress>)
}
```

With the default SPARQL semantics that Neptune used before release 1.0.2.1, both queries would return the empty result. The reason is that `?o = "127.0.0.2"^^<http://example.com/IPAddress>` when evaluated for `?o := "127.0.0.1"^^<http://example.com/IPAddress>` produces an `error` rather than `false` because there are no explicit comparison rules specified for the custom data type `<http://example.com/IPAddress>`. As a result, the negated version

in the second query also produces an `error`. In both queries, the `error` causes the candidate solution to be filtered out.

Starting with release 1.0.2.1, Neptune has extended the SPARQL inequality operator in accord with the specification. See the SPARQL 1.1 section on operator extensibility, which allows engines to define additional rules on how to compare across user-defined and non-comparable built-in data types.

Using this option, Neptune now treats a comparison of any two data types that is not explicitly defined in the operator-mapping table as evaluating to `true` if the literal values and data types are syntactically equal, and false otherwise. An `error` is not produced in any case.

Using these new semantics, the second query would return `"127.0.0.1"^^<http://example.com/IPAddress>` instead of an empty result.

Handling of Out-of-Range Literals in Neptune SPARQL

XSD semantics define each numeric type with its value space, except for `integer` and `decimal`. These definitions limit each type to a range of values. For example, the range of an `xsd:byte` range is from -128 to +127, inclusive. Any value outside of this range is considered invalid.

If you try to assign a literal value outside of the value space of a type (for example, if you try to set an `xsd:byte` to a literal value of 999), Neptune accepts the out-of-range value as-is, without rounding or truncating it. But it doesn't persist it as a numeric value because the given type can't represent it.

That is, Neptune accepts `"999"^^xsd:byte` even though it is a value outside of the defined `xsd:byte` value range. However, after the value is persisted in the database, it can only be used in exact match semantics, in an object position of a triple pattern. No range filter can be executed on it because out-of-range literals are not treated as numeric values.

The SPARQL 1.1 specification defines range operators in the form `numeric-`*operator*`-numeric`, `string-`*operator*`-string`, `literal-`*operator*`-literal`, and so forth. Neptune can't execute a range comparison operator anything like `invalid-literal-`*operator*`-numeric-value`.

openCypher specification compliance in Amazon Neptune

The Amazon Neptune release of openCypher generally supports the clauses, operators, expressions, functions, and syntax defined in the by the current openCypher specification, which is

the Cypher Query Language Reference Version 9. Limitations and differences in Neptune support for openCypher are called out below.

Amazon Neptune also supports several features beyond the scope of the openCypher specification. Refer to openCypher extensions in Amazon Neptune for details.

> **Note**
>
> The current Neo4j implementation of Cypher contains functionality that is not contained in the openCypher specification mentioned above. If you are migrating current Cypher code to Neptune, see Neptune compatibility with Neo4j and Rewriting Cypher queries to run in openCypher on Neptune for more information.

Support for openCypher clauses in Neptune

Neptune supports the following clauses, except as noted:

- `MATCH` – Supported, except that *`shortestPath()`* and *`allShortestPaths()`* are not currently supported.
- `OPTIONAL MATCH`
- *`MANDATORY MATCH`* – is **not** currently supported in Neptune. Neptune does, however, support custom ID values in `MATCH` queries.
- `RETURN` – Supported, except when used with non-static values for `SKIP` or `LIMIT`. For example, the following currently does not work:

  ```
  MATCH (n)
  RETURN n LIMIT toInteger(rand())    // Does NOT work!
  ```

- `WITH` – Supported, except when used with non-static values for `SKIP` or `LIMIT`. For example, the following currently does not work:

  ```
  MATCH (n)
  WITH n SKIP toInteger(rand())
  WITH count() AS count
  RETURN count > 0 AS nonEmpty    // Does NOT work!
  ```

- `UNWIND`

- `WHERE`
- `ORDER BY`
- `SKIP`
- `LIMIT`
- `CREATE` – Neptune lets you create custom ID values in `CREATE` queries.
- `DELETE`
- `SET`
- `REMOVE`
- `MERGE` – Neptune supports custom ID values in `MERGE` queries.
- *`CALL[YIELD...]`* – is **not** currently supported in Neptune.
- `UNION, UNION ALL` – read-only queries are supported, but mutation queries are **not** currently supported.
- `USING` – `USING` is supported from engine version 1.3.2.0. See Query hints for more information.

Support for openCypher operators in Neptune

Neptune supports the following operators, except as noted:

General operators

- `DISTINCT`
- The `.` operator for accessing properties of a nested literal map.

Mathematical operators

- The **+** addition operator.
- The **-** subtraction operator.
- The ***** multiplication operator.
- The **/** division operator.
- The **%** modulo division operator.
- The **^** exponentiation operator *`is NOT supported`*.

Comparison operators

- The **=** addition operator.
- The **<>** inequality operator.
- The **<** less-than operator is supported except when either of the arguments is a Path, List, or Map.
- The **>** greater-than operator is supported except when either of the arguments is a Path, List, or Map.
- The **<=** less-than-or-equal-to operator is supported except when either of the arguments is a Path, List, or Map.
- The **>=** greater-than-or-equal-to operator is supported except when either of the arguments is a Path, List, or Map.
- `IS NULL`
- `IS NOT NULL`
- `STARTS WITH` is supported if the data being searched for is a string.
- `ENDS WITH` is supported if the data being searched for is a string.
- `CONTAINS` is supported if the data being searched for is a string.

Boolean operators

- `AND`
- `OR`
- `XOR`
- `NOT`

String operators

- The **+** concatenation operator.

List operators

- The **+** concatenation operator.
- `IN` (checks for the presence of an item in the list)

Support for openCypher expressions in Neptune

Neptune supports the following expressions, except as noted:

- `CASE`
- The **`[]`** expression is is **not** currently supported in Neptune for accessing dynamically computed property keys within a node, relationship, or map. For example, the following does not work:

```
MATCH (n)
WITH [5, n, {key: 'value'}] AS list
RETURN list[1].name
```

Support for openCypher functions in Neptune

Neptune supports the following functions, except as noted:

Predicate functions

- `exists()`

Scalar functions

- `coalesce()`
- `endNode()`
- `epochmillis()`
- `head()`
- `id()`
- `last()`
- `length()`
- `randomUUID()`
- `properties()`
- `removeKeyFromMap`
- `size()` – this overloaded method currently only works for pattern expressions, lists, and strings

- `startNode()`
- `timestamp()`
- `toBoolean()`
- `toFloat()`
- `toInteger()`
- `type()`

Aggregating functions

- `avg()`
- `collect()`
- `count()`
- `max()`
- `min()`
- `percentileDisc()`
- `stDev()`
- `percentileCont()`
- `stDevP()`
- `sum()`

List functions

- `join()` (concatenates strings in a list into a single string)
- `keys()`
- `labels()`
- `nodes()`
- `range()`
- `relationships()`
- `reverse()`
- `tail()`

Mathematical functions – numeric

- `abs()`
- `ceil()`
- `floor()`
- `rand()`
- `round()`
- `sign()`

Mathematical functions – logarithmic

- `e()`
- `exp()`
- `log()`
- `log10()`
- `sqrt()`

Mathematical functions – trigonometric

- `acos()`
- `asin()`
- `atan()`
- `atan2()`
- `cos()`
- `cot()`
- `degrees()`
- `pi()`
- `radians()`
- `sin()`
- `tan()`

String functions

- `join()` (concatenates strings in a list into a single string)
- `left()`
- `lTrim()`
- `replace()`
- `reverse()`
- `right()`
- `rTrim()`
- `split()`
- `substring()`
- `toLower()`
- `toString()`
- `toUpper()`
- `trim()`

User-defined functions

`User-defined functions` are **not** currently supported in Neptune.

Neptune-specific openCypher implementation details

The following sections describe ways in which the Neptune implementation of openCypher may differ from or go beyond the openCypher spec.

Variable-length path (VLP) evaluations in Neptune

Variable length path (VLP) evaluations discover paths between nodes in the graph. Path length can be unrestricted in a query. To prevent cycles, the openCypher spec specifies that each edge must be traversed at most once per solution.

For VLPs, the Neptune implementation deviates from the openCypher spec in that it only supports constant values for property equality filters. Take the following query:

```
MATCH (x)-[:route*1..2 {dist:33, code:x.name}]->(y) return x,y
```

Because the `x.name` property equality filter value is a not a constant, this query results in an `UnsupportedOperationException` with the message: `Property predicate over variable-length relationships with non-constant expression is not supported in this release.`

Temporal support in the Neptune openCypher implementation (Neptune database 1.3.1.0 and below)

Neptune currently provides limited support for temporal function in openCypher. It supports the `DateTime` data type for temporal types.

The `datetime()` function can be used to get the current UTC date and time like this:

```
RETURN  datetime() as res
```

Date and time values can be parsed from strings in a "*date*T*time*" format where *date* and *time* are both expressed in one of the supported forms below:

Supported date formats

- `yyyy-MM-dd`
- `yyyyMMdd`
- `yyyy-MM`
- `yyyy-DDD`
- `yyyyDDD`
- `yyyy`

Supported time formats

- `HH:mm:ssZ`
- `HHmmssZ`
- `HH:mm:ssZ`
- `HH:mmZ`
- `HHmmZ`
- `HHZ`
- `HHmmss`

- `HH:mm:ss`
- `HH:mm`
- `HHmm`
- `HH`

For example:

```
RETURN datetime('2022-01-01T00:01')      // or another example:
RETURN datetime('2022T0001')
```

Note that all date/time values in Neptune openCypher are stored and retrieved as UTC values.

Neptune openCypher uses a `statement` clock, meaning that the same instant in time is used throughout the duration of a query. A different query within the same transaction may use a different instant in time.

Neptune doesn't support use of a function within a call to `datetime()`. For example, the following won't work:

```
CREATE (:n {date:datetime(tostring(2021))})  // ---> NOT ALLOWED!
```

Neptune does support the `epochmillis()` function that converts a `datetime` to `epochmillis`. For example:

```
MATCH (n) RETURN epochMillis(n.someDateTime)
1698972364782
```

Neptune doesn't currently support other functions and operations on `DateTime` objects, such as addition and subtraction.

Temporal support in the Neptune openCypher implementation (Neptune Analytics and Neptune Database 1.3.2.0 and above)

The following datetime functionality for OpenCypher applies to Neptune Analytics. Alternatively, you can use the labmode parameter `DatetimeMillisecond=enabled` for enabling the following datetime functionality on Neptune engine release version 1.3.2.0 and above. For more details about using this functionality in labmode, see Extended datetime support.

- Support for milliseconds. Datetime literal will always be returned with milliseconds, even if milliseconds is 0. (Previous behavior was to truncate milliseconds.)

```
CREATE (:event {time: datetime('2024-04-01T23:59:59Z')})

# Returning the date returns with 000 suffixed representing milliseconds
MATCH(n:event)
RETURN n.time as datetime

{
  "results" : [ {
    "n" : {
      "~id" : "0fe88f7f-a9d9-470a-bbf2-fd6dd5bf1a7d",
      "~entityType" : "node",
      "~labels" : [ "event" ],
      "~properties" : {
        "time" : "2024-04-01T23:59:59.000Z"
      }
    }
  } ]
}
```

- Support for calling the datetime() function over stored properties or intermediate results. For example, the following queries were not possible prior to this feature.

  Datetime() over properties:

```
// Create node with property 'time' stored as string
CREATE (:event {time: '2024-04-01T23:59:59Z'})

// Match and return this property as datetime
MATCH(n:event)
RETURN datetime(n.time) as datetime
```

  Datetime() over intermediate results:

```
// Parse datetime from parameter
UNWIND $list as myDate
RETURN datetime(myDate) as d
```

- It is now also possible to save datetime perperties that are created the in cases mentioned above.

Saving datetime from the string property of one property to another:

```
// Create node with property 'time' stored as string
CREATE (:event {time: '2024-04-01T23:59:59Z', name: 'crash'})

// Match and update the same property to datetime type
MATCH(n:event {name: 'crash'})
SET n.time = datetime(n.time)

// Match and update another node's property
MATCH(e:event {name: 'crash'})
MATCH(n:server {name: e.servername})
SET n.time = datetime(e.time)
```

Batch create nodes from a parameter with a datetime property:

```
// Batch create from parameter
UNWIND $list as events
CREATE (n:crash) {time: datetime(events.time)}
// Parameter value
{
  "x":[
    {"time":"2024-01-01T23:59:29", "name":"crash1"},
    {"time":"2023-01-01T00:00:00Z", "name":"crash2"}
  ]
}
```

- Support for a larger subset of ISO8601 datetime formats. See below.

Supported formats

The format of a datetime value is [Date]T[Time][Timezone], where T is the separator. If an explicit timezone is not provided, UTC (Z) is assumed to be the default.

Timezone

Supported timezone formats are:

- +/-HH:mm
- +/-HHmm

- +/-HH

The presence of a timezone in a datetime string is optional. In case the timezone offset is 0, Z can be used instead of the timezone postfix above to indicate UTC time. The supported range of a timezone is from -14:00 to +14:00.

Date

If no timezone is present, or the timezone is UTC (Z), the supported date formats are as follows:

> **Note**
>
> DDD refers to an ordinal date, which represents a day of the year from 001 to 365 (366 in leap years). For example, 2024-002 represents Jan 2, 2024.

- `yyyy-MM-dd`
- `yyyyMMdd`
- `yyyy-MM`
- `yyyyMM`
- `yyyy-DDD`
- `yyyyDDD`
- `yyyy`

If a timezone other than Z is chosen, the supported date formats are limited to the following:

- `yyyy-MM-dd`
- `yyyy-DDD`
- `yyyyDDD`

The supported range for dates is from 1400-01-01 to 9999-12-31.

Time

If no timezaone is present, or the timezone is UTC (Z), the supported time formats are:

- `HH:mm:ss.SSS`
- `HH:mm:ss`
- `HHmmss.SSS`
- `HHmmss`
- `HH:mm`
- `HHmm`
- `HH`

If a timezone other than Z is chosen, the supported time formats are limited to the following:

- `HH:mm:ss`
- `HH:mm:ss.SSS`

Differences in Neptune openCypher language semantics

Neptune represents node and relationship IDs as strings rather than integers. The ID equals the ID supplied via the data loader. If there is a namespace for the column, the namespace plus the ID. Consequently, the `id` function returns a string instead of an integer.

The `INTEGER` datatype is limited to 64 bits. When converting larger floating point or string values to an integer using the `TOINTEGER` function, negative values are truncated to `LLONG_MIN` and positive values are truncated to `LLONG_MAX`.

For example:

```
RETURN TOINTEGER(2^100)
>  9223372036854775807

RETURN TOINTEGER(-1 * 2^100)
>  -9223372036854775808
```

Multi-valued properties

Although openCypher CREATE does not create multi-valued properties, they can exist in data created using Gremlin (Neptune Database) or when loading data (Neptune Database and Neptune Analytics). If Neptune openCypher encounters a multi-value property, one of the values is arbitrarily chosen, creating a non-deterministic result.

Handling of NaN valuse

Neptune's handling of the comparison of `NaN` property values is undefined. Relying on such comparisons may result in unexpected or non-deterministic results.

Neptune Graph Data Model

The basic unit of Amazon Neptune graph data is a four-position (quad) element, which is similar to a Resource Description Framework (RDF) quad. The following are the four positions of a Neptune quad:

- `subject  (S)`
- `predicate  (P)`
- `object  (O)`
- `graph  (G)`

Each quad is a statement that makes an assertion about one or more resources. A statement can assert the existence of a relationship between two resources, or it can attach a property (key-value pair) to a resource. You can think of the quad predicate value generally as the verb of the statement. It describes the type of relationship or property that's being defined. The object is the target of the relationship, or the value of the property. The following are examples:

- A relationship between two vertices can be represented by storing the source vertex identifier in the `S` position, the target vertex identifier in the `O` position, and the edge label in the `P` position.
- A property can be represented by storing the element identifier in the `S` position, the property key in the `P` position, and the property value in the `O` position.

The graph position `G` is used differently in the different stacks. For RDF data in Neptune, the `G` position contains a named graph identifier. For property graphs in Gremlin, it is used to store the edge ID value in the case of an edge. In all other cases, it defaults to a fixed value.

A set of quad statements with shared resource identifiers creates a graph.

Dictionary of user-facing values

Neptune does not store most user-facing values directly in the various indexes it maintains. Instead, it stores them separately in a dictionary and replaces them in the indexes with 8-byte identifiers.

- All user-facing values that would go in `S`, `P`, or `G` indexes are stored in the dictionary in this way.
- In the `O` index, numeric values are stored directly in the index (inlined). This includes `date` and `datetime` values (represented as milliseconds from the epoch).

- All other user-facing values that would go in the 0 index are stored in the dictionary and represented in the index by IDs.

The dictionary contains a forward mapping of user-facing values to 8-byte IDs in a `value_to_id` index.

It stores the reverse mapping of 8-byte IDs to values in one of two indexes, depending on the size of the values:

- An `id_to_value` index maps IDs to user-facing values that are smaller than 767 bytes after internal encoding.
- An `id_to_blob` index maps IDs to user-facing values that are larger.

How Statements Are Indexed in Neptune

When you query a graph of quads, for each quad position, you can either specify a value constraint, or not. The query returns all the quads that match the value constraints that you specified.

Neptune uses indexes to resolve graph queries patterns. These indexes are over the four primary components of a graph edge: Subject (source vertex in LPG); Predicate (RDF), or Property or Edge Label (LPG); Object (target vertex or property value in LPG); and Graph (RDF) or Edge Identifier (LPG). There are 16 (2^4) possible access patterns for these four quad component positions. You can query all 16 patterns efficiently without having to scan and filter by using six indexes. Each quad statement index uses a key that is composed of the four position values concatenated in a different order. One possible combination of quad statement indexes that would cover all 16 access paths is:

```
     Access Pattern                                        Index key order
-----------------------------------------------------    ---------------
 1.  ????  (No constraints; returns every quad)              SPOG
 2.  SPOG  (Every position is constrained)                   SPOG
 3.  SPO?  (S, P, and O are constrained; G is not)           SPOG
 4.  SP??  (S and P are constrained; O and G are not)        SPOG
 5.  S???  (S is constrained; P, O, and G are not)           SPOG
 6.  S??G  (S and G are constrained; P and O are not)        SPOG

 7.  ?POG  (P, O, and G are constrained; S is not)           POGS
 8.  ?PO?  (P and O are constrained; S and G are not)        POGS
```

```
 9.  ?P??  (P is constrained; S, O, and G are not)          POGS

10.  ?P?G  (P and G are constrained; S and O are not)       GPSO
11.  SP?G  (S, P, and G are constrained; O is not)          GPSO
12.  ???G  (G is constrained; S, P, and O are not)          GPSO

13.  S?OG  (S, O, and G are constrained; P is not)          OGSP
14.  ??OG  (O and G are constrained; S and P are not)       OGSP
15.  ??O?  (O is constrained; S, P, and G are not)          OGSP

16.  S?O?  (S and O are constrained; P and G are not)       OSGP
```

Neptune creates and maintains only three out of those six indexes by default:

- `SPOG` – Uses a key composed of `Subject + Predicate + Object + Graph`.
- `POGS` – Uses a key composed of `Predicate + Object + Graph + Subject`.
- `GPSO` – Uses a key composed of `Graph + Predicate + Subject + Object`.

These three indexes handle many of the most common access patterns. Maintaining only three full statement indexes instead of six greatly reduces the resources that you need to support rapid access without scanning and filtering. For example, the `SPOG` index allows efficient lookup whenever a prefix of the positions, such as the vertex or vertex and property identifier, is bound. The `POGS` index allows efficient access when only the edge or property label stored in `P` position is bound.

The low-level API for finding statements takes a statement pattern in which some positions are known and the rest are left for discovery by index search. By composing the known positions into a key prefix according to the index key order for one of the statement indexes, Neptune performs a range scan to retrieve all the statements matching the known positions.

However, one of the statement indexes that Neptune does *not* create by default is a reverse traversal `OSGP` index, which can gather predicates across objects and subjects. Instead, Neptune by default tracks distinct predicates in a separate index that it uses to do a union scan of `{all P x POGS}`. When you are working with Gremlin, a predicate corresponds to a property or an edge label.

If the number of distinct predicates in a graph becomes large, the default Neptune access strategy can become inefficient. In Gremlin, for example, an `in()` step where no edge labels are given, or any step that uses `in()` internally such as `both()` or `drop()`, may become quite inefficient.

Enabling OSGP Index Creation Using Lab Mode

If your data model creates a large number of distinct predicates, you may experience reduced performance and higher operational costs that can be dramatically improved by using Lab Mode to enable the OSGP index in addition to the three indexes that Neptune maintains by default.

Enabling the OSGP index can have a few down-sides:

- The insert rate may slow by up to 23%.
- Storage increases by up to 20%.
- Read queries that touch all indexes equally (which is quite rare) may have increased latencies.

In general, however, it is worth enabling the OSGP index for DB Clusters with a large number of distinct predicates. Object-based searches become highly efficient (for example, finding all incoming edges to a vertex, or all subjects connected to a given object), and as a result dropping vertices becomes much more efficient too.

> **Important**
>
> You can only enable the OSGP index in an empty DB cluster, before you load any data into it.

Gremlin statements in the Neptune data model

Gremlin property-graph data is expressed in the SPOG model using three classes of statements, namely:

- Vertex Label Statements
- Edge Statements
- Property Statements

For an explanation of how these are used in Gremlin queries, see Understanding how Gremlin queries work in Neptune.

Graph schema and metadata

Neptune provides several ways to discover and inspect the structure of your graph data without scanning the entire database. Use these features to understand your graph's topology, optimize query planning, and provide schema context to applications.

- **Getting a quick summary report about your graph** – Returns high-level statistics about your graph, including node and edge counts and the list of distinct labels and predicates.
- **Property graph schema** – Retrieves the complete schema of your property graph, including node labels, edge labels, property data types, and label triples that describe how node types connect through edge types. Designed for LLM query generation, graph visualization, and schema discovery.
- **Managing statistics for the Neptune DFE to use** – Provides statistics used by the DFE query optimizer for query plan selection, such as cardinality estimates and predicate selectivity.

Topics

- Getting a quick summary report about your graph
- Property graph schema
- Managing statistics for the Neptune DFE to use

Getting a quick summary report about your graph

The Neptune graph summary API retrieves the following information about your graph:

- For property (PG) graphs, the graph summary API returns a read-only list of node and edge labels and property keys, along with counts of nodes, edges, and properties.
- For resource description framework (RDF) graphs, the graph summary API returns a read-only list of classes and predicate keys, along with counts of quads, subjects, and predicates.

> **Note**
>
> The graph summary API was introduced in Neptune engine release 1.2.1.0.

With the graph summary API, you can quickly gain a high-level understanding of your graph data size and content. You can also use the API interactively within a Neptune notebook using the `%summary` Neptune Workbench magic. In a graph application, the API can be used to improve search results by providing discovered node or edge labels as part of the search.

Graph summary data is drawn from the DFE statistics computed by the Neptune DFE engine during runtime, and is available whenever DFE statistics are available. Statistics are enabled by default when you create a new Neptune DB cluster.

> **Note**
>
> Statistics generation is disabled on `t3` and `t4` instance types (that is, on `db.t3.medium` and `db.t4g.medium` instance types) to conserve memory. As a result, graph summary data is not available either on those instance types.

You can check the status of DFE statistics using the statistics status API. As long as auto-generation of statistics has not been disabled, statistics are automatically updated periodically.

If you want to be sure that statistics are as up to date as possible when you request a graph summary, you can manually trigger a statistics update right before retrieving the summary. If the graph is changing while the statistics are being computed, they will necessarily lag slightly behind, but not by much.

Using the graph summary API to retrieve graph summary information

For a property graph that you query using Gremlin or openCypher, you can retrieve a graph summary from the property-graph summary endpoint. There is both a long and a short URI for this endpoint:

- `https://`*`your-neptune-host`*`:`*`port`*`/propertygraph/statistics/summary`
- `https://`*`your-neptune-host`*`:`*`port`*`/pg/statistics/summary`

For an RDF graph that you query using SPARQL, you can retrieve a graph summary from the RDF summary endpoint:

- `https://`*`your-neptune-host`*`:`*`port`*`/rdf/statistics/summary`

These endpoints are read-only, and only support an HTTP `GET` operation. If $GRAPH_SUMMARY_ENDPOINT is set to the address of whichever endpoint you want to query, you can retrieve the summary data using `curl` and HTTP `GET` as follows:

```
curl -G "$GRAPH_SUMMARY_ENDPOINT"
```

If no statistics are available when you try to retrieve a graph summary, the response looks like this:

```
{
  "detailedMessage": "Statistics are not available. Summary can only be generated after
 statistics are available.",
  "requestId": "48c1f788-f80b-b69c-d728-3f6df579a5f6",
  "code": "StatisticsNotAvailableException"
}
```

The `mode` URL query parameter for the graph summary API

The graph summary API accepts a URL query parameter named `mode`, which can take one of two values, namely `basic` (the default) and `detailed`. For an RDF graph, the `detailed` mode graph summary response contains an additional `subjectStructures` field. For a property graph, the detailed graph summary response contains two additional fields, namely `nodeStructures` and `edgeStructures`.

To request a `detailed` graph summary response, include the `mode` parameter as follows:

```
curl -G "$GRAPH_SUMMARY_ENDPOINT?mode=detailed"
```

If the `mode` parameter isn't present, `basic` mode is used by default, so while it is possible to specify `?mode=basic` explicitly, this is not necessary.

Graph summary response for a property graph (PG)

For an empty property graph, the detailed graph summary response looks like this:

```
{
  "status" : "200 OK",
  "payload" : {
    "version" : "v1",
    "lastStatisticsComputationTime" : "2023-01-10T07:58:47.972Z",
    "graphSummary" : {
      "numNodes" : 0,
```

```
      "numEdges" : 0,
      "numNodeLabels" : 0,
      "numEdgeLabels" : 0,
      "nodeLabels" : [ ],
      "edgeLabels" : [ ],
      "numNodeProperties" : 0,
      "numEdgeProperties" : 0,
      "nodeProperties" : [ ],
      "edgeProperties" : [ ],
      "totalNodePropertyValues" : 0,
      "totalEdgePropertyValues" : 0,
      "nodeStructures" : [ ],
      "edgeStructures" : [ ]
    }
  }
}
```

A property graph (PG) summary response has the following fields:

- **`status`** – the HTTP return code of the request. If the request succeeded, the code is 200.

  See Common graph summary errors for a list of common errors.
- **`payload`**
  - **`version`** – The version of this graph summary response.
  - **`lastStatisticsComputationTime`** – The timestamp, in ISO 8601 format, of the time at which Neptune last computed statistics.
  - **`graphSummary`**
    - **`numNodes`** – The number of nodes in the graph.
    - **`numEdges`** – The number of edges in the graph.
    - **`numNodeLabels`** – The number of distinct node labels in the graph.
    - **`numEdgeLabels`** – The number of distinct edge labels in the graph.
    - **`nodeLabels`** – List of distinct node labels in the graph.
    - **`edgeLabels`** – List of distinct edge labels in the graph.
    - **`numNodeProperties`** – The number of distinct node properties in the graph.
    - **`numEdgeProperties`** – The number of distinct edge properties in the graph.
    - **`nodeProperties`** – List of distinct node properties in the graph, along with the count of nodes where each property is used.

- **`edgeProperties`** – List of distinct edge properties in the graph along with the count of edges where each property is used.
- **`totalNodePropertyValues`** – Total number of usages of all node properties.
- **`totalEdgePropertyValues`** – Total number of usages of all edge properties.
- **`nodeStructures`** – *This field is only present when `mode=detailed` is specified in the request.* It contains a list of node structures, each of which contains the following fields:
  - **`count`** – Number of nodes that have this specific structure.
  - **`nodeProperties`** – List of node properties present in this specific structure.
  - **`distinctOutgoingEdgeLabels`** – List of distinct outgoing edge labels present in this specific structure.
- **`edgeStructures`** – *This field is only present when `mode=detailed` is specified in the request.* It contains a list of edge structures, each of which contains the following fields:
  - **`count`** – Number of edges that have this specific structure.
  - **`edgeProperties`** – List of edge properties present in this specific structure.

Graph summary response for an RDF graph

For an empty RDF graph, the detailed graph summary response looks like this:

```
{
  "status" : "200 OK",
  "payload" : {
    "version" : "v1",
    "lastStatisticsComputationTime" : "2023-01-10T07:58:47.972Z",
    "graphSummary" : {
      "numDistinctSubjects" : 0,
      "numDistinctPredicates" : 0,
      "numQuads" : 0,
      "numClasses" : 0,
      "classes" : [ ],
      "predicates" : [ ],
      "subjectStructures" : [ ]
    }
  }
}
```

An RDF graph summary response has the following fields:

- **`status`** – the HTTP return code of the request. If the request succeeded, the code is 200.

  See Common graph summary errors for a list of common errors.
- **`payload`**
  - **`version`** – The version of this graph summary response.
  - **`lastStatisticsComputationTime`** – The timestamp, in ISO 8601 format, of the time at which Neptune last computed statistics.
  - **`graphSummary`**
    - **`numDistinctSubjects`** – The number of distinct subjects in the graph.
    - **`numDistinctPredicates`** – The number of distinct predicates in the graph.
    - **`numQuads`** – The number of quads in the graph.
    - **`numClasses`** – The number of classes in the graph.
    - **`classes`** – List of classes in the graph.
    - **`predicates`** – List of predicates in the graph, along with the predicate counts.
    - **`subjectStructures`** – *This field is only present when `mode=detailed` is specified in the request.* It contains a list of subject structures, each of which contains the following fields:
      - **`count`** – Number of occurrences of this specific structure.
      - **`predicates`** – List of predicates present in this specific structure.

Sample property-graph (PG) summary response

Here is the detailed summary response for a property graph that contains the sample property-graph air routes dataset:

```
{
  "status" : "200 OK",
  "payload" : {
    "version" : "v1",
    "lastStatisticsComputationTime" : "2023-03-01T14:35:03.804Z",
    "graphSummary" : {
      "numNodes" : 3748,
      "numEdges" : 51300,
      "numNodeLabels" : 4,
      "numEdgeLabels" : 2,
      "nodeLabels" : [
        "continent",
```

```
        "country",
        "version",
        "airport"
      ],
      "edgeLabels" : [
        "contains",
        "route"
      ],
      "numNodeProperties" : 14,
      "numEdgeProperties" : 1,
      "nodeProperties" : [
        {
          "desc" : 3748
        },
        {
          "code" : 3748
        },
        {
          "type" : 3748
        },
        {
          "country" : 3503
        },
        {
          "longest" : 3503
        },
        {
          "city" : 3503
        },
        {
          "lon" : 3503
        },
        {
          "elev" : 3503
        },
        {
          "icao" : 3503
        },
        {
          "region" : 3503
        },
        {
          "runways" : 3503
        },
```

```
        {
          "lat" : 3503
        },
        {
          "date" : 1
        },
        {
          "author" : 1
        }
      ],
      "edgeProperties" : [
        {
          "dist" : 50532
        }
      ],
      "totalNodePropertyValues" : 42773,
      "totalEdgePropertyValues" : 50532,
      "nodeStructures" : [
        {
          "count" : 3471,
          "nodeProperties" : [
            "city",
            "code",
            "country",
            "desc",
            "elev",
            "icao",
            "lat",
            "lon",
            "longest",
            "region",
            "runways",
            "type"
          ],
          "distinctOutgoingEdgeLabels" : [
            "route"
          ]
        },
        {
          "count" : 161,
          "nodeProperties" : [
            "code",
            "desc",
            "type"
```

```
          ],
          "distinctOutgoingEdgeLabels" : [
            "contains"
          ]
        },
        {
          "count" : 83,
          "nodeProperties" : [
            "code",
            "desc",
            "type"
          ],
          "distinctOutgoingEdgeLabels" : [ ]
        },
        {
          "count" : 32,
          "nodeProperties" : [
            "city",
            "code",
            "country",
            "desc",
            "elev",
            "icao",
            "lat",
            "lon",
            "longest",
            "region",
            "runways",
            "type"
          ],
          "distinctOutgoingEdgeLabels" : [ ]
        },
        {
          "count" : 1,
          "nodeProperties" : [
            "author",
            "code",
            "date",
            "desc",
            "type"
          ],
          "distinctOutgoingEdgeLabels" : [ ]
        }
      ],
```

```
      "edgeStructures" : [
        {
          "count" : 50532,
          "edgeProperties" : [
            "dist"
          ]
        }
      ]
    }
  }
}
```

Sample RDF graph summary response

Here is the detailed summary response for an RDF graph that contains the sample RDF air routes dataset:

```
{
  "status" : "200 OK",
  "payload" : {
    "version" : "v1",
    "lastStatisticsComputationTime" : "2023-03-01T14:54:13.903Z",
    "graphSummary" : {
      "numDistinctSubjects" : 54403,
      "numDistinctPredicates" : 19,
      "numQuads" : 158571,
      "numClasses" : 4,
      "classes" : [
        "http://kelvinlawrence.net/air-routes/class/Version",
        "http://kelvinlawrence.net/air-routes/class/Airport",
        "http://kelvinlawrence.net/air-routes/class/Continent",
        "http://kelvinlawrence.net/air-routes/class/Country"
      ],
      "predicates" : [
        {
          "http://kelvinlawrence.net/air-routes/objectProperty/route" : 50656
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/dist" : 50656
        },
        {
          "http://kelvinlawrence.net/air-routes/objectProperty/contains" : 7004
        },
```

```
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/code" : 3747
        },
        {
          "http://www.w3.org/2000/01/rdf-schema#label" : 3747
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/type" : 3747
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/desc" : 3747
        },
        {
          "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" : 3747
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/icao" : 3502
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/lat" : 3502
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/region" : 3502
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/runways" : 3502
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/longest" : 3502
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/elev" : 3502
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/lon" : 3502
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/country" : 3502
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/city" : 3502
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/author" : 1
```

```
        },
        {
          "http://kelvinlawrence.net/air-routes/datatypeProperty/date" : 1
        }
      ],
      "subjectStructures" : [
        {
          "count" : 50656,
          "predicates" : [
            "http://kelvinlawrence.net/air-routes/datatypeProperty/dist"
          ]
        },
        {
          "count" : 3471,
          "predicates" : [
            "http://kelvinlawrence.net/air-routes/datatypeProperty/city",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/code",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/country",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/desc",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/elev",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/icao",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/lat",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/lon",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/longest",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/region",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/runways",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/type",
            "http://kelvinlawrence.net/air-routes/objectProperty/route",
            "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
            "http://www.w3.org/2000/01/rdf-schema#label"
          ]
        },
        {
          "count" : 238,
          "predicates" : [
            "http://kelvinlawrence.net/air-routes/datatypeProperty/code",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/desc",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/type",
            "http://kelvinlawrence.net/air-routes/objectProperty/contains",
            "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
            "http://www.w3.org/2000/01/rdf-schema#label"
          ]
        },
        {
```

```
          "count" : 31,
          "predicates" : [
            "http://kelvinlawrence.net/air-routes/datatypeProperty/city",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/code",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/country",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/desc",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/elev",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/icao",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/lat",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/lon",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/longest",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/region",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/runways",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/type",
            "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
            "http://www.w3.org/2000/01/rdf-schema#label"
          ]
        },
        {
          "count" : 6,
          "predicates" : [
            "http://kelvinlawrence.net/air-routes/datatypeProperty/code",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/desc",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/type",
            "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
            "http://www.w3.org/2000/01/rdf-schema#label"
          ]
        },
        {
          "count" : 1,
          "predicates" : [
            "http://kelvinlawrence.net/air-routes/datatypeProperty/author",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/code",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/date",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/desc",
            "http://kelvinlawrence.net/air-routes/datatypeProperty/type",
            "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
            "http://www.w3.org/2000/01/rdf-schema#label"
          ]
        }
      ]
    }
  }
```

```
}
```

Using AWS Identity and Access Management (IAM) authentication with graph summary endpoints

You can access graph summary endpoints securely with IAM authentication by using awscurl or any other tool that works with HTTPS and IAM. See Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled to see how to set up the proper credentials. Once you have done that, you can then make requests like this:

```
awscurl "$GRAPH_SUMMARY_ENDPOINT" \
    --region (your region) \
    --service neptune-db
```

> **Important**
>
> The IAM identity or role that creates the temporary credentials must have an IAM policy attached that allows the GetGraphSummary IAM action.

See IAM Authentication Errors for a list of common IAM errors that you may encounter.

Common error codes that a graph summary request may return

Neptune service error code	HTT stat	Message	Error Scenario	Mitigation
`AccessDeniedException`	403	Missing Authentication Token.	Unsigned or incorrectly signed request was sent to Neptune database with IAM enabled.	Sign the request with SigV4 before sending (see IAM and graph summaries).
	403	User: *(user ARN)* is not authorized to perform: neptune-db:GetGraphSummary	IAM policy does not allow the action GetGraphSummary when the graph summary	Make sure that the IAM policy attached to the user or role making the request allows

Neptune service error code	HTT stat	Message	Error Scenario	Mitigation
		on resource: *(resource ARN)*.	request was sent to Neptune database with IAM enabled.	the `GetGraphSummary` action.
`BadRequestException`	400	Statistics are disabled, so graph summary is also disabled.	Trying to fetch summary on burstable instance types (`t3` or `t4g`) where statistics are disabled.	Use an instance type where statistics generation is enabled (all supported instances except `t3` and `t4g`).
	400	Bad route: */rdf/statistics/summarypathapi*	Request sent to invalid path.	Use correct route for graph summary endpoint.
`InvalidParameterException`	400	Request contains unknown parameters: '*(unknown parameter or parameters)* '.	When an invalid parameter is specified in the request.	Only use valid parameters (such as `mode`) in the request.
`InvalidParameterException`	400	URI query parameter 'mode' has unsupported value '*(invalid value)*'.	When the URL parameter 'mode' in the request is followed by an invalid value.	Use valid values (such as `basic` or `detailed`) when specifying the URL parameter 'mode'.
`MethodNotAllowedException`	405	Method Not Allowed.	Calling summary endpoint with any HTTP method other than `GET` (such as `POST` or `DELETE`).	Use HTTP `GET` method when calling summary endpoint.

Neptune service error code	HTT stat	Message	Error Scenario	Mitigation
`StatisticsNotAvailableException`	400	Statistics are not computed yet, graph summary will be available after statistics computation is complete.	There are no statistics available when the request is sent to the summary endpoint.	Wait until statistics generation is complete. You can check the status of statistics generation using the statistics status API.
	400	Statistics limit reached, thus graph summary is not available.	Statistics generation has stopped because it reached statistics size limits.	Graph summary is not available on this graph.

For example, if you make a request to graph summary endpoint in a Neptune database that has IAM authentication enabled, and the necessary permissions are not present in the requestor's IAM policy, then you would get a response like the following:

```
{
  "detailedMessage": "User: arn:aws:iam::(account ID):(user or user name) is not
 authorized to perform: neptune-db:GetGraphSummary on resource: arn:aws:neptune-
db:(region):(account ID):(cluster resource ID)/*",
  "requestId": "7ac2b98e-b626-d239-1d05-74b4c88fce82",
  "code": "AccessDeniedException"
}
```

Property graph schema

The `neptune.graph.pg_schema()` procedure provides a comprehensive overview of your property graph structure. It returns all node labels, edge labels, properties with their data types, and label triples (`{~from, ~type, ~to}` patterns that describe how node types connect through edge types).

This procedure is currently available only through the openCypher endpoint and discovers the schema for all property graph data.

Use this procedure for tasks such as:

- **AI and LLM query generation** – Give LLMs the graph structure they need to generate valid Cypher queries from natural language (Text-to-Cypher, GraphRAG applications).
- **Graph visualization and exploration** – Tools like Graph Explorer use schema information to render interactive visual representations of graph data without scanning the entire database.
- **Application schema discovery** – Applications that need to understand graph structure at startup, such as GraphQL schema generators or data validation tools.

Comparison with Neptune Analytics

In Neptune Analytics, `neptune.graph.pg_schema()` is synchronous. It computes the schema on every call.

In Neptune Database, you explicitly trigger an asynchronous schema computation by calling `neptune.graph.pg_schema.compute()`, which returns immediately. The computation runs in the background while you poll for completion using `neptune.graph.pg_schema()`. Once computed, Neptune persists the schema and returns it instantly on subsequent reads without recomputation. Partial results are also available while the computation is still in progress. You can also stop a running computation and resume it later.

Comparison with the Graph Summary API

The Graph Summary API does not provide label triples or property data types. The property graph schema procedure fills this gap. Label triples show the specific relationship patterns in your graph. For example, a `Person` connects to a `Company` via a `worksAt` edge. This information is critical for LLMs to generate semantically correct queries.

Prerequisites

Engine version

The property graph schema procedure requires Neptune engine version 1.4.8.0 or later.

IAM permissions

The following IAM actions are required for each schema operation:

- `CALL neptune.graph.pg_schema()` – requires `neptune-db:ReadDataViaQuery`.

- `CALL neptune.graph.pg_schema.compute()` – requires `neptune-db:ReadDataViaQuery` and `neptune-db:WriteDataViaQuery`.
- `CALL neptune.graph.pg_schema.stop()` – requires `neptune-db:ReadDataViaQuery` and `neptune-db:WriteDataViaQuery`.

The `compute()` and `stop()` operations require write permissions because they modify internal state used to cache and persist the schema.

Example IAM policy

The following policy grants the minimum permissions needed for all schema operations:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": [
      "neptune-db:ReadDataViaQuery",
      "neptune-db:WriteDataViaQuery"
    ],
    "Resource": "arn:aws:neptune-db:us-east-1:123456789012:cluster-resource-id/*"
  }]
}
```

To grant read-only access to the schema (without the ability to trigger computation), use only `neptune-db:ReadDataViaQuery`.

Writer and reader instances

You can trigger schema computation only on the writer instance. Read replica instances can read the schema (which is replicated from the writer) but cannot run `compute()` or `stop()`.

API reference

Read schema

Retrieves the current schema and computation status.

Syntax:

AWS CLI

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "CALL neptune.graph.pg_schema()"
```

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_query(
    openCypherQuery='CALL neptune.graph.pg_schema()'
)

print(response)
```

awscurl

```
awscurl -X POST https://your-neptune-endpoint:port/openCypher \
  -H "Content-Type: application/x-www-form-urlencoded" \
  --region us-east-1 --service neptune-db \
  -d 'query=CALL neptune.graph.pg_schema()'
```

Behavior: Returns immediately with the current schema and status. Always non-blocking. If no schema has been computed, returns state: `"NotStarted"` with empty schema fields. If a computation is in progress, returns partial results with state: `"InProgress"`.

Response format:

The response contains a schema object with the following fields:

Status object:

- `state` (String) – Current lifecycle state: `NotStarted`, `InProgress`, `Completed`, `Stopped`, `Failed`. Beginning with Neptune engine version 1.5.0.0, an interrupted computation can also report `Paused`. For more information, see States.
- `concurrency` (String) – Number of threads used for the computation. 0 means auto (determined based on hardware). Range: 1 (lowest) to 16 (highest).
- `lastComputedTimestamp` (String) – ISO-8601 UTC timestamp of the last successful computation (e.g., `2026-05-29T08:00:00Z`)
- `progressPercentage` (String) – Computation progress: 0 when not started, 0–99 during computation, 100 when completed
- `errorMessage` (String) – The reason a request was rejected or the computation is `Paused` or `Failed`. For example, a restart or failover interrupted the computation, or the computation failed.

Schema object:

- `nodeLabels` – Array of all unique node labels in the graph
- `edgeLabels` – Array of all unique edge labels in the graph
- `nodeLabelDetails` – For each node label: properties and their data types
- `edgeLabelDetails` – For each edge label: properties and their data types
- `labelTriples` – Array of relationship patterns: `{~from, ~type, ~to}` describing which node types connect through which edge types

Supported data types: `String`, `Int`, `Long`, `Double`, `Bool`, `Date`

If a property has multiple data types across different nodes (for example, some nodes store `age` as `Int` and others as `String`), all observed types are listed in the `datatypes` array.

Compute schema

Triggers a background schema computation.

Syntax:

AWS CLI

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
```

```
  --open-cypher-query "CALL neptune.graph.pg_schema.compute()"
```

With optional concurrency parameter:

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "CALL neptune.graph.pg_schema.compute({concurrency: 2})"
```

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_query(
    openCypherQuery='CALL neptune.graph.pg_schema.compute()'
)

print(response)
```

awscurl

```
awscurl -X POST https://your-neptune-endpoint:port/openCypher \
  -H "Content-Type: application/x-www-form-urlencoded" \
  --region us-east-1 --service neptune-db \
  -d 'query=CALL neptune.graph.pg_schema.compute()'
```

With optional concurrency parameter:

```
awscurl -X POST https://your-neptune-endpoint:port/openCypher \
  -H "Content-Type: application/x-www-form-urlencoded" \
  --region us-east-1 --service neptune-db \
  -d 'query=CALL neptune.graph.pg_schema.compute({concurrency: 2})'
```

IAM actions required: `neptune-db:ReadDataViaQuery` and `neptune-db:WriteDataViaQuery`

Parameters:

- `concurrency` (Integer, optional) – Number of threads for the background computation. 0 (default) = determined automatically based on hardware. Range: 1 (lowest) to 16 (highest). Use lower values on smaller instances to reduce resource impact.

Behavior:

- Returns immediately with the current status. The computation runs asynchronously in the background.
- If called when state is `Stopped` or `Paused`, the computation resumes from where it left off. For details about these states, see States.
- If called when state is `Completed`, starts a fresh recomputation. The previous schema continues serving reads until the new computation completes.
- If called when a computation is already `InProgress`, Neptune rejects the request with an error message.
- If called during an active bulk load, Neptune rejects the request with an error message.

Response: Returns the status object showing state: `"InProgress"` with the `concurrency` and `progressPercentage` fields.

Stop schema computation

Stops a running background computation.

Syntax:

AWS CLI

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "CALL neptune.graph.pg_schema.stop()"
```

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
```

```
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_query(
    openCypherQuery='CALL neptune.graph.pg_schema.stop()'
)

print(response)
```

awscurl

```
awscurl -X POST https://your-neptune-endpoint:port/openCypher \
  -H "Content-Type: application/x-www-form-urlencoded" \
  --region us-east-1 --service neptune-db \
  -d 'query=CALL neptune.graph.pg_schema.stop()'
```

IAM actions required: `neptune-db:ReadDataViaQuery` and `neptune-db:WriteDataViaQuery`

Behavior:

- Stops the running computation. Progress is saved so it can resume from where it left off when you call `compute()` again.
- A stopped computation does *not* automatically resume on engine restart. You must explicitly call `compute()`.

Response: Returns the status object showing state: `"Stopped"` with the current `progressPercentage`.

Using YIELD with schema results

You can use `YIELD` to extract schema fields and combine them with other queries. The following example retrieves all node labels and counts the number of nodes for each label. The `collSort()` function sorts the list alphabetically:

```
CALL neptune.graph.pg_schema()
  YIELD schema
```

```
  WITH schema.nodeLabels as nl
  UNWIND collSort(nl) as label
  MATCH (n)
  WHERE label in labels(n)
  RETURN label, COUNT(n) as count
```

Sample output:

```
{
  "results": [{
      "label": "airport",
      "count": 3503
    }, {
      "label": "continent",
      "count": 7
    }, {
      "label": "country",
      "count": 237
    }, {
      "label": "version",
      "count": 1
    }]
}
```

Schema computation lifecycle

Asynchronous operation

Schema computation is an asynchronous operation. When you call `neptune.graph.pg_schema.compute()`, it returns immediately with the current status. The computation runs in the background. You poll for progress and completion by calling `neptune.graph.pg_schema()`, which returns the current state and `progressPercentage`.

States

The schema computation moves through the following states:

- `NotStarted` – No schema has been computed yet. `pg_schema()` returns an empty schema.
- `InProgress` – A background computation is running. `pg_schema()` returns partial results (a union of the last complete schema and discoveries from the current computation).
- `Completed` – The computation finished successfully. The full schema is available.

- `Stopped` – The computation was stopped by an explicit call to `stop()`. Partial results are available. Progress is saved so the computation can resume from where it left off when you call `compute()`. A stopped computation does not carry an `errorMessage`.
- `Paused` – Beginning with Neptune engine version 1.5.0.0, an engine restart or failover transitions an `InProgress` computation to `Paused` rather than `Stopped`. Partial results are available, and the computation saves its progress. To resume from where it left off, call `compute()`. Resuming returns the computation to `InProgress` and then `Completed`.
- `Failed` – The computation encountered an error. The last successfully computed schema (if any) remains available. The status object includes an `errorMessage` explaining the failure.

Persistence and restart behavior

The computed schema is persisted and survives engine restarts. The restart behavior depends on the state at the time of restart:

- `InProgress` – If the engine restarts or fails over during computation, the computation is interrupted. Beginning with Neptune engine version 1.5.0.0, the computation transitions to `Paused`; in earlier versions it transitions to `Stopped`. Call `compute()` to resume from where it left off. The computation preserves progress and continues from its last checkpoint.
- `Paused` – The computation does *not* automatically resume. You must call `compute()` to continue from where it left off.
- `Stopped` – The computation does *not* automatically resume. You must call `compute()` to continue from where it left off.
- `Completed` – The schema is loaded and available immediately.

Partial results

While a computation is in progress, `pg_schema()` returns partial results. These include any previously completed schema merged with the labels, properties, and triples discovered so far in the current computation. This means you do not have to wait for a full computation to complete before retrieving useful schema information.

Read replicas

Read replica instances can read the schema using `CALL neptune.graph.pg_schema()`. Neptune replicates the schema from the writer instance and makes it available on replicas almost immediately as schema elements are discovered on the writer.

Read replicas cannot run `compute()` or `stop()`. These calls return an error:

- `compute()` – `"Schema cannot be computed on read replica"`
- `stop()` – `"Schema compute cannot be stopped on read replica"`

Best practices

- **Recompute after mutations** – The schema does not update automatically when data changes. Recompute the schema after bulk loads or significant data mutations. Use the `lastComputedTimestamp` field to determine whether the schema is outdated relative to recent changes in your graph.
- **Concurrency** – The default concurrency value (0) automatically adapts to your instance hardware. For most workloads this is the recommended setting. If the background computation impacts your query workload, specify a lower value (for example, 1 or 2) to reduce resource usage.
- **Stop and resume** – If the background computation impacts your query workload, stop it with `stop()` and resume later during a lower-traffic period by calling `compute()` again. The computation continues from where it left off.
- **Handle restarts gracefully** – If the engine restarts or fails over while a schema computation is in progress, the computation is interrupted. Beginning with Neptune engine version 1.5.0.0, it transitions to `Paused`. Call `compute()` to resume from where it left off. Progress is preserved.
- **Large databases** – For databases with large storage volumes (multiple TB), a full schema computation may take extended time. You can start a compute, let it run until 10–20% progress, then stop. The partial results collected during this window provide a useful schema sample with many labels, properties, and triples already discovered. Read the partial schema with `pg_schema()` while the computation is in progress or after stopping. Resume later when your workload allows.

Limitations

- **Deletes require recompute** – Deleted labels, properties, and triples are only removed from the schema after the next full recomputation. Until then, deleted elements might still appear in schema results.
- **OpenCypher only** – You can call this procedure only through the openCypher query endpoint.

- **Cannot compute during bulk load** – Neptune rejects schema computation while a bulk load operation is active. Trigger compute after the bulk load completes.

Sample output

The following example shows the schema output for the air-routes dataset:

```
awscurl -X POST https://your-neptune-endpoint:port/openCypher \
  -H "Content-Type: application/x-www-form-urlencoded" \
  --region us-east-1 --service neptune-db \
  -d 'query=CALL neptune.graph.pg_schema()'
```

```
{
  "results": [{
      "schema": {
        "edgeLabelDetails": {
          "route": {
            "properties": {
              "dist": ["Int"]
            }
          },
          "contains": {
            "properties": {}
          }
        },
        "edgeLabels": ["route", "contains"],
        "status": {
          "concurrency": "16",
          "lastComputedTimestamp": "2026-06-04T23:58:17Z",
          "state": "Completed",
          "progressPercentage": "100"
        },
        "nodeLabels": ["version", "continent", "airport", "country"],
        "labelTriples": [{
            "~type": "route",
            "~from": "airport",
            "~to": "airport"
          }, {
            "~type": "contains",
            "~from": "country",
            "~to": "airport"
          }, {
```

```
            "~type": "contains",
            "~from": "continent",
            "~to": "airport"
          }],
        "nodeLabelDetails": {
          "continent": {
            "properties": {
              "type": ["String"],
              "code": ["String"],
              "desc": ["String"]
            }
          },
          "airport": {
            "properties": {
              "type": ["String"],
              "city": ["String"],
              "icao": ["String"],
              "code": ["String"],
              "country": ["String"],
              "lat": ["Double"],
              "longest": ["Int"],
              "runways": ["Int"],
              "desc": ["String"],
              "lon": ["Double"],
              "region": ["String"],
              "elev": ["Int"]
            }
          },
          "country": {
            "properties": {
              "type": ["String"],
              "code": ["String"],
              "desc": ["String"]
            }
          },
          "version": {
            "properties": {
              "date": ["String"],
              "desc": ["String"],
              "author": ["String"],
              "type": ["String"],
              "code": ["String"]
            }
          }
```

```
        }
      }
    }]
}
```

Managing statistics for the Neptune DFE to use

> **Note**
>
> Support for openCypher depends on the DFE query engine in Neptune.
> The DFE engine is controlled using the neptune_dfe_query_engine instance parameter in an instance's DB parameter group.

The DFE engine uses information about the data in your Neptune graph to make effective trade-offs when planning query execution. This information takes the form of statistics that include so-called characteristic sets and predicate statistics that can guide query planning.

Starting with engine release 1.2.1.0, you can retrieve summary information about your graph from these statistics using the GetGraphSummary API or the `summary` endpoint.

These DFE statistics are currently re-generated whenever either more than 10% of data in your graph has changed or when the latest statistics are more than 10 days old. However, these triggers may change in the future.

> **Note**
>
> Statistics generation is disabled on T3 and T4g instances because it can exceed the memory capacity of those instance types.

You can manage the generation of DFE statistics through one of the following endpoints:

- `https://`*`your-neptune-host`*`:`*`port`*`/rdf/statistics` (for SPARQL).
- `https://`*`your-neptune-host`*`:`*`port`*`/propertygraph/statistics` (for Gremlin and openCypher), and its alternate version: `https://`*`your-neptune-host`*`:`*`port`*`/pg/statistics`.

> **Note**
>
> As of engine release 1.1.1.0, the Gremlin statistics endpoint (`https://your-neptune-host:port/gremlin/statistics`) is being deprecated in favor of the `propertygraph` or pg endpoint. It is still supported for backward compatibility but may be removed in future releases.
> As of engine release 1.2.1.0, the SPARQL statistics endpoint (`https://your-neptune-host:port/sparql/statistics`) is being deprecated in favor of the `rdf` endpoint. It is still supported for backward compatibility but may be removed in future releases.

> **Note**
>
> If a DFE statistics endpoint is on a reader instance, the only requests that it can process are status requests. Other requests will fail with a `ReadOnlyViolationException`.

Size limits for DFE statistic generation

Currently, DFE statistics generation halts if either of the following size limits is reached:

- The number of characteristic sets generated may not exceed 50,000.
- The number of predicate statistics generated may not exceed one million.

These limits may change.

Current status of DFE statistics

You can check the current status of DFE statistics as follows:

> **Note**
>
> The following examples use the property graph endpoint and commands. For RDF data, use `get-sparql-statistics` for the AWS CLI, and `/rdf/statistics` for REST endpoints.

AWS CLI

```
aws neptunedata get-propertygraph-statistics \
  --endpoint-url https://your-neptune-endpoint:port \
  --region us-east-1
```

> **Note**
>
> Replace *us-east-1* with the Region of your Neptune cluster. You can omit `--region` if you have configured a default Region.

For more information, see get-propertygraph-statistics in the AWS CLI Command Reference.

SDK

```
import boto3
import json
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_propertygraph_statistics()

print(json.dumps(response['payload'], indent=2, default=str))
```

awscurl

```
awscurl https://your-neptune-endpoint:port/propertygraph/statistics \
  --region us-east-1 \
  --service neptune-db
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

```
curl -G https://your-neptune-endpoint:port/propertygraph/statistics
```

The response to a status request contains the following fields:

- `status` – the HTTP return code of the request. If the request succeeded, the code is `200`. See Common errors for a list of common errors.
- `payload`:
  - `autoCompute` – (Boolean) Indicates whether or not automatic statistics generation is enabled.
  - `active` – (Boolean) Indicates whether or not DFE statistics generation is enabled at all.
  - `statisticsId` – Reports the ID of the current statistics generation run. A value of `-1` indicates that no statistics have been generated.
  - `date` – The UTC time at which DFE statistics have most recently been generated, in ISO 8601 format.

    > **Note**
    >
    > Prior to engine release 1.2.1.0, this was represented with minute precision, but from engine release 1.2.1.0 forward, it is represented with millisecond precision (for example, `2023-01-24T00:47:43.319Z`).

  - `note` – A note about problems in the case where statistics are invalid.
  - `signatureInfo` – Contains information about the characteristic sets generated in the statistics (prior to engine release 1.2.1.0, this field was named `summary`). These are generally not directly actionable:
    - `signatureCount` – The total number of signatures across all characteristic sets.
    - `instanceCount` – The total number of characteristic-set instances.
    - `predicateCount` – The total number of unique predicates.

The response to a status request when no statistics have been generated looks like this:

```
{
  "status" : "200 OK",
  "payload" : {
    "autoCompute" : true,
    "active" : false,
    "statisticsId" : -1
   }
}
```

If DFE statistics are available, the response looks like this:

```
{
  "status" : "200 OK",
  "payload" : {
    "autoCompute" : true,
    "active" : true,
    "statisticsId" : 1588893232718,
    "date" : "2020-05-07T23:13Z",
    "summary" : {
      "signatureCount" : 5,
      "instanceCount" : 1000,
      "predicateCount" : 20
    }
  }
}
```

If the generation of DFE statistics failed, for example because it exceeded the statistics size limitation, the response looks like this:

```
{
  "status" : "200 OK",
  "payload" : {
    "autoCompute" : true,
    "active" : false,
    "statisticsId" : 1588713528304,
    "date" : "2020-05-05T21:18Z",
    "note" : "Limit reached: Statistics are not available"
  }
}
```

Disabling automatic generation of DFE statistics

By default, auto-generation of DFE statistics is enabled when you enable DFE.

You can disable auto-generation as follows:

> **Note**
>
> The following examples use the property graph endpoint and commands. For RDF data, use `manage-sparql-statistics` for the AWS CLI, and `/rdf/statistics` for REST endpoints.

AWS CLI

```
aws neptunedata manage-propertygraph-statistics \
  --endpoint-url https://your-neptune-endpoint:port \
  --mode disableAutoCompute
```

For more information, see manage-propertygraph-statistics in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.manage_propertygraph_statistics(mode='disableAutoCompute')

print(response['status'])
```

awscurl

```
awscurl https://your-neptune-endpoint:port/propertygraph/statistics \
  --region us-east-1 \
  --service neptune-db \
```

```
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{"mode":"disableAutoCompute"}'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

```
curl -X POST -d '{ "mode" : "disableAutoCompute" }' https://your-neptune-endpoint:port/propertygraph/statistics
```

If the request is successful, the HTTP response code is `200` and the response is:

```
{
  "status" : "200 OK"
}
```

You can confirm that automatic generation is disabled by issuing a status request and checking that the `autoCompute` field in the response is set to `false`.

Disabling auto-generation of statistics does not terminate a statistics compution that is in progress.

If you make a request to disable auto-generation to a reader instance rather than the writer instance of your DB cluster, the request fails with an HTTP return code of 400 and output like the following:

```
{
  "detailedMessage" : "Writes are not permitted on a read replica instance",
  "code" : "ReadOnlyViolationException",
  "requestId":"8eb8d3e5-0996-4a1b-616a-74e0ec32d5f7"
}
```

See Common errors for a list of other common errors.

Re-enabling automatic generation of DFE statistics

By default, auto-generation of DFE statistics is already enabled when you enable DFE. If you disable auto-generation, you can re-enable it later as follows:

> **Note**
>
> The following examples use the property graph endpoint and commands. For RDF data, use `manage-sparql-statistics` for the AWS CLI, and `/rdf/statistics` for REST endpoints.

AWS CLI

```
aws neptunedata manage-propertygraph-statistics \
  --endpoint-url https://your-neptune-endpoint:port \
  --mode enableAutoCompute
```

For more information, see manage-propertygraph-statistics in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.manage_propertygraph_statistics(mode='enableAutoCompute')

print(response['status'])
```

awscurl

```
awscurl https://your-neptune-endpoint:port/propertygraph/statistics \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
```

```
  -H 'Content-Type: application/json' \
  -d '{"mode":"enableAutoCompute"}'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

```
curl -X POST -d '{ "mode" : "enableAutoCompute" }' https://your-neptune-endpoint:port/propertygraph/statistics
```

If the request is successful, the HTTP response code is `200` and the response is:

```
{
  "status" : "200 OK"
}
```

You can confirm that automatic generation is enabled by issuing a status request and checking that the `autoCompute` field in the response is set to `true`.

Manually triggering the generation of DFE statistics

You can initiate DFE statistics generation manually as follows:

> **Note**
>
> The following examples use the property graph endpoint and commands. For RDF data, use `manage-sparql-statistics` for the AWS CLI, and `/rdf/statistics` for REST endpoints.

AWS CLI

```
aws neptunedata manage-propertygraph-statistics \
```

```
  --endpoint-url https://your-neptune-endpoint:port \
  --mode refresh
```

For more information, see manage-propertygraph-statistics in the AWS CLI Command Reference.

SDK

```
import boto3
import json
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.manage_propertygraph_statistics(mode='refresh')

print(json.dumps(response['payload'], indent=2, default=str))
```

awscurl

```
awscurl https://your-neptune-endpoint:port/propertygraph/statistics \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{"mode":"refresh"}'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

```
curl -X POST -d '{ "mode" : "refresh" }' https://your-neptune-endpoint:port/
propertygraph/statistics
```

If the request succeeds, the output is as follows, with an HTTP return code of 200:

```
{
  "status" : "200 OK",
  "payload" : {
    "statisticsId" : 1588893232718
  }
}
```

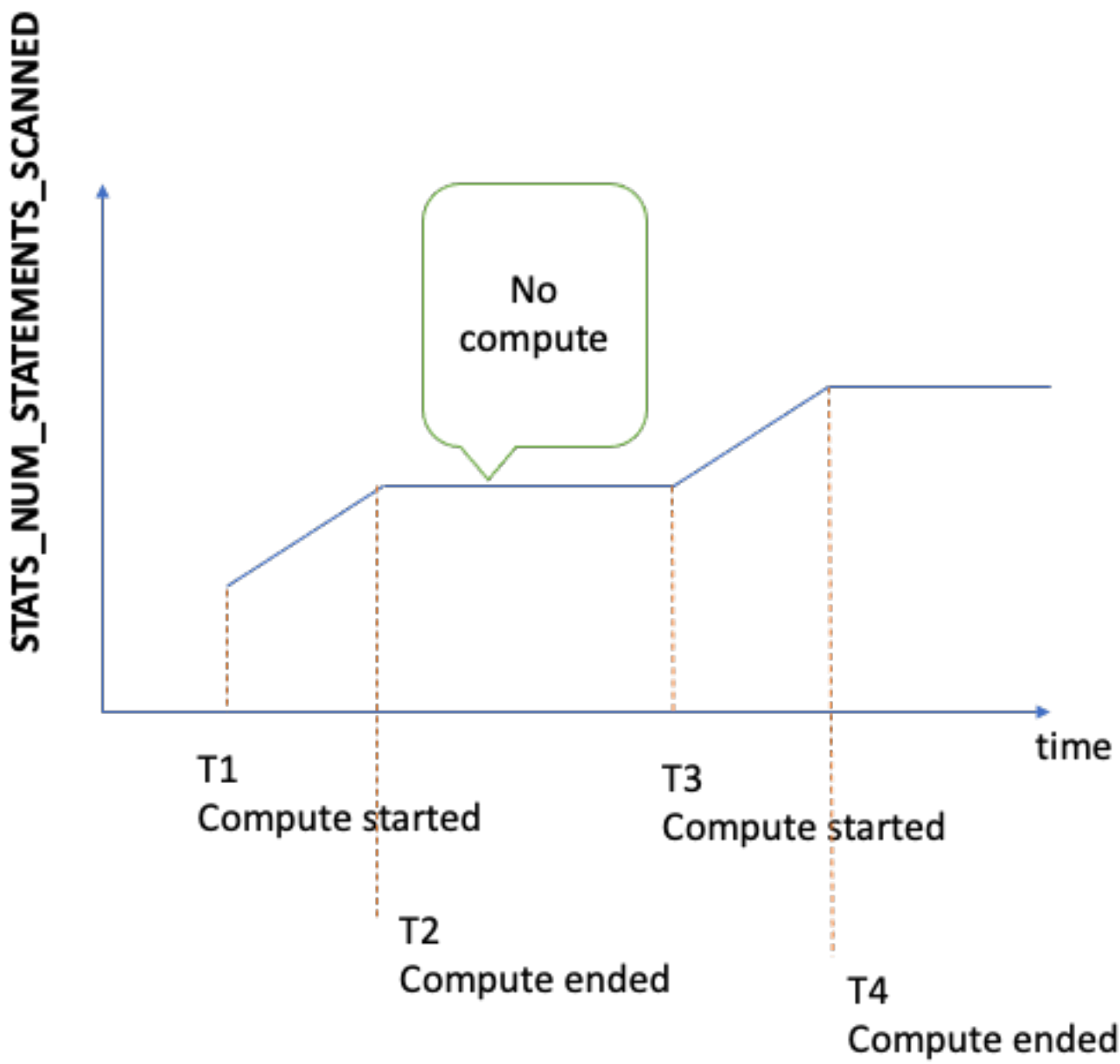
The `statisticsId` in the output is the ID of the statistics generation run that is currently occurring. If a run was already in process at the time of the request, the request returns the ID of that run rather than initiating a new one. Only one statistics generation run can occur at a time.

If a fail-over happens while DFE statistics are being generated, the new writer node will pick up the last processed checkpoint and resume the statistics run from there.

Using the `StatsNumStatementsScanned` CloudWatch metric to monitor statistics computation

The `StatsNumStatementsScanned` CloudWatch metric returns the total number of statements scanned for statistics computation since the server started. It is updated at each statistics computation slice.

Every time statistics computation is triggered, this number increases, and when no computation is happening, it remains constant. Looking at a plot of `StatsNumStatementsScanned` values over time therefore gives you a pretty clear picture of when statistics computation was happening and how fast:



When computation is happening, the slope of the graph shows you how fast (the steeper the slope, the faster statistics are being computed).

If the graph is simply a flat line at 0, the statistics feature has been enabled, but no statistics have been computed at all. If the statistics feature has been disabled, or if you're using an engine version that does not support statistics computation, the `StatsNumStatementsScanned` does not exist.

As mentioned earlier, you can disable statistics computation using the statistics API, but leaving it off can result in statistics not being up to date, which in turn can result in poor query plan generation for the DFE engine.

See Monitoring Neptune Using Amazon CloudWatch for information about how to use CloudWatch.

Using AWS Identity and Access Management (IAM) authentication with DFE statistics endpoints

You can access DFE statistics endpoints securely with IAM authentication by using the AWS CLI, awscurl, or any other tool that works with HTTPS and IAM. For information about setting up the proper credentials, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

The following examples show how to make an authenticated status request:

> **Note**
>
> The following examples use the property graph endpoint and commands. For RDF data, use the equivalent SPARQL commands (`get-sparql-statistics`, `manage-sparql-statistics`) for the AWS CLI, and `/rdf/statistics` for REST endpoints.

AWS CLI

```
aws neptunedata get-propertygraph-statistics \
  --endpoint-url https://your-neptune-endpoint:port
```

The AWS CLI handles IAM authentication automatically using your configured credentials.

SDK

```
import boto3
import json
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_propertygraph_statistics()
```

```
print(json.dumps(response['payload'], indent=2, default=str))
```

The SDK handles IAM authentication automatically using your configured credentials.

awscurl

```
awscurl https://your-neptune-endpoint:port/propertygraph/statistics \
  --region us-east-1 \
  --service neptune-db
```

> **Note**
>
> Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -G https://your-neptune-endpoint:port/propertygraph/statistics
```

> **Note**
>
> This example does not include IAM authentication. To use IAM authentication with **curl**, you must manually sign the request. Consider using the AWS CLI, SDK, or **awscurl** instead.

The following examples manually initiate statistics generation with authentication:

AWS CLI

```
aws neptunedata manage-propertygraph-statistics \
  --endpoint-url https://your-neptune-endpoint:port \
  --mode refresh
```

SDK

```
import boto3
import json
from botocore.config import Config
```

```
client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.manage_propertygraph_statistics(mode='refresh')

print(json.dumps(response['payload'], indent=2, default=str))
```

awscurl

Create a JSON file named `request.json` that contains:

```
{ "mode" : "refresh" }
```

Then run:

```
awscurl https://your-neptune-endpoint:port/propertygraph/statistics \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d @request.json
```

curl

```
curl -X POST https://your-neptune-endpoint:port/propertygraph/statistics \
  -H 'Content-Type: application/json' \
  -d '{ "mode" : "refresh" }'
```

> **Note**
>
> This example does not include IAM authentication. To use IAM authentication with **curl**, you must manually sign the request. Consider using the AWS CLI, SDK, or **awscurl** instead.

Deleting DFE statistics

You can delete all statistics in the database as follows:

> **Note**
>
> The following examples use the property graph endpoint and commands. For RDF data, use `delete-sparql-statistics` for the AWS CLI, and `/rdf/statistics` for REST endpoints.

AWS CLI

```
aws neptunedata delete-propertygraph-statistics \
  --endpoint-url https://your-neptune-endpoint:port
```

For more information, see delete-propertygraph-statistics in the AWS CLI Command Reference.

SDK

```
import boto3
import json
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.delete_propertygraph_statistics()

print(json.dumps(response['payload'], indent=2, default=str))
```

awscurl

```
awscurl https://your-neptune-endpoint:port/propertygraph/statistics \
  --region us-east-1 \
  --service neptune-db \
  -X DELETE
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

```
curl -X DELETE https://your-neptune-endpoint:port/propertygraph/statistics
```

Valid HTTP return codes are:

- `200` – the delete was successful.

  In this case, a typical response would look like:

```
{
  "status" : "200 OK",
  "payload" : {
      "active" : false,
      "statisticsId" : -1
  }
}
```

- `204` – there were no statistics to delete.

  In this case, the response is blank (no response).

If you send a delete request to a statistics endpoint on a reader node, a `ReadOnlyViolationException` is thrown.

Common error codes for DFE statistics request

The following is a list of common errors that can occur when you make a request to a statistics endpoint:

- `AccessDeniedException` – *Return code:* `400`. *Message:* `Missing Authentication Token`.
- `BadRequestException` (for Gremlin and openCypher) – *Return code:* `400`. *Message:* `Bad route: /pg/statistics`.
- `BadRequestException` (for RDF data) – *Return code:* `400`. *Message:* `Bad route: /rdf/statistics`.
- `InvalidParameterException` – *Return code:* `400`. *Message:* `Statistics command parameter 'mode' has unsupported value '`*`the invalid value`*`'`.
- `MissingParameterException` – *Return code:* `400`. *Message:* `Content-type header not specified.`.
- `ReadOnlyViolationException` – *Return code:* `400`. *Message:* `Writes are not permitted on a read replica instance`.

For example, if you make a request when the DFE and statistics are not enabled, you would get a response like the following:

```
{
  "code" : "BadRequestException",
  "requestId" : "b2b8f8ee-18f1-e164-49ea-836381a3e174",
  "detailedMessage" : "Bad route: /sparql/statistics"
}
```

The Neptune lookup cache can accelerate read queries

Amazon Neptune implements a lookup cache that uses the R5d instance's NVMe-based SSD to improve read performance for queries with frequent, repetitive lookups of property values or RDF literals. The lookup cache temporarily stores these values in the NVMe SSD volume where they can be accessed rapidly.

Read queries that return the properties of a large number of vertices and edges, or many RDF triples, can have a high latency if the property values or literals need to be retrieved from cluster storage volumes rather than memory. Examples include long-running read queries that return a large number of full names from an identity graph, or of IP addresses from a fraud-detection graph. As the number of property values or RDF literals returned by your query increases, available memory decreases and your query execution can significantly degrade.

Use cases for the Neptune lookup cache

The lookup cache only helps when your read queries are returning the properties of a very large number of vertices and edges, or of RDF triples.

To optimize query performance, Amazon Neptune uses the R5d instance type to create a large cache for such property values or literals. Retrieving them from the cache is then much faster than retrieving them from cluster storage volumes.

As a rule of thumb, it's only worthwhile to enable the lookup cache if all three of the following conditions are met:

- You have been observing increased latency in read queries.
- You're also observing a drop in the `BufferCacheHitRatio` CloudWatch metric when running read queries (see Monitoring Neptune Using Amazon CloudWatch).
- Your read queries are spending a lot of time in materializing return values prior to rendering the results (see the Gremlin-profile example below for a way to determine how many property values are being materialized for a query).

> **Note**
>
> This feature is helpful *only* in the specific scenario described above. For example, the lookup cache doesn't help aggregation queries at all. Unless you are running queries that

would benefit from the lookup cache, there is no reason to use an R5d instance type instead of an equivalent and less expensive R5 instance type.

If you're using Gremlin, you can assess the materialization costs of a query with the Gremlin `profile` API. Under "Index Operations', it shows the number of terms materialized during execution:

```
Index Operations
Query execution:
    # of statement index ops: 3
    # of unique statement index ops: 3
    Duplication ratio: 1.0
    # of terms materialized: 5273
Serialization:
    # of statement index ops: 200
    # of unique statement index ops: 140
    Duplication ratio: 1.43
    # of terms materialized: 32693
```

The number of non-numerical terms that are materialized is directly proportional to the number of term look-ups that Neptune has to perform.

Using the lookup cache

The lookup cache is only available on an R5d instance type, where it is automatically enabled by default. Neptune R5d instances have the same specifications as R5 instances, plus up to 1.8 TB of local NVMe-based SSD storage. Lookup caches are instance-specific, and workloads that benefit can be directed specifically to R5d instances in a Neptune cluster, while other workloads can be directed to R5 or other instance types.

To use the lookup cache on a Neptune instance, simply upgrade that instance to the R5d instance type. When you do, Neptune automatically sets the neptune_lookup_cache DB cluster parameter to 1 (enabled), and creates the lookup cache on that particular instance. You can then use the Instance Status API to confirm that the cache has been enabled.

Similarly, to disable the lookup cache on a given instance, scale the instance down from an R5d instance type to an equivalent R5 instance type.

When an R5d instance is launched, the lookup cache is enabled and in cold-start mode, meaning that it is empty. Neptune first checks in the lookup cache for property values or RDF literals while processing queries, and adds them if they are not yet present. This gradually warm up the cache.

When you direct the read queries that require property-value or RDF-literal lookups to an R5d *reader* instance, read performance degrades slightly while its cache is warming up. When the cache is warmed up, however, read performance speeds up significantly and you may also see a drop in I/O costs related to lookups hitting the cache rather than cluster storage. Memory utilization also improves.

If your *writer* instance is an R5d, it warms up its lookup cache automatically on every write operation. This approach does increase latency for write queries slightly, but warms up the lookup cache more efficiently. Then if you direct the read queries that require property-value or RDF-literal lookups to the writer instance, you start getting improved read performance immediately, since the values have already been cached there.

Also, if you are running the bulk loader on an R5d writer instance, you may notice that its performance is slightly degraded because of the cache.

Because the lookup cache is specific to each node, host replacement resets the cache to a cold start.

You can temporarily disable the lookup cache on all instances in your DB cluster by setting the neptune_lookup_cache DB cluster parameter to `0` (disabled). In general, however, it makes more sense to disable the cache on specific instances by scaling them down from R5d to R5 instance types.

Transaction Semantics in Neptune

Amazon Neptune is designed to support highly concurrent online transactional processing (OLTP) workloads over data graphs. The W3C SPARQL Query Language for RDF specification and the Apache TinkerPop Gremlin Graph Traversal Language documentation do not define transaction semantics for concurrent query processing. Because ACID support and well-defined transaction guarantees can be very important, we enforce strict semantics to help avoid data anomalies.

This section defines these semantics and illustrates how they apply to some common use cases in Neptune.

Topics

- Definition of Isolation Levels
- Transaction Isolation Levels in Neptune
- Examples of Neptune transaction semantics

Definition of Isolation Levels

The "I" in `ACID` stands for *isolation*. The degree of isolation of a transaction determines how much or little other concurrent transactions can affect the data that it operates on.

The SQL:1992 Standard created a vocabulary for describing isolation levels. It defines three types of interactions (that it calls *phenomena*) that can occur between two concurrent transactions, `Tx1` and `Tx2`:

- `Dirty read` – This occurs when `Tx1` modifies an item, and then `Tx2` reads that item before `Tx1` has committed the change. Then, if `Tx1` never succeeds in committing the change, or rolls it back, `Tx2` has read a value that never made it into the database.
- `Non-repeatable read` – This happens when `Tx1` reads an item, then `Tx2` modifies or deletes that item and commits the change, and then `Tx1` tries to reread the item. `Tx1` now reads a different value than before, or finds that the item no longer exists.
- `Phantom read` – This happens when `Tx1` reads a set of items that satisfy a search criterion, and then `Tx2` adds a new item that satisfies the search criterion, and then `Tx1` repeats the search. `Tx1` now obtains a different set of items than it did before.

Each of these three types of interaction can cause inconsistencies in the resulting data in a database.

The SQL:1992 standard defined four isolation levels that have different guarantees in terms of the three types of interaction and the inconsistencies that they can produce. At all four levels, a transaction can be guaranteed to execute completely or not at all:

- `READ UNCOMMITTED` – Allows all three kinds of interaction (that is, dirty reads, non-repeatable reads, and phantom reads).
- `READ COMMITTED` – Dirty reads are not possible, but nonrepeatable and phantom reads are.
- `REPEATABLE READ` – Neither dirty reads nor nonrepeatable reads are possible, but phantom reads still are.
- `SERIALIZABLE` – None of the three types of interaction phenomena can occur.

Multiversion concurrency control (MVCC) allows one other kind of isolation, namely *SNAPSHOT* isolation. This guarantees that a transaction operates on a snapshot of data as it exists when the transaction begins, and that no other transaction can change that snapshot.

Transaction Isolation Levels in Neptune

Amazon Neptune implements different transaction isolation levels for read-only queries and for mutation queries. SPARQL and Gremlin queries are classified as read-only or mutation based on the following criteria:

- In SPARQL, there is a clear distinction between read queries (`SELECT`, `ASK`, `CONSTRUCT`, and `DESCRIBE` as defined in the SPARQL 1.1 Query Language specification), and mutation queries (`INSERT` and `DELETE` as defined in the SPARQL 1.1 Update specification).

  Note that Neptune treats multiple mutation queries submitted together (for example, in a `POST` message, separated by semicolons) as a single transaction. They are guaranteed either to succeed or fail as an atomic unit, and in the case of failure, partial changes are rolled back.
- However, in Gremlin, Neptune classifies a query as a read-only query or a mutation query based on whether it contains any query-path steps such as `addE()`, `addV()`, `property()`, or `drop()` that manipulates data. If the query contains any such path step, it is classified and executed as a mutation query.

It is also possible to use standing sessions in Gremlin. For more information, see Gremlin script-based sessions. In these sessions, all queries, including read-only queries, are executed under the same isolation as mutation queries on the writer endpoint.

Using bolt read-write sessions in openCypher, all queries including read-only queries are executed under the same isolation as mutation queries, on the writer endpoint.

Topics

- Read-only query isolation in Neptune
- Mutation query isolation in Neptune
- Conflict Resolution Using Lock-Wait Timeouts
- Range locks and false conflicts

Read-only query isolation in Neptune

Neptune evaluates read-only queries under snapshot isolation semantics. This means that a read-only query logically operates on a consistent snapshot of the database taken when query evaluation begins. Neptune can then guarantee that none of the following phenomena will happen:

- `Dirty reads` – Read-only queries in Neptune will never see uncommitted data from a concurrent transaction.
- `Non-repeatable reads` – A read-only transaction that reads the same data more than once will always get back the same values.
- `Phantom reads` – A read-only transaction will never read data that was added after the transaction began.

Because snapshot isolation is achieved using multiversion concurrency control (MVCC), read-only queries have no need to lock data and therefore do not block mutation queries.

Read replicas only accept read-only queries, so all queries against read replicas execute under `SNAPSHOT` isolation semantics.

The only additional consideration when querying a read replica is that there can be a small replication lag between the writer and read replicas. This means that an update made on the writer might take a short time to be propagated to the read replica you are reading from. The actual replication time depends on the write-load against the primary instance. Neptune architecture

supports low-latency replication and the replication lag is instrumented in an Amazon CloudWatch metric.

Still, because of the `SNAPSHOT` isolation level, read queries always see a consistent state of the database, even if it is not the most recent one.

In cases where you require a strong guarantee that a query observes the result of a previous update, send the query to the writer endpoint itself rather than to a read replica.

Mutation query isolation in Neptune

Reads made as part of mutation queries are executed under `READ COMMITTED` transaction isolation, which rules out the possibility of dirty reads. Going beyond the usual guarantees provided for `READ COMMITTED` transaction isolation, Neptune provides the strong guarantee that neither `NON-REPEATABLE` nor `PHANTOM` reads can happen.

These strong guarantees are achieved by locking records and ranges of records when reading data. This prevents concurrent transactions from making insertions or deletions in index ranges after they have been read, thus guaranteeing repeatable reads.

> **Note**
>
> However, a concurrent mutation transaction `Tx2` could begin after the start of mutation transaction `Tx1`, and could commit a change before `Tx1` had locked data to read it. In that case, `Tx1` would see `Tx2`'s change just as if `Tx2` had completed before `Tx1` started. Because this only applies to committed changes, a `dirty read` could never occur.

To understand the locking mechanism that Neptune uses for mutation queries, it helps first to understand the details of the Neptune Graph Data Model and Indexing Strategy. Neptune manages data using three indexes, namely `SPOG`, `POGS`, and `GPSO`.

To achieve repeatable reads for the `READ COMMITTED` transaction level, Neptune takes range locks in the index that is being used. For example, if a mutation query reads all properties and outgoing edges of a vertex named `person1`, the node would lock the entire range defined by the prefix `S=person1` in the `SPOG` index before reading the data.

The same mechanism applies when using other indexes. For example, when a mutation transaction looks up all the source-target vertex pairs for a given edge label using the `POGS` index, the range

for the edge label in the P position would be locked. Any concurrent transaction, regardless of whether it was a read-only or mutation query, could still perform reads within the locked range. However, any mutation involving insertion or deletion of new records in the locked prefix range would require an exclusive lock and would be prevented.

In other words, when a range of the index has been read by a mutation transaction, there is a strong guarantee that this range will not be modified by any concurrent transactions until the end of the reading transaction. This guarantees that no `non-repeatable reads` will occur.

Conflict Resolution Using Lock-Wait Timeouts

If a second transaction tries to modify a record in a range that a first transaction has locked, Neptune detects the conflict immediately and blocks the second transaction.

If no dependency deadlock is detected, Neptune automatically applies a lock-wait timeout mechanism, in which the blocked transaction waits for up to 60 seconds for the transaction that holds the lock to finish and release the lock.

- If the lock-wait timeout expires before the lock is released, the blocked transaction is rolled back.
- If the lock is released within the lock-wait timeout, the second transaction is unblocked and can finish successfully without needing to retry.

However, if Neptune detects a dependency deadlock between the two transactions, automatic reconciliation of the conflict is not possible. In this case, Neptune immediately cancels and rolls back one of the two transactions without initiating a lock-wait timeout. Neptune makes a best effort to roll back the transaction that has the fewest records inserted or deleted.

Measuring lock-wait time (engine ≥ 1.4.5.0)

Starting with engine version 1.4.5.0, you can observe exactly how long a mutation query was blocked by using two slow-query-log counters:

Counter	Description
`sharedLocksWaitTimeMillis`	Time spent waiting to obtain shared (S) locks, which allow multiple readers but block writers.
`exclusiveLocksWaitTimeMillis`	Time spent waiting to obtain exclusive (X) locks, which block all other access.

These two fields appear in the `storageCounters` object only when you enable slow-query logging in debug mode (`neptune_enable_slow_query_log=debug`).

> **Tip**
>
> If `sharedLocksWaitTimeMillis + exclusiveLocksWaitTimeMillis` approaches the query's `overallRunTimeMs`, the query is bottlenecked by lock contention rather than CPU, network, or I/O.

Practical tips for reducing contention:

- **Stagger conflicting jobs** – Run heavy batch mutations during periods of lower user activity.
- **Break large mutations into smaller chunks** – Smaller transactions hold locks for less time, reducing the chance of timeouts.

Range locks and false conflicts

Neptune takes range locks using gap locks. A gap lock is a lock on a gap between index records, or a lock on the gap before the first or after the last index record.

Neptune uses a so-called dictionary table to associate numeric ID values with specific string literals. Here is a sample state of such a Neptune dictionary: table:

String	ID
type	1
default_graph	2
person_3	3
person_1	5
knows	6
person_2	7
age	8

String	ID
edge_1	9
lives_in	10
New York	11
Person	12
Place	13
edge_2	14

The strings above belong to a property-graph model, but the concepts apply equally to all RDF graph models as well.

The corresponding state of the SPOG (Subject-Predicate-Object_Graph) index is shown below on the left. On the right, the corresponding strings are shown, to help understand what the index data means.

S (ID)	P (ID)	O (ID)	G (ID)		S (string)	P (string)	O (string)	G (string)
3	1	12	2		person_3	type	Person	default_graph
5	1	12	2		person_1	type	Person	default_graph
5	6	3	9		person_1	knows	person_3	edge_1
5	8	40	2		person_1	age	40	default_graph
5	10	11	14		person_1	lives_in	New York	edge_2

S (ID)	P (ID)	O (ID)	G (ID)		S (string)	P (string)	O (string)	G (string)
7	1	12	2		person_2	type	Person	default_graph
11	1	13	2		New York	type	Place	default_graph

Now, if a mutation query reads all properties and outgoing edges of a vertex named `person_1`, the node would lock the entire range defined by the prefix `S=person_1` in the SPOG index before reading the data. The range lock would place gap locks on all matching records and the first record that is not a match. Matching records would be locked, and non-matching records would not be locked. Neptune would place the gap-locks as follows:

- `5 1 12 2` *(gap 1)*
- `5 6 3 9` *(gap 2)*
- `5 8 40 2` *(gap 3)*
- `5 10 11 14` *(gap 4)*
- `7 1 12 2` *(gap 5)*

This locks the following records:

- `5 1 12 2`
- `5 6 3 9`
- `5 8 40 2`
- `5 10 11 14`

In this state, the following operations are legitimately blocked:

- Insertion of a new property or edge for `S=person_1`. A new property different from `type` or a new edge would have to go in either gap 2, gap 3, gap 4, or gap 5, all of which are locked.
- Deletion of any of the existing records.

At the same time, a few concurrent operations would be blocked falsely (generating false conflicts):

- Any property or edge insertions for `S=person_3` are blocked because they would have to go in gap 1.
- Any new vertex insertion which gets assigned an ID between 3 and 5 would be blocked because it would have to go in gap 1.
- Any new vertex insertion which gets assigned an ID between 5 and 7 would be blocked because it would have to go in gap 5.

Gap locks are not precise enough to lock the gap for one specific predicate (for example, to lock gap5 for predicate S=5).

The range locks are only placed in the index where the read happens. In the case above, records are locked only in the SPOG index, not in POGS or GPSO. Reads for a query may be performed across all indexes depending on the access patterns, which can be listed using the `explain` APIs (for Sparql and for Gremlin).

> **Note**
>
> Gap locks can also be taken for safe concurrent updates on underlying indexes, which can also lead to false conflicts. These gap locks are placed independent of isolation level or read operations performed by the transaction.

False conflicts can happen not only when *concurrent* transactions collide because of gap locks, but also in some cases when a transaction is being retried after any sort of failure. If the roll-back that was triggered by the failure is still in progress and the locks previously taken for the transaction have not yet been fully released, the retry will encounter a false conflict and fail.

Under a high load, you might typically find that 3-4% of write queries fail because of false conflicts. For an external client, such false conflicts are hard to predict, and should be handled using retries.

Examples of Neptune transaction semantics

The following examples illustrate different use cases for transaction semantics in Amazon Neptune.

Topics

- Example 1 – Inserting a Property Only If It Does Not Exist
- Example 2 – Asserting That a Property Value Is Globally Unique
- Example 3 – Changing a Property If Another Property Has a Specified Value
- Example 4 – Replacing an Existing Property
- Example 5 – Avoiding Dangling Properties or Edges

Example 1 – Inserting a Property Only If It Does Not Exist

Suppose that you want to ensure that a property is set only once. For example, suppose that multiple queries are trying to assign a person a credit score concurrently. You only want one instance of the property to be inserted, and the other queries to fail because the property has already been set.

```
# GREMLIN:
g.V('person1').hasLabel('Person').coalesce(has('creditScore'), property('creditScore',
 'AAA+'))

# SPARQL:
INSERT { :person1 :creditScore "AAA+" .}
WHERE  { :person1 rdf:type :Person .
         FILTER NOT EXISTS { :person1 :creditScore ?o .} }
```

The Gremlin `property()` step inserts a property with the given key and value. The `coalesce()` step executes the first argument in the first step, and if it fails, then it executes the second step:

Before inserting the value for the `creditScore` property for a given `person1` vertex, a transaction must try to read the possibly non-existent `creditScore` value for `person1`. This attempted read locks the SP range for `S=person1` and `P=creditScore` in the SPOG index where the `creditScore` value either exists or will be written.

Taking this range lock prevents any concurrent transaction from inserting a `creditScore` value concurrently. When there are multiple parallel transactions, at most one of them can update the value at a time. This rules out the anomaly of more than one `creditScore` property being created.

Example 2 – Asserting That a Property Value Is Globally Unique

Suppose that you want to insert a person with a Social Security number as a primary key. You would want your mutation query to guarantee that, at a global level, no one else in the database has that same Social Security number:

```
# GREMLIN:
g.V().has('ssn', 123456789).fold()
  .coalesce(__.unfold(),
            __.addV('Person').property('name', 'John Doe').property('ssn', 123456789'))

# SPARQL:
INSERT { :person1 rdf:type :Person .
         :person1 :name "John Doe" .
         :person1 :ssn 123456789 .}
WHERE  { FILTER NOT EXISTS { ?person :ssn 123456789 } }
```

This example is similar to the previous one. The main difference is that the range lock is taken on the POGS index rather than the SPOG index.

The transaction executing the query must read the pattern, `?person :ssn 123456789`, in which the P and O positions are bound. The range lock is taken on the POGS index for `P=ssn` and `O=123456789`.

- If the pattern does exist, no action is taken.
- If it does not exist, the lock prevents any concurrent transaction from inserting that Social Security number also

Example 3 – Changing a Property If Another Property Has a Specified Value

Suppose that various events in a game move a person from level one to level two, and assign them a new `level2Score` property set to zero. You need to be sure that multiple concurrent instances of such a transaction could not create multiple instances of the level-two score property. The queries in Gremlin and SPARQL might look like the following.

```
# GREMLIN:
g.V('person1').hasLabel('Person').has('level', 1)
 .property('level2Score', 0)
 .property(Cardinality.single, 'level', 2)
```

```
# SPARQL:
DELETE { :person1 :level 1 .}
INSERT { :person1 :level2Score 0 .
         :person1 :level 2 .}
WHERE  { :person1 rdf:type :Person .
         :person1 :level 1 .}
```

In Gremlin, when `Cardinality.single` is specified, the `property()` step either adds a new property or replaces an existing property value with the new value that is specified.

Any update to a property value, such as increasing the `level` from 1 to 2, is implemented as a deletion of the current record and insertion of a new record with the new property value. In this case, the record with level number 1 is deleted and a record with level number 2 is reinserted.

For the transaction to be able to add `level2Score` and update the `level` from 1 to 2, it must first validate that the `level` value is currently equal to 1. In doing so, it takes a range lock on the SPO prefix for S=`person1`, P=`level`, and O=1 in the SPOG index. This lock prevents concurrent transactions from deleting the version 1 triple, and as a result, no conflicting concurrent updates can happen.

Example 4 – Replacing an Existing Property

Certain events might update a person's credit score to a new value (here BBB). But you want to be sure that concurrent events of that type can't create multiple credit score properties for a person.

```
# GREMLIN:
g.V('person1').hasLabel('Person')
 .sideEffect(properties('creditScore').drop())
 .property('creditScore', 'BBB')

# SPARQL:
DELETE { :person1 :creditScore ?o .}
INSERT { :person1 :creditScore "BBB" .}
WHERE  { :person1 rdf:type :Person .
         :person1 :creditScore ?o .}
```

This case is similar to example 3, except that instead of locking the SPO prefix, Neptune locks the SP prefix with S=`person1` and P=`creditScore` only. This prevents concurrent transactions from inserting or deleting any triples with the `creditScore` property for the `person1` subject.

Example 5 – Avoiding Dangling Properties or Edges

The update on an entity should not leave a dangling element, that is, a property or edge associated to an entity that is not typed. This is only an issue in SPARQL, because Gremlin has built-in constraints to prevent dangling elements.

```
# SPARQL:
tx1: INSERT { :person1 :age 23 } WHERE { :person1 rdf:type :Person }
tx2: DELETE { :person1 ?p ?o }
```

The `INSERT` query must read and lock the SPO prefix with `S=person1`, `P=rdf:type`, and `O=Person` in the SPOG index. The lock prevents the `DELETE` query from succeeding in parallel.

In the race between the `DELETE` query trying to delete the `:person1 rdf:type :Person` record and the `INSERT` query reading the record and creating a range lock on its SPO in the SPOG index, the following outcomes are possible:

- If the `INSERT` query commits before the `DELETE` query reads and deletes all records for `:person1`, `:person1` is removed entirely from the database, including the newly inserted record.
- If the `DELETE` query commits before the `INSERT` query tries to read the `:person1 rdf:type :Person` record, the read observes the committed change. That is, it does not find any `:person1 rdf:type :Person` record and hence becomes a no-op.
- If the `INSERT` query reads before the `DELETE` query does, the `:person1 rdf:type :Person` triple is locked and the `DELETE` query is blocked until the INSERT query commits, as in the first case previously.
- If the `DELETE` reads before the `INSERT` query, and the `INSERT` query tries to read and take a lock on the SPO prefix for the record, a conflict is detected. This is because the triple has been marked for removal, and the `INSERT` then fails.

In all these different possible sequences of events, no dangling edge is created.

Amazon Neptune DB Clusters and Instances

An Amazon Neptune *DB cluster* manages access to your data through queries. A cluster consists of:

- One *primary DB instance.*.
- Up to 15 *read-replica DB instances.*.

All the instances in a cluster share the same underlying managed storage volume, which is designed for reliability and high availability.

You connect to the DB instances in your DB cluster through Neptune endpoints.

The primary DB instance in a Neptune DB cluster

The primary DB instance coordinates all write operations to the DB cluster's underlying storage volume. It also supports read operations.

There can only be one primary DB instance in a Neptune DB cluster. If the primary instance becomes unavailable, Neptune automatically fails over to one of the read-replica instances with a priority that you can specify.

Read-replica DB instances in a Neptune DB cluster

After you create the primary instance for a DB cluster, you can create up to 15 read-replica instances in your DB cluster to support read-only queries.

Neptune read-replica DB instances work well for scaling read capacity because they are fully dedicated to read operations on your cluster volume. All write operations are managed by the primary instance. Each read-replica DB instance has its own endpoint.

Because the cluster storage volume is shared among all instances in a cluster, all read-replica instances return the same data for query results with very little replication lag. This lag is usually much less than 100 milliseconds after the primary instance writes an update, although it can be somewhat longer when the volume of write operations is very large.

Having one or more read-replica instances available in different Availability Zones can increase availability, because read-replicas serve as failover targets for the primary instance. That is, if the primary instance fails, Neptune promotes a read-replica instance to become the primary instance.

When this happens, there is a brief interruption while the promoted instance is rebooted, during which read and write requests made to the primary instance fail with an exception.

By contrast, if your DB cluster doesn't include any read-replica instances, your DB cluster remains unavailable when the primary instance fails until it has been re-created. Re-creating the primary instance takes considerably longer than promoting a read-replica.

To ensure high availability, we recommend that you create one or more read-replica instances that have the same DB instance class as the primary instance and are located in different Availability Zones than the primary instance. See Fault tolerance for a Neptune DB cluster.

Using the console, you can create a Multi-AZ deployment by simply specifying Multi-AZ when creating a DB cluster. If a DB cluster is in a single Availability Zone, you can make it a Multi-AZ DB cluster adding a Neptune replica in a different Availability Zone.

> **Note**
>
> You can't create an encrypted read-replica instance for an unencrypted Neptune DB cluster, or an unencrypted read-replica instance for an encrypted Neptune DB cluster.

For details on how to create a Neptune read-replica DB instance, see Creating a Neptune reader instance using the console.

Sizing DB instances in a Neptune DB cluster

Size the instances in your Neptune DB cluster based on your CPU and memory requirements. The number of vCPUs on an instance determines the number of query threads that handle incoming queries. The amount of memory on an instance determines the size of the buffer cache, used for storing copies of data pages fetched from the underlying storage volume.

Each Neptune DB instance has a number of query threads equal to 2 x number of vCPUs on that instance. An `r5.4xlarge`, for example, with 16 vCPUs, has 32 query threads, and can therefore process 32 queries concurrently.

Additional queries that arrive while all query threads are occupied are put into a server-side queue, and processed in a FIFO manner as query threads become available. This server-side queue can hold approximately 8000 pending requests. Once it's full, Neptune respond to additional requests with a `ThrottlingException`. You can monitor the number of pending requests with the

`MainRequestQueuePendingRequests` CloudWatch metric, or by using the Gremlin query status endpoint with the `includeWaiting` parameter.

Query execution time from a client perspective includes of any time spent in the queue, in addition to the time taken to actually execute the query.

A sustained concurrent write load that utilizes all the query threads on the primary DB instance ideally shows 90% or more CPU utilization, which indicates that all the query threads on the server are actively engaged in doing useful work. However, actual CPU utilization is often somewhat lower, even under a sustained concurrent write load. This is usually because query threads are waiting on I/O operations to the underlying storage volume to complete. Neptune uses quorum writes that make six copies of your data across three Availability Zones, and four out of those six storage nodes must acknowledge a write for it to be considered durable. While a query thread waits for this quorum from the storage volume, it is stalled, which reduces CPU utilization.

If you have a serial write load where you are performing one write after another and waiting for the first to complete before beginning the next, you can expect the CPU utilization to be lower still. The exact amount will be a function of the number of vCPUs and query threads (the more query threads, the less overall CPU per query), with some reduction caused by waiting for I/O.

For more information about how best to size DB instances, see Choosing instance types for Amazon Neptune. For the pricing of each instance-type, please see the Neptune pricing page.

Monitoring DB instance performance in Neptune

You can use CloudWatch metrics in Neptune to monitor the performance of your DB instances and keep track of query latency as observed by the client. See Using CloudWatch to monitor DB instance performance in Neptune.

Amazon Neptune storage, reliability and availability

Amazon Neptune uses a distributed and shared storage architecture that scales automatically as your database storage needs grow.

Neptune data is stored in a cluster volume, which is a single, virtual volume that uses Non-Volatile Memory Express (NVMe) SSD-based drives. A cluster volume can grow to a maximum size of 128 tebibytes (TiB) (64 TiB in the China Regions and the AWS GovCloud (US) Regions). A cluster volume consists of copies of the data across three Availability Zones (AZs) in a single AWS Region, which provides high durability and availability.

The Storage section covers these topics in detail:

- **Allocation** – You are charged for space allocated, as determined by the storage high water mark.
- **Storage billing** – Storage costs are billed based on the high water mark, and Neptune also offers an I/O–Optimized pricing option for I/O–intensive workloads.
- **Reliability and high availability** – Neptune replicates your data across three Availability Zones for high durability and automatic fault recovery.

Connecting to Amazon Neptune Endpoints

Amazon Neptune uses a cluster of DB instances rather than a single instance. Each Neptune connection is handled by a specific DB instance. When you connect to a Neptune cluster, the host name and port that you specify point to an intermediate handler called an *endpoint*. An endpoint is a URL that contains a host address and a port. Neptune endpoints use encrypted Transport Layer Security/Secure Sockets Layer (TLS/SSL) connections.

Neptune uses the endpoint mechanism to abstract these connections so that you don't have to hardcode the hostnames, or write your own logic for rerouting connections when some DB instances are unavailable.

Using endpoints, you can map each connection to the appropriate instance or group of instances, depending on your use case. Custom endpoints let you connect to subsets of DB instances. The following endpoints are available in a Neptune DB cluster:

Neptune cluster endpoints

A cluster endpoint is an endpoint for a Neptune DB cluster that connects to the current primary DB instance for that DB cluster. Each Neptune DB cluster has a cluster endpoint and one primary DB instance.

The cluster endpoint provides failover support for read/write connections to the DB cluster. Use the cluster endpoint for all write operations on the DB cluster, including inserts, updates, deletes, and data definition language (DDL) changes. You can also use the cluster endpoint for read operations, such as queries.

If the current primary DB instance of a DB cluster fails, Neptune automatically fails over to a new primary DB instance. During a failover, the DB cluster continues to serve connection requests to the cluster endpoint from the new primary DB instance, with minimal interruption of service.

The following example illustrates a cluster endpoint for a Neptune DB cluster.

```
mydbcluster.cluster-123456789012.us-east-1.neptune.amazonaws.com:8182
```

Neptune reader endpoints

A reader endpoint is an endpoint for a Neptune DB cluster that connects to one of the available Neptune replicas for that DB cluster. Each Neptune DB cluster has a reader endpoint. If there is

more than one Neptune replica, the reader endpoint directs each connection request to one of the Neptune replicas.

The reader endpoint provides round-robin routing for read-only connections to the DB cluster. Use the reader endpoint for read operations, such as queries.

You can't use the reader endpoint for write operations unless you have a single-instance cluster (a cluster with no read-replicas). In that case and that case only, the reader can be used for write operations as well as read operations.

The reader endpoint round-robin routing works by changing the host that the DNS entry points to. Each time you resolve the DNS, you get a different IP, and connections are opened against those IPs. After a connection is established, all the requests for that connection are sent to the same host. The client must create a new connection and resolve the DNS record again to get a connection to potentially different read replica.

> **Note**
>
> WebSockets connections are often kept alive for long periods. To get different read replicas, do the following:
>
> - Ensure that your client resolves the DNS entry each time it connects.
> - Close the connection and reconnect.

Various client software might resolve DNS in different ways. For example, if your client resolves DNS and then uses the IP for every connection, it directs all requests to a single host.

DNS caching for clients or proxies resolves the DNS name to the same endpoint from the cache. This is a problem for both round robin routing and failover scenarios.

> **Note**
>
> Disable any DNS caching settings to force DNS resolution each time.

The DB cluster distributes connection requests to the reader endpoint among available Neptune replicas. If the DB cluster contains only a primary DB instance, the reader endpoint serves connection requests from the primary DB instance. If a Neptune replica is created for that DB

cluster, the reader endpoint continues to serve connection requests to the reader endpoint from the new Neptune replica, with minimal interruption in service.

The following example illustrates a reader endpoint for a Neptune DB cluster.

```
mydbcluster.cluster-ro-123456789012.us-east-1.neptune.amazonaws.com:8182
```

Neptune instance endpoints

An instance endpoint is an endpoint for a DB instance in a Neptune DB cluster that connects to that specific DB instance. Each DB instance in a DB cluster, regardless of instance type, has its own unique instance endpoint. So, there is one instance endpoint for the current primary DB instance of the DB cluster. There is also one instance endpoint for each of the Neptune replicas in the DB cluster.

The instance endpoint provides direct control over connections to the DB cluster, for scenarios where using the cluster endpoint or reader endpoint might not be appropriate. For example, your client application might require fine-grained load balancing based on workload type. In this case, you can configure multiple clients to connect to different Neptune replicas in a DB cluster to distribute read workloads.

The following example illustrates an instance endpoint for a DB instance in a Neptune DB cluster.

```
mydbinstance.123456789012.us-east-1.neptune.amazonaws.com:8182
```

Neptune custom endpoints

A custom endpoint for a Neptune cluster represents a set of DB instances that you choose. When you connect to the endpoint, Neptune chooses one of the instances in the group to handle the connection. You define which instances this endpoint refers to, and you decide what purpose the endpoint serves.

A Neptune DB cluster has no custom endpoints until you create one, and you can create up to five custom endpoints for each provisioned Neptune cluster.

The custom endpoint provides load-balanced database connections based on criteria other than the read-only or read/write capability of the DB instances. Because the connection can go to any DB instance associated with the endpoint, make sure that all the instances within that group share the same performance and memory capacity characteristics. When you use custom endpoints, you typically don't use the reader endpoint for that cluster.

This feature is intended for advanced users with specialized kinds of workloads where it isn't practical to keep all the Neptune Replicas in the cluster identical. With custom endpoints, you can adjust the capacity of the DB instances used with each connection.

For example, if you define several custom endpoints that connect to groups of instances with different instance classes, you can then direct users with different performance needs to the endpoints that best suit their use cases.

The following example illustrates a custom endpoint for a DB instance in a Neptune DB cluster:

```
myendpoint.cluster-custom-123456789012.us-east-1.neptune.amazonaws.com:8182
```

See Working with custom endpoints for more information.

Neptune endpoint considerations

Consider the following when working with Neptune endpoints:

- Before using an instance endpoint to connect to a specific DB instance in a DB cluster, consider using the cluster endpoint or reader endpoint for the DB cluster instead.

  The cluster endpoint and reader endpoint provide support for high-availability scenarios. If the primary DB instance of a DB cluster fails, Neptune automatically fails over to a new primary DB instance. It does so by either promoting an existing Neptune replica to a new primary DB instance or creating a new primary DB instance. If a failover occurs, you can use the cluster endpoint to reconnect to the newly promoted or created primary DB instance, or use the reader endpoint to reconnect to one of the other Neptune replicas in the DB cluster.

  If you don't take this approach, you can still make sure that you're connecting to the right DB instance in the DB cluster for the intended operation. To do so, you can manually or programmatically discover the resulting set of available DB instances in the DB cluster and confirm their instance types after failover, before using the instance endpoint of a specific DB instance.

  For more information about failovers, see Fault tolerance for a Neptune DB cluster.

- The reader endpoint only directs connections to available Neptune replicas in a Neptune DB cluster. It does not direct specific queries.

> **Important**
>
> Neptune does not load balance.

If you want to load balance queries to distribute the read workload for a DB cluster, you must manage that in your application. You must use instance endpoints to connect directly to Neptune replicas to balance the load.

- The reader endpoint round-robin routing works by changing the host that the DNS entry points to. The client must create a new connection and resolve the DNS record again to get a connection to a potentially new read replica.

- During a failover, the reader endpoint might direct connections to the new primary DB instance of a DB cluster for a short time, when a Neptune replica is promoted to the new primary DB instance.

Working with custom endpoints in Neptune

When you add a DB instance to a custom endpoint or remove it from a custom endpoint, any existing connections to that DB instance remain active.

You can define a list of DB instances to include in a custom endpoint (the *static* list), or one to exclude from the custom endpoint (the. *exclusion* list). You can use the inclusion/exclusion mechanism to subdivide the DB instances into groups and make sure that the custom endpoints covers all the DB instances in the cluster. Each custom endpoint can contain only one of these list types.

In the AWS Management Console, the choice is represented by the check box **Attach future instances added to this cluster**. When you keep check box clear, the custom endpoint uses a static list containing only the DB instances specified in the dialog. When you check the check box, the custom endpoint uses an exclusion list. In that case, the custom endpoint represents all DB instances in the cluster (including any that you add in the future) except the ones left unselected in the dialog.

Neptune doesn't change the DB instances specified in the static or exclusion lists when DB instances change roles between primary instance and Neptune Replica because of failover or promotion.

You can associate a DB instance with more than one custom endpoint. For example, suppose you add a new DB instance to a cluster. In that case the DB instance is added to all custom endpoints for which it is eligible. The static or exclusion list defined for it determines which DB instance can be added to it.

If an endpoint includes a static list of DB instances, newly added Neptune Replicas aren't added to it. Conversely, if the endpoint has an exclusion list, newly added Neptune Replicas are added to it provided that they aren't named in the exclusion list.

If a Neptune Replica becomes unavailable, it remains associated with its custom endpoints. This is true whether it is unhealthy, stopped, rebooting, or unavailable for another reason. However, as long as it remains unavailable you can't connect to it through any endpoint.

Because newly created Neptune clusters have no custom endpoints, you must create and manage them yourself. This is also true for Neptune clusters restored from snapshots, because custom endpoints are not included in the snapshot. You have create them again after restoring, and choose new endpoint names if the restored cluster is in the same region as the original one.

Creating a custom endpoint

Manage custom endpoints using the Neptune console. Do this by navigating to the details page for your Neptune cluster and use the controls in the **Custom Endpoints** section.

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. Navigate to the cluster detail page.
3. Choose the `Create custom endpoint` action in the **Endpoints** section.
4. Choose a name for the custom endpoint that is unique for your user ID and region. The name must be 63 characters or less in length and take the following form:

   ```
   endpointName.cluster-custom-customerDnsIdentifier.dnsSuffix
   ```

   Because custom endpoint names don't include the name of your cluster, you don't have to change those names if you rename a cluster. However, you can't reuse the same custom

endpoint name for more than one cluster in the same region. Give each custom endpoint a name that is unique across the clusters owned by your user ID within a particular region.

5. To choose a list of DB instances that remains the same even as the cluster expands, keep the check box **Attach future instances added to this cluster** clear. When that check box is checked, the custom endpoint dynamically adds any new instances as that are added to the cluster.

Viewing custom endpoints

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. Navigate to the cluster detail page of your DB cluster.
3. The **Endpoints** section only contains information about custom endpoints (details about the built-in endpoints are listed in the main **Details** section). To see details for a specific custom endpoint, select its name to bring up the detail page for that endpoint.

Editing a custom endpoint

You can edit the properties of a custom endpoint to change which DB instances are associated with it. You can also switch between a static list and an exclusion list.

You can't connect to or use a custom endpoint while the changes from an edit action are in progress. It might take some minutes after you make a change before the endpoint status returns to **Available** and you can connect again.

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. Navigate to the cluster detail page.
3. In the **Endpoints** section, choose the name of the custom endpoint you want to edit.
4. In the detail page for that endpoint, choose the **Edit** action.

Deleting a custom endpoint

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. Navigate to the cluster detail page.

3. In the **Endpoints** section, choose the name of the custom endpoint you want to delete.
4. In the detail page for that endpoint, choose the **Delete** action.

Neptune Lab Mode

You can use Amazon Neptune *lab mode* to enable new features that are in the current Neptune engine release, but that aren't yet ready for production use and aren't enabled by default. This lets you try out these features in your development and test environments.

Using Neptune Lab Mode

Use the neptune_lab_mode DB cluster parameter to enable or disable features. You do this by including `(feature name)`=enabled or `(feature name)`=disabled in the value of the neptune_lab_mode parameter in the DB Cluster Parameter group.

For example, in this engine release you might set the neptune_lab_mode parameter to `Streams=disabled, ReadWriteConflictDetection=enabled`.

For information about how to edit the DB cluster parameter group for your database, see Editing a Parameter Group. Note that you cannot edit the default DB cluster parameter group; if you are using the default group, you must create a new DB cluster parameter group before you can set the neptune_lab_mode parameter.

> **Note**
>
> When you make a change to a static DB cluster parameter such as neptune_lab_mode, you must re-start the primary (writer) instance of the cluster for the change to take effect. Before Release: 1.2.0.0 (2022-07-21), all the read-replicas in a DB cluster would then automatically be rebooted when the primary instance restarted.
> Beginning with Release: 1.2.0.0 (2022-07-21), restarting the primary instance does not cause any of the replicas to restart. This means that you must restart each instance separately to pick up a DB cluster parameter change (see Parameter groups).

> **Important**
>
> At present, if you supply the wrong lab-mode parameters or your request fails for another reason, you may not be notified of the failure. You should always verify that a lab-mode change request has succeeded by calling the status API as shown below:
>
> ```
> curl -G https://your-neptune-endpoint:port/status
> ```

The status results include lab-mode information which will show whether or not the changes you requested were made:

```
{
  "status":"healthy",
  "startTime":"Wed Dec 29 02:29:24 UTC 2021",
  "dbEngineVersion":"development",
  "role":"writer",
  "dfeQueryEngine":"viaQueryHint",
  "gremlin":{"version":"tinkerpop-3.5.2"},
  "sparql":{"version":"sparql-1.1"},
  "opencypher":{"version":"Neptune-9.0.20190305-1.0"},
  "labMode":{
    "ObjectIndex":"disabled",
    "ReadWriteConflictDetection":"enabled"
  },
  "features":{
    "LookupCache":{"status":"Available"},
    "ResultCache":{"status":"disabled"},
    "IAMAuthentication":"disabled",
    "Streams":"disabled",
    "AuditLog":"disabled"
  },
  "settings":{"clusterQueryTimeoutInMs":"120000"}
}
```

The following features are currently accessed using lab mode:

The OSGP index

Neptune can now maintain a fourth index, namely the OSGP index, which is useful for data sets having a large number of predicates (see Enabling an OSGP Index).

You can enable an OSGP index in a new, empty Neptune DB cluster by setting `ObjectIndex=enabled` in the `neptune_lab_mode` DB cluster parameter. An OSGP index can **only** be enabled in a new, empty DB cluster.

By default, the OSGP index is disabled.

> **Note**
>
> After setting the `neptune_lab_mode` DB cluster parameter so as to enable the OSGP index, you must restart the writer instance of the cluster for the change to take effect.

> **Warning**
>
> If you disable an enabled OSGP index by setting `ObjectIndex=disabled` and then later re-enable it after adding more data, the index will not build correctly. On-demand rebuilding of the index is not supported, so you should only enable the OSGP index when the database is empty.

Enabling dictionary garbage collection

Dictionary garbage collection can be enabled for property graph data when neptune-streams is not enabled via the `DictionaryGCMode` parameter. The concurrency can be controlled via the `DictionaryGCConcurrency` parameter. See Dictionary garbage collection for more details.

Formalized Transaction Semantics

Neptune has updated the formal semantics for concurrent transactions (see Transaction Semantics in Neptune).

Use `ReadWriteConflictDetection` as the name in the `neptune_lab_mode` parameter that enables or disables formalized transaction semantics.

By default, formalized transaction semantics are already enabled. If you want to revert to the earlier behavior, include `ReadWriteConflictDetection=disabled` in the value set for the DB Cluster `neptune_lab_mode` parameter.

Extended datetime support

Neptune has extended support for the datetime functionality. To enable datetime with extended formats, include `DatetimeMillisecond=enabled` in the value set for the DB Cluster `neptune_lab_mode` parameter.

StrictTimeoutValidation

> **Note**
>
> This feature is available starting in Neptune engine release 1.3.2.0.

Default value: enabled (disabled by default prior to Neptune engine release 1.4.0.0)

Allowed values: enabled/disabled

When this parameter is `enabled`, a per-query timeout value specified as a request option or a query hint cannot exceed the value set globally in the `neptune_query_timeout` parameter group setting. If the per-query timeout exceeds the global setting, Neptune throws an `InvalidParameterException`. In engine versions prior to 1.4.0.0, this parameter was `disabled` by default and had to be explicitly enabled.

This setting can be confirmed in a response on the `/status` endpoint when the value is `disabled`.

For more information, see Per-query timeouts.

AccurateQRCMemoryEstimation

> **Note**
>
> This feature is available starting in Neptune engine release 1.4.0.0.

Default value: disabled

Allowed values: enabled/disabled

Gremlin query result cache, when enabled, allows caching of query results on the database. By deafult the approximate estimate is used to determine the size of the result cached, with this lab mode param `AccurateQRCMemoryEstimation` enabled, the size estimation for cached results will use accurate size estimates instead of approximate. This labmode parameter is available starting from Neptune engine release version 1.4.0.0.

UndoLogPurgeConfig

> **Note**
>
> This feature is available starting in Neptune engine release 1.2.0.0.

Default value: default

Allowed values: default or aggressive

`UndoLogPurgeConfig` controls how aggressively Neptune purges undo logs. You enable it through the `neptune_lab_mode` DB cluster parameter by including `UndoLogPurgeConfig=aggressive` in the parameter value.

When set to `default`, Neptune uses the standard number of threads for undo log purging. When set to `aggressive`, Neptune uses an increased number of threads to expedite the cleanup of undo logs. The number of purge threads scales with the instance size, so pairing this setting with a larger writer instance increases the rate at which undo logs drain.

Undo log purging is performed only by the writer instance. Because `neptune_lab_mode` is a static parameter, you must reboot the writer instance for a change to `UndoLogPurgeConfig` to take effect. Beginning with Neptune engine release 1.2.0.0, restarting the writer no longer restarts the read replicas. If you want the setting active on a replica in case of a failover, you must reboot that replica separately.

To revert to standard purging, set `UndoLogPurgeConfig=default` in the `neptune_lab_mode` parameter and reboot the writer instance.

After the setting is active, you can track the `UndoLogListSize` and `NumUndoPagesPurged` CloudWatch metrics to observe the net drain rate under your production load. This is useful when you need to bring `UndoLogListSize` below the threshold required before a version upgrade (see Upgrading to engine version 1.2.0.0 or above from a version earlier than 1.2.0.0).

The Amazon Neptune alternative query engine (DFE)

Amazon Neptune has an alternative query engine known as the Data Flow Engine (DFE). DFE uses DB instance resources such as CPU cores, memory, and I/O more efficiently than the original Neptune engine.

> **Note**
>
> With large data sets, DFE might not run well on t3 instances.

DFE runs SPARQL, Gremlin, and openCypher queries, and supports a wide variety of plan types, including left-deep, bushy, and hybrid ones. Plan operators can invoke both compute operations, which run on a reserved set of compute cores, and I/O operations, each of which runs on its own thread in an I/O thread pool.

DFE uses pre-generated statistics about your Neptune graph data to make informed decisions about how to structure queries. See DFE statistics for information about how these statistics are generated.

The choice of plan type and the number of compute threads used is made automatically based on pre-generated statistics and on the resources that are available in the Neptune head node. The order of results is not predetermined for plans that have internal compute parallelism.

Controlling where Neptune DFE is used

By default, the neptune_dfe_query_engine instance parameter of an instance is set to `viaQueryHint`, which causes DFE to be used only for openCypher queries and for Gremlin and SPARQL queries that explicitly include the `useDFE` query hint set to `true`.

You can fully enable DFE so that it is used wherever possible by setting the `neptune_dfe_query_engine` instance parameter to `enabled`.

You can also disable DFE by including the `useDFE` query hint for a particular Gremlin query or SPARQL query. This query hint lets you prevent DFE from executing that particular query.

You can determine whether or not DFE is enabled in an instance using an Instance Status call, like this:

```
curl -G https://your-neptune-endpoint:port/status
```

The status response then specifies whether DFE is enabled or not:

```
{
  "status":"healthy",
  "startTime":"Wed Dec 29 02:29:24 UTC 2021",
  "dbEngineVersion":"development",
  "role":"writer",
  "dfeQueryEngine":"viaQueryHint",
  "gremlin":{"version":"tinkerpop-3.5.2"},
  "sparql":{"version":"sparql-1.1"},
  "opencypher":{"version":"Neptune-9.0.20190305-1.0"},
  "labMode":{
    "ObjectIndex":"disabled",
    "ReadWriteConflictDetection":"enabled"
  },
  "features":{
    "ResultCache":{"status":"disabled"},
    "IAMAuthentication":"disabled",
    "Streams":"disabled",
    "AuditLog":"disabled"
  },
  "settings":{"clusterQueryTimeoutInMs":"120000"}
}
```

The Gremlin `explain` and `profile` results tell you whether a query is being executed by DFE. See Information contained in a Gremlin `explain` report for `explain` and DFE `profile` reports for `profile`.

Similarly, SPARQL `explain` tells you whether a SPARQL query is being executed by DFE. See Example of SPARQL `explain` output when the DFE is enabled and `DFENode` operator for more details.

Query constructs supported by Neptune DFE

Neptune DFE supports SPARQL and Gremlin query constructs. For SPARQL, DFE handles the subset of conjunctive basic graph patterns. For Gremlin, starting with Neptune engine release 1.5.0.0, DFE natively supports most Gremlin steps. For a categorized list of supported and unsupported Gremlin steps, see Gremlin step coverage in DFE.

You can find out whether one of your queries is being executed in whole or in part by DFE as follows:

- In Gremlin, `explain` and `profile` results tell you what parts of a query are being executed by DFE, if any. See Information contained in a Gremlin `explain` report for `explain` and DFE `profile` reports for `profile`. Also, see Tuning Gremlin queries using `explain` and `profile`.

  Details about Neptune engine support for individual Gremlin steps are documented in Gremlin step support.
- Similarly, SPARQL `explain` tells you whether a SPARQL query is being executed by DFE. See Example of SPARQL `explain` output when the DFE is enabled and `DFENode` operator for more details.

Amazon Neptune JDBC connectivity

Amazon Neptune has released an open-source JDBC driver that supports openCypher, Gremlin, SQL-Gremlin, and SPARQL queries. JDBC connectivity makes it easy to connect to Neptune with business intelligence (BI) tools such as Tableau. There is no additional cost to using the JDBC driver with Neptune — you still pay only for the Neptune resources that are consumed.

The driver is compatible with JDBC 4.2, and requires at least Java 8. See the JDBC API documentation for information about how to use a JDBC driver.

The GitHub project, where you can file issues and open feature requests, contains detailed documentation for the driver:

JDBC Driver for Amazon Neptune

- Using SQL with the JDBC driver
- Using Gremlin with the JDBC Driver
- Using openCypher with the JDBC Driver
- Using SPARQL with the JDBC Driver

Getting started with the Neptune JDBC driver

To use the Neptune JDBC driver to connect to a Neptune instance, either the JDBC driver must be deployed on an Amazon EC2 instance in the same VPC as your Neptune DB cluster, or the instance must be available through an SSH tunnel or load balancer. An SSH tunnel can be set up in the driver internally, or it can be set up externally.

You can download the driver here. The driver comes packaged as a single JAR file with a name like `neptune-jdbc-1.0.0-all.jar`. To use it, place the JAR file in the `classpath` of your application. Or, if your application uses Maven or Gradle, you can use the appropriate Maven or Gradle commands to install the driver from the JAR.

The driver needs a JDBC connection URL to connect with Neptune, in a form like this:

```
jdbc:neptune:(connection
 type)://(host);property=value;property=value;...;property=value
```

The sections for each query language in the GitHub project describe the properties that you can set in the JDBC connection URL for that query language.

If the JAR file is in your application's `classpath`, no other configuration is necessary. You can connect the driver using the JDBC `DriverManager` interface and a Neptune connection string. For example, if your Neptune DB cluster is accessible through the endpoint `neptune-example.com` on port 8182, you would be able to connect with openCypher like this:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;

void example() {
    String url = "jdbc:neptune:opencypher://bolt://neptune-example:8182";

    Connection connection = DriverManager.getConnection(url);
    Statement statement = connection.createStatement();

    connection.close();
}
```

The documentation sections in the GitHub project for each query language describe how to construct the connection string when using that query language.

Using Tableau with the Neptune JDBC driver

To use Tableau with the Neptune JDBC driver, start by downloading and installing the most recent version of Tableau Desktop. Download the JAR file for the Neptune JDBC driver, and also the Neptune Tableau connector file (a `.taco` file).

To connect to Tableau for Neptune on a Mac

1. Place the Neptune JDBC driver JAR file in the `/Users/`*`(your user name)`*`/Library/Tableau/Drivers` folder.
2. Place the Neptune Tableau connector `.taco` file in the `/Users/`*`(your user name)`*`/Documents/My Tableau Repository/Connectors` folder.
3. If you have IAM authentication enabled, set up the environment for it. Note that environment variables set in `.zprofile/`, `.zshenv/`, `.bash_profile`, and so forth, will not work. The environment variables must be set so that they can be loaded by a GUI application.

   One way to set your credentials is by placing your access key and secret key in the `/Users/`*`(your user name)`*`/.aws/credentials` file.

An easy way to set the service region is to open a terminal and enter the following command, using your application's region (for example, `us-east-1`):

```
launchctl setenv SERVICE_REGION region name
```

There are other ways to set environment variables that persist after a restart, but whatever technique you use must set variables that are accessible to a GUI application.

4. To get environment variables to load into a GUI on the Mac, enter this command on a terminal:

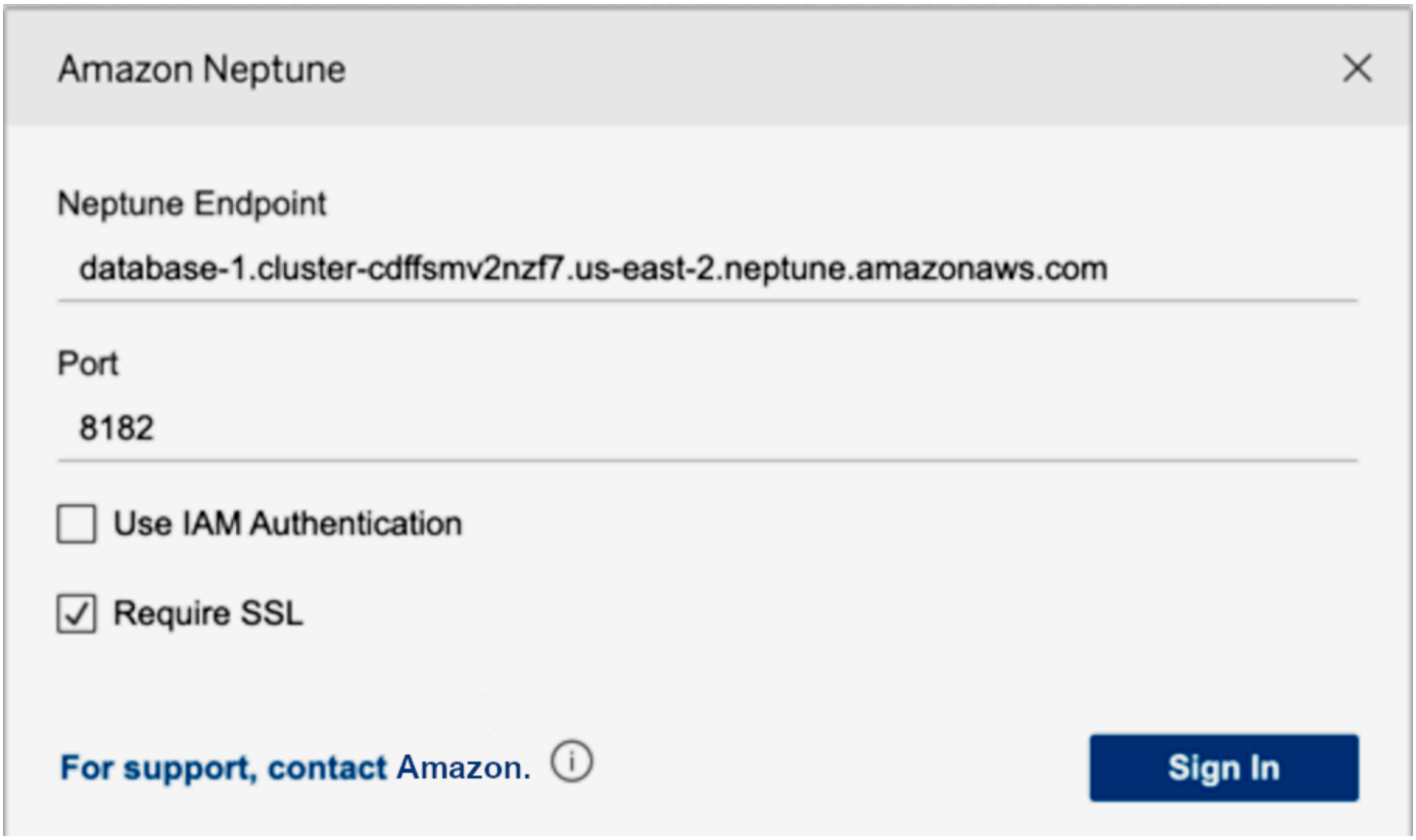
```
/Applications/Tableau/Desktop/2021.1.app/Contents/MacOS/Tableau
```

To connect to Tableau for Neptune on a Windows machine

1. Place the Neptune JDBC driver JAR file in the `C:\Program Files\Tableau\Drivers` folder.
2. Place the Neptune Tableau connector `.taco` file in the `C:\Users\(your user name)\Documents\My Tableau Repository\Connectors` folder.
3. If you have IAM authentication enabled, set up the environment for it.

   This can be as simple as setting user `ACCESS_KEY`, `SECRET_KEY`, and `SERVICE_REGION` environment variables.

With Tableau open, select **More** on the left side of the window. If the Tableau connector file is properly located, you can select **Amazon Neptune by AWS** in the list that appears:



You should not have to edit the port, or add any connection options. Enter your Neptune endpoint and set your IAM and SSL configuration (you must enable SSL if you are using IAM).

When you select **Sign In**, it may take more than 30 seconds to connect if your graph is large. Tableau is collecting vertex and edge tables and join vertices on edges, as well as creating visualizations.

Troubleshooting a JDBC driver connection

If the driver fails to connect to the server, use the `isValid` function of the JDBC `Connection` object to check whether the connection is valid. If the function returns `false`, meaning that the connection is invalid, check that the endpoint being connected to is correct and that you are in the VPC of your Neptune DB cluster or that you have a valid SSH tunnel to the cluster.

If you get a `No suitable driver found for (connection string)` response from the `DriverManager.getConnection` call, there is likely an issue at the beginning of your connection string. Make sure that your connection string starts like this:

```
jdbc:neptune:opencypher://...
```

To gather more information about the connection, you can add a `LogLevel` to your connection string like this:

```
jdbc:neptune:opencypher://(JDBC URL):(port);logLevel=trace
```

Alternatively, you can add `properties.put("logLevel", "trace")` in your input properties to log trace information.

Amazon Neptune engine updates

Amazon Neptune releases engine updates regularly. You can determine which engine release version you currently have installed using the instance-status API.

Engine releases are listed at Engine releases for Amazon Neptune, and patches are listed at Latest Updates.

You can find more information about how the updates are released and how to upgrade the Neptune engine in your database at Cluster maintenance. For example, version numbering is explained in Engine version numbers.

Exception Handling and Retries

Building robust applications on Neptune often means preparing for the unexpected, especially when it comes to handling errors returned by the database. One of the most common responses to server-side exceptions is to retry the failed operation. While retry logic is essential for resilient systems, you need to recognize that not all errors should be treated the same way. Rather than relying on generic retry behaviors, a thoughtful approach can help you build more reliable and efficient applications.

Why retry logic matters

Retry logic is a critical component of any distributed application. Transient issues such as network instability, temporary resource constraints, or concurrent modification conflicts can cause operations to fail. In many cases, these failures don't indicate a permanent problem and can be resolved by waiting and trying again. Implementing a solid retry strategy acknowledges the reality of imperfect environments in distributed systems, ensuring stronger reliability and continuity with less need for manual intervention.

The risks of indiscriminate retries

Retrying every error by default can lead to several unintended consequences:

- **Increased contention** – When operations that fail due to high concurrency are retried repeatedly, the overall contention can get worse. This might result in a cycle of failed transactions and degraded performance.
- **Resource exhaustion** – Indiscriminate retries can consume additional system resources, both on the client and server side. This can potentially lead to throttling or even service degradation.
- **Increased latency for clients** – Excessive retries can cause significant delays for client applications, especially if each retry involves waiting periods. This can negatively impact user experience and downstream processes.

Developing a practical retry strategy

To build a resilient and efficient application, develop a retry strategy that's tailored to the specific error conditions your application might encounter. Here are some considerations to guide your approach:

- **Identify retryable errors** – Not all exceptions should be retried. For example, syntax errors, authentication failures, or invalid queries should not trigger a retry. Neptune provides error codes and general recommendations for which errors are safe to retry, but you need to implement the logic that fits your use case.
- **Implement exponential backoff** – For transient errors, use an exponential backoff strategy to progressively increase the wait time between retries. This helps alleviate contention and reduces the risk of cascading failures.
- **Consider initial pause length** – Performing the first retry too quickly might just end with the same error if the server hasn't been given enough time to release resources that the query needs to succeed. A longer pause in the right situations could reduce wasted requests and server pressure.
- **Add jitter to backoff** – While exponential backoff is effective, it can still lead to synchronized retry storms if many clients fail at the same time and then retry together. Adding jitter, a small random variation to the backoff delay, helps spread out retry attempts thereby reducing the chance of all clients retrying simultaneously and causing another spike in load.
- **Limit retry attempts** – Set a reasonable maximum number of retries to prevent infinite loops and resource exhaustion.
- **Monitor and adjust** – Continuously monitor your application's error rate and adjust your retry strategy as needed. If you notice a high number of retries for a particular operation, consider whether the operation can be optimized or serialized.

For a deeper discussion of exponential backoff and jitter as a general cloud design pattern, see Retry with backoff in AWS Prescriptive Guidance.

Example scenarios

The right retry strategy depends on the nature of the failure, the workload, and the error patterns you observe. The following table summarizes some common failure scenarios and how the retry strategy considerations apply to each. Explanatory paragraphs follow for additional context.

Scenario	Retryable?	Backoff & Jitter	Initial Pause	Retry Limit	Monitor & Adjust
Occasional CME	Yes	Short backoff, add jitter	Short (for example, 100ms)	High	Watch for rising CME Rates

Scenario	Retryable?	Backoff & Jitter	Initial Pause	Retry Limit	Monitor & Adjust
on Short Queries					
Frequent CME on Longer-Running Queries	Yes	Longer backoff, add jitter	Longer (for example, 2s)	Moderate	Investigate and reduce contention
Memory Limits on Expensive Queries	Yes	Long backoff	Long (for example, 5-10s)	Low	Optimize query, alert if persistent
Timeout on Moderate Queries	Maybe	Moderate backoff, add jitter	Moderate (for example, 1s)	Low to Moderate	Assess server load and query design

Scenario 1: Occasional CME on short queries

For a workload where `ConcurrentModificationException` appears infrequently during short, simple updates, these errors are typically transient and safe to retry. Use a short initial pause (for example, 100 milliseconds) before the first retry. This time allows any brief lock to clear. Combine this with a short exponential backoff and jitter to avoid synchronized retries. Since the cost of retrying is low, a higher retry limit is reasonable. Still, monitor the CME rate to catch any trend toward increased contention in your data.

Scenario 2: Frequent CME on long-running queries

If your application sees frequent CMEs on long-running queries, this suggests more severe contention. In this case, start with a longer initial pause (for example, 2 seconds), to give the current query holding the lock enough time to complete. Use a longer exponential backoff and add jitter. Limit the number of retries to avoid excessive delays and resource usage. If contention persists, review your workload for patterns and consider serializing updates or reducing concurrency to address the root cause.

Scenario 3: Memory limits on expensive queries

When memory-based errors occur during a known resource-intensive query, retries can make sense, but only after a long initial pause (for example, 5 to 10 seconds or more) to allow the server to release resources. Use a long backoff strategy and set a low retry limit, since repeated failures are unlikely to resolve without changes to the query or workload. Persistent errors should trigger alerts and prompt a review of query complexity and resource usage.

Scenario 4: Timeout on moderate queries

A timeout on a moderately expensive query is a more ambiguous case. Sometimes, a retry might succeed if the timeout was due to a temporary spike in server load or network conditions. Start with a moderate initial pause (for example, 1 second) to give the system a chance to recover. Apply a moderate backoff and add jitter to avoid synchronized retries. Keep the retry limit low to moderate, since repeated timeouts might indicate a deeper issue with the query or the server's capacity. Monitor for patterns: if timeouts become frequent, assess whether the query needs optimization or if the Neptune cluster is under-provisioned.

Monitoring and observability

Monitoring is a critical part of any retry strategy. Effective observability helps you understand how well your retry logic is working and provides early signals when something in your workload or cluster configuration needs attention.

MainRequestQueuePendingRequests

This CloudWatch metric tracks the number of requests waiting in Neptune's input queue. A rising value indicates that queries are backing up, which can be a sign of excessive contention, under-provisioned resources, or retry storms. Monitoring this metric helps you spot when your retry strategy is causing or compounding queuing issues, and can prompt you to adjust your approach before failures escalate.

Other CloudWatch metrics

Other Neptune metrics like `CPUUtilization`, `TotalRequestsPerSecond`, and query latency provide additional context. For example, high CPU and I/O combined with growing queue lengths might indicate that your cluster is overloaded or that queries are too large or too frequent. CloudWatch alarms can be set on these metrics to alert you to abnormal behavior and help you correlate spikes in errors or retries with underlying resource constraints.

Neptune Status and Query APIs

The Neptune Status API for Gremlin and its analogous APIs for OpenCypher and SPARQL give a real-time view of the queries accepted and running on the cluster which is useful for diagnosing bottlenecks or understanding the impact of retry logic in real time.

By combining these monitoring tools, you can:

- Detect when retries are contributing to queuing and performance degradation.
- Identify when to scale your Neptune cluster or optimize queries.
- Validate that your retry strategy is resolving transient failures without masking deeper issues.
- Receive early warnings about emerging contention or resource exhaustion.

Proactive monitoring and alerting are essential for maintaining a healthy Neptune deployment, especially as your application's concurrency and complexity grow.

Securing your Amazon Neptune database

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS compliance programs. To learn about the compliance programs that apply to Amazon Neptune, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Neptune. The following topics show you how to configure Neptune to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Neptune resources.

Topics

- Amazon Neptune operating system upgrades
- Protecting data in your Amazon Neptune database
- Authenticating your Amazon Neptune database with AWS Identity and Access Management
- Enabling IAM database authentication in Amazon Neptune
- Connecting to your Amazon Neptune database using AWS Identity and Access Management authentication
- Managing access to Amazon Neptune databases using IAM policies
- Using service-linked roles for Amazon Neptune
- Using temporary credentials to connect to Amazon Neptune
- Logging and monitoring usage and performance in Amazon Neptune

- Amazon Neptune and interface VPC endpoints (AWS PrivateLink)
- Compliance considerations for Amazon Neptune
- Building resilient and disaster-tolerant Amazon Neptune deployments

Amazon Neptune operating system upgrades

Amazon Neptune ensures continuous improvements in database performance, security, and stability through regular OS upgrades. These upgrades are applied to both Neptune Databases and Neptune Analytics, following a structured update process. Amazon Neptune releases OS upgrades at least once per month.

Neptune Database OS updates that do not require downtime are automatically applied during the maintenance window. Certain OS updates (eg: kernel version upgrade) requires an instance restart. These updates are optional and don't have a set date to apply them. However, if you don't apply these updates, they may eventually become required and automatically applied during your instance's maintenance window.

Neptune Analytics - Neptune Analytics OS upgrades are seamless and require no customer action. These updates are automatically applied in the background without downtime.

To maintain security and compliance, we recommend that you apply all updates made available by Amazon Neptune routinely during your maintenance window. Staying current on all optional and mandatory updates helps incorporate critical security patches and ensures alignment with various compliance obligations. Outdated OS versions may result in non-compliance with regulatory requirements.

Minimizing downtime for OS upgrades that require a restart

For OS upgrades that require a restart, we recommend that you update the reader instances in a cluster first, then the writer instance to maximize the availability of your cluster. We don't recommend updating reader and writer instances at the same time, because you could incur longer downtime in the event of a failover.

Applying OS upgrades to your Neptune DB instance

Neptune DB instances occasionally require operating system updates. Amazon Neptune upgrades the operating system to a newer version to improve database performance and customers overall

security posture. Typically, the updates take about 10 minutes. Operating system updates don't change the DB engine version or DB instance class of a DB instance.

To be notified when a new optional update becomes available, you can subscribe to `RDS-EVENT-0230` in the security patching event category. For information about subscribing to Amazon Neptune events, see Subscribing to Neptune event notification.

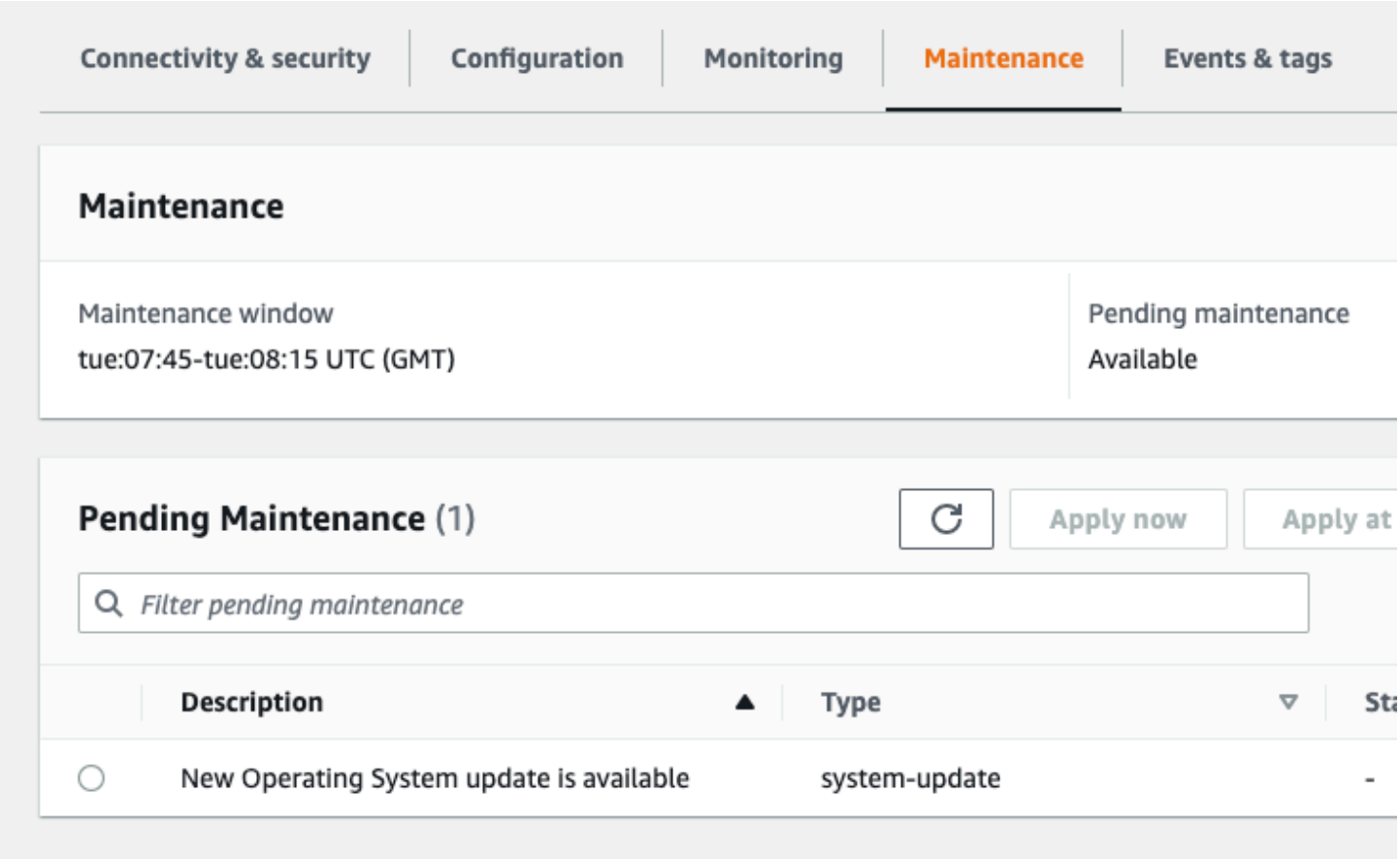
> **Important**
>
> Your Amazon Neptune DB instance will be taken offline during the operating system upgrade. You can minimize cluster downtime by having a multi-instance cluster. If you do not have a multi-instance cluster then you can choose to temporarily create one by adding secondary instance(s) to perform this maintenance, then deleting the additional reader instance(s) once the maintenance is completed (regular charges for the secondary instance will apply).

You can use the AWS Management Console or the AWS CLI to determine whether an update is available.

Using the AWS Management Console

To determine whether an update is available using the AWS Management Console:

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Clusters**, and then select the instance.
3. Choose **Maintenance**.
4. In the **Pending Maintenance** section, find the operating system update.



You can select the operating system update and click **Apply now** or **Apply at next maintenance window** in the **Pending Maintenance** section. If the maintenance value is **next window**, defer the maintenance items by choosing **Defer upgrade**. You can't defer a maintenance action if it has already started.

Alternatively, you can choose the instance from a list of clusters by clicking on **Clusters** in the navigation pane and select **Apply now** or **Apply at next maintenance window** from the **Actions** menu.

Using the AWS CLI

To determine whether an update is available using the AWS CLI, call the `describe-pending-maintenance-actions` command:

```
aws neptune describe-pending-maintenance-actions
```

```
{
    "ResourceIdentifier": "arn:aws:rds:us-east-1:123456789012:db:myneptune",
    "PendingMaintenanceActionDetails": [
```

```
        {
            "Action": "system-update",
            "Description": "New Operating System update is available"
        }
    ]
}
```

To apply the Operating system updates, call the `apply-pending-maintenance-action` command:

```
aws neptune apply-pending-maintenance-action \
    --apply-action system-update \
    --resource-identifier (ARN of your DB instance) \
    --opt-in-type immediate
```

Protecting data in your Amazon Neptune database

The AWS shared responsibility model applies to data protection in Amazon Neptune. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see Data Privacy FAQ. For information about data protection in Europe, see the General Data Protection Regulation (GDPR) Center.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.

- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Neptune or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

> **Important**
>
> TLS 1.3 is only supported for Neptune engine version 1.3.2.0 and up.

You use AWS published API calls to manage Neptune through the network. Clients must support Transport Layer Security (TLS) 1.2 or later using strong cipher suites, as described in Encrypting connects to Neptune databases. Most modern systems such as Java 7 and later support these modes.

The following sections describe further how Neptune data is protected.

Topics

- Securing your Amazon Neptune database with Amazon VPC
- Encrypting connections to your Amazon Neptune database with SSL/HTTPS
- Encrypting Neptune resources at rest

Securing your Amazon Neptune database with Amazon VPC

An Amazon Neptune DB cluster can *only* be created in an Amazon Virtual Private Cloud (Amazon VPC), and its endpoints are accessible within that VPC, usually from an Amazon Elastic Compute Cloud (Amazon EC2) instance running in that VPC. Alternatively, it can be accessed using a public endpoint. For more information on public endpoints, see Neptune public endpoints.

You can secure your Neptune data by limiting access to the VPC where your Neptune DB cluster is located, as described in Connecting to an Amazon Neptune cluster.

Encrypting connections to your Amazon Neptune database with SSL/HTTPS

Amazon Neptune only allows Secure Sockets Layer (SSL) connections through HTTPS to any instance or cluster endpoint.

Neptune requires at least TLS version 1.2, using the following strong cipher suites:

- `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`
- `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384`
- `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`
- `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`

Starting with Neptune engine version 1.3.2.0, Neptune supports TLS version 1.3 using the following cipher suites:

- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384

Even where HTTP connections are allowed in earlier engine versions, any DB cluster that uses a new DB cluster parameter group is required to use SSL by default. *To protect your data, Neptune endpoints in engine version `1.0.4.0` and above only support HTTPS requests.* See Using the HTTP REST endpoint to connect to a Neptune DB instance for more information.

Neptune automatically provides SSL certificates for your Neptune DB instances. You don't need to request any certificates. The certificates are provided when you create a new instance.

Neptune assigns a single wildcard SSL certificate to the instances in your account for each AWS Region. The certificate provides entries for the cluster endpoints, cluster read-only endpoints, and instance endpoints.

Certificate Details

The following entries are included in the provided certificate:

- Cluster endpoint — `*.cluster-`*`a1b2c3d4wxyz`*`.`*`region`*`.neptune.amazonaws.com`
- Read-only endpoint — `*.cluster-ro-`*`a1b2c3d4wxyz`*`.`*`region`*`.neptune.amazonaws.com`
- Instance endpoints — `*.`*`a1b2c3d4wxyz`*`.`*`region`*`.neptune.amazonaws.com`

Only the entries listed here are supported.

Proxy Connections

The certificates support only the hostnames that are listed in the previous section.

If you are using a load balancer or a proxy server (such as HAProxy), you must use SSL termination and have your own SSL certificate on the proxy server.

SSL passthrough doesn't work because the provided SSL certificates don't match the proxy server hostname.

Root CA Certificates

The certificates for Neptune instances are normally validated using the local trust store of the operating system or SDK (such as the Java SDK).

If you need to provide a root certificate manually, you can download the Amazon Root CA certificate in PEM format from the Amazon Trust Services Policy Repository.

More Information

For more information about connecting to Neptune endpoints with SSL, see the section called "Installing the Gremlin console" and the section called "HTTP REST".

Encrypting Neptune resources at rest

Data-at-rest encryption is the AWS recommendation. For more information, see Data-at-Rest and Data-in-Transit encryption. Encryption is enforced in the AWS Console when you create a new Neptune DB Cluster or a new Neptune Global DB. This provides an additional layer of data protection. It secures your data from unauthorized access to the underlying storage and helps fulfill compliance requirements for data-at-rest encryption.

To manage the keys used for encrypting and decrypting your Neptune resources, you use AWS Key Management Service (AWS KMS). AWS KMS combines secure, highly available hardware and software to provide a key management system scaled for the cloud. Using AWS KMS, you can create encryption keys and define the policies that control how these keys can be used. AWS KMS supports AWS CloudTrail, so you can audit key usage to verify that keys are being used appropriately.

At rest, all related logs, backups, and snapshots are encrypted for any encrypted Neptune DB clusters. The Neptune encryption does not apply to logs exported to Amazon CloudWatch.

Encryption of Neptune resources

When you create a Neptune DB Cluster or a Neptune Global DB, you can supply the AWS KMS key identifier for your encryption key. If you don't specify a AWS KMS key identifier, Neptune uses your default Amazon RDS encryption key (`aws/rds`) in the Region. AWS KMS creates a default encryption key for each Region in your AWS account. For a Neptune Global cluster, there will be as many AWS KMS keys as Regions in it.

After you create a Neptune resource, you can't change the encryption key for that resource. So, be sure to determine your encryption key requirements before you create your Neptune resource. If a different AWS KMS key is required, you can use a snapshot of the existing Neptune DB Cluster to create a new one with a different AWS KMS key (see Restoring from a DB Cluster Snapshot).

You can use the Amazon Resource Name (ARN) of a key from another account to encrypt a Neptune resource. If you create a Neptune resource with the same AWS account that owns the AWS KMS encryption key, the AWS KMS key ID that you pass can be the AWS KMS key alias instead of the key's ARN.

> **Important**
>
> If Neptune loses access to the encryption key for a Neptune DB Cluster - for example, when Neptune access to a key is revoked - the encrypted cluster is placed into a terminal state and can only be restored from a backup. We strongly recommend that you always enable backups for encrypted Neptune DB Clusters to guard against the loss of encrypted data in your databases.

Key permissions needed when enabling encryption

The IAM user or role creating a Neptune DB Cluster must have at least the following permissions for the KMS key:

- `"kms:Encrypt"`
- `"kms:Decrypt"`
- `"kms:GenerateDataKey"`
- `"kms:ReEncryptTo"`
- `"kms:GenerateDataKeyWithoutPlaintext"`

- "kms:CreateGrant"
- "kms:ReEncryptFrom"
- "kms:DescribeKey"

Here is an example (for us-east-1 region) of a key policy that includes the necessary permissions:

JSON

```
{
  "Version":"2012-10-17",
  "Id": "key-consolepolicy-3",
  "Statement": [
    {
      "Sid": "Enable Permissions for root principal",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws-eusc:iam::123456789012:root"
      },
      "Action": "kms:*",
      "Resource": "*"
    },
    {
      "Sid": "Allow use of the key for Neptune",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws-eusc:iam::123456789012:role/NeptuneFullAccess"
      },
      "Action": [
        "kms:Encrypt",
        "kms:Decrypt",
        "kms:GenerateDataKey",
        "kms:ReEncryptTo",
        "kms:GenerateDataKeyWithoutPlaintext",
        "kms:CreateGrant",
        "kms:ReEncryptFrom",
        "kms:DescribeKey"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "kms:ViaService": "rds.eusc-de-east-1.amazonaws.com"
```

```
        }
      }
    },
    {
      "Sid": "Deny use of the key for non Neptune",
      "Effect": "Deny",
      "Principal": {
        "AWS": "arn:aws-eusc:iam::123456789012:role/NeptuneFullAccess"
      },
      "Action": [
        "kms:*"
      ],
      "Resource": "*",
      "Condition": {
        "StringNotEquals": {
          "kms:ViaService": "rds.eusc-de-east-1.amazonaws.com"
        }
      }
    }
  ]
}
```

- The first statement in this policy is optional. It gives access to the user's root principal.
- The second statement provides access to all the required AWS KMS APIs for this role, scoped down to the RDS Service Principal.
- The third statement tightens the security more by enforcing that this key is not usable by this role for any other AWS service.

You could also scope `createGrant` permissions down further by adding:

```
"Condition": {
  "Bool": {
    "kms:GrantIsForAWSResource": true
  }
}
```

Limitations of Neptune Encryption

The following limitations exist for Neptune Encryption:

- You cannot convert an unencrypted Neptune DB Cluster to an encrypted one. You can only enable encryption for a Neptune DB Cluster when it is created. However, you can restore an unencrypted Neptune DB Cluster snapshot to an encrypted Neptune DB Cluster. To do this, specify a KMS encryption key when you restore from the unencrypted Neptune DB Cluster snapshot.
- For compatibility reasons, it is still possible to create an unencrypted Neptune DB Cluster via the CLI and AWS SDKs. The console only allows creation of encrypted Neptune DB Clusters.
- You cannot mix encrypted and unencrypted Neptune DB Clusters in the same Neptune Global DB. Either all the clusters are encrypted or all the clusters are unencrypted. This is enforced in the Neptune Global DB configuration.

Authenticating your Amazon Neptune database with AWS Identity and Access Management

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Neptune resources. IAM is an AWS service that you can use with no additional charge.

You can use AWS Identity and Access Management (IAM) to authenticate to your Neptune DB instance or DB cluster. When IAM database authentication is enabled, each request must be signed using AWS Signature Version 4.

AWS Signature Version 4 adds authentication information to AWS requests. For security, all requests to Neptune DB clusters with IAM authentication enabled must be signed with an access key. This key consists of an access key ID and secret access key. The authentication is managed externally using IAM policies.

Neptune authenticates on connection, and for WebSockets connections it verifies the permissions periodically to ensure that the user still has access.

> **Note**
>
> - Revoking, deleting, or rotating of credentials associated with the IAM user is not recommended because it does not terminate any connections that are already open.

- There are limits on the number of concurrent WebSocket connections per database instance, and on how long a connection can remain open. For more information, see WebSockets Limits.

IAM Use Depends on Your Role

How you use AWS Identity and Access Management (IAM) differs, depending on the work you do in Neptune.

Service user – If you use the Neptune service to do your job, then your administrator provides you with the credentials and permissions that you need for using the Neptune data plane. As you need more access to do your work, understanding how data access is managed can help you request the right permissions from your administrator.

Service administrator – If you're in charge of Neptune resources at your company, you probably have access to Neptune management actions, which correspond to the Neptune managment API. It may also be your job to determine which Neptune data-access actions and resources service users need in order to do their jobs. An IAM administrator can then apply IAM policies to change the permissions of your service users.

IAM administrator – If you're an IAM administrator, you will need to write IAM policies to manage both management and data access to Neptune. To view example Neptune identity-based policies that you can use, see Using different kinds of IAM policies for controlling access to Neptune.

Authenticating with Identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity called the AWS account *root user* that has complete access to all AWS services and resources. We strongly recommend that you don't use the root user for everyday tasks. For tasks that require root user credentials, see Tasks that require root user credentials in the *IAM User Guide*.

IAM Users and Groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM Roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Enabling IAM database authentication in Amazon Neptune

By default, IAM database authentication is disabled when you create an Amazon Neptune DB cluster. You can enable IAM database authentication (or disable it again) using the AWS Management Console.

To create a new Neptune DB cluster with IAM authentication by using the console, follow the instructions for creating a Neptune DB cluster in Launching a Neptune DB cluster using the AWS Management Console.

During the creation process, under **Additional settings**, choose **Turn on IAM Authentication**.

To enable or disable IAM authentication for an existing DB instance or cluster

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Clusters**.
3. Choose the Neptune DB cluster that you want to modify, and then choose **Modify**.
4. Under **Additional settings**, for **IAM DB Authentication**, choose either **Turn on IAM Authentication** or **Turn off IAM Authentication** (to disable). Then choose **Next**.
5. Under **Scheduling of modifications**, choose when to apply the change, and then choose **Submit**.

> **Important**
>
> Enabling or disabling IAM database authentication causes the DB engine to restart. This restart terminates all existing connections to the cluster and causes a brief interruption in availability. Plan to make this change during a maintenance window or at a time when the impact of a connection interruption is minimal.

Connecting to your Amazon Neptune database using AWS Identity and Access Management authentication

Amazon Neptune resources that have IAM DB authentication enabled require all HTTP requests to be signed using AWS Signature Version 4. For general information about signing requests with AWS Signature Version 4, see Signing AWS API requests.

AWS Signature Version 4 is the process to add authentication information to AWS requests. For security, most requests to AWS must be signed with an access key, which consists of an access key ID and secret access key.

> **Note**
>
> If you are using temporary credentials, they expire after a specified interval, *including the session token*.

You must update your session token when you request new credentials. For more information, see Using Temporary Security Credentials to Request Access to AWS Resources.

Important

Accessing Neptune with IAM-based authentication requires that you create HTTP requests and sign the requests yourself.

How Signature Version 4 Works

1. You create a canonical request.
2. You use the canonical request and some other information to create a string-to-sign.
3. You use your AWS secret access key to derive a signing key, and then use that signing key and the string-to-sign to create a signature.
4. You add the resulting signature to the HTTP request in a header or as a query string parameter.

When Neptune receives the request, it performs the same steps that you did to calculate the signature. Neptune then compares the calculated signature to the one you sent with the request. If the signatures match, the request is processed. If the signatures don't match, the request is denied.

For general information about signing requests with AWS Signature Version 4, see Signature Version 4 Signing Process in the *AWS General Reference*.

The following sections contain examples that show how to send signed requests to the Gremlin and SPARQL endpoints of a Neptune DB instance with IAM authentication enabled.

Topics

- Prerequisites for connecting Amazon Neptune databases using IAM authentication
- Connecting to Amazon Neptune databases using IAM authentication from the command line
- Connecting to Amazon Neptune databases using IAM authentication with Gremlin console
- Connecting to Amazon Neptune databases using IAM with Gremlin Java
- Connecting to Amazon Neptune databases using IAM authentication with Java and SPARQL

- Connecting to Amazon Neptune databases using IAM authentication with SPARQL and Node.js
- Connecting to Amazon Neptune databases using IAM authentication with Python
- Connecting to Amazon Neptune databases using IAM authentication with Gremlin Python
- Connecting to Amazon Neptune databases using IAM authentication with Gremlin JavaScript
- Connecting to Amazon Neptune databases using IAM authentication with Gremlin Go
- Connecting to Amazon Neptune databases using IAM authentication with Gremlin .NET

Prerequisites for connecting Amazon Neptune databases using IAM authentication

The following are instructions for installing Apache Maven and Java 8 on an Amazon EC2 instance. These are required for the Amazon Neptune Signature Version 4 authentication samples.

To Install Apache Maven and Java 8 on your EC2 instance

1. Connect to your Amazon EC2 instance with an SSH client.
2. Install Apache Maven on your EC2 instance. If using Amazon Linux 2023 (preferred), use:

   ```
   sudo dnf update -y
   sudo dnf install maven -y
   ```

   If using Amazon Linux 2, download the latest binary from https://maven.apache.org/download.cgi:

   ```
   sudo yum remove maven -y
   wget https://dlcdn.apache.org/maven/maven-3/ <version>/binaries/apache-maven-
   <version>-bin.tar.gz
   sudo tar -xzf apache-maven-<version>-bin.tar.gz -C /opt/
   sudo ln -sf /opt/apache-maven-<version> /opt/maven
   echo 'export MAVEN_HOME=/opt/maven' >> ~/.bashrc
   echo 'export PATH=$MAVEN_HOME/bin:$PATH' >> ~/.bashrc
   source ~/.bashrc
   ```
3. The Gremlin libraries require Java 8. Enter the following to install Java 8 on your EC2 instance.

   ```
   sudo yum install java-1.8.0-devel
   ```
4. Enter the following to set Java 8 as the default runtime on your EC2 instance.

```
sudo /usr/sbin/alternatives --config java
```

When prompted, enter the number for Java 8.

5. Enter the following to set Java 8 as the default compiler on your EC2 instance.

```
sudo /usr/sbin/alternatives --config javac
```

When prompted, enter the number for Java 8.

Connecting to Amazon Neptune databases using IAM authentication from the command line

Having a command-line tool for submitting queries to your Neptune DB cluster is very handy, as illustrated in many of the examples in this documentation. The curl tool is an excellent option for communicating with Neptune endpoints if IAM authentication is not enabled.

However, to keep your data secure it is best to enable IAM authentication.

When IAM authentication is enabled, every request must be signed using Signature Version 4 (Sig4). The third-party awscurl command-line tool uses the same syntax as `curl`, and can sign queries using Sig4 signing. The Using `awscurl` section below explains how to use `awscurl` securely with temporary credentials.

Setting up a command-line tool to use HTTPS

Neptune requires that all connections use HTTPS. Any command line tool like `curl` or `awscurl` needs access to appropriate certificates in order to use HTTPS. As long as `curl` or `awscurl` can locate the appropriate certificates, they handle HTTPS connections just like HTTP connections, without needing extra parameters. Examples in this documentation are based on that scenario.

To learn how to obtain such certificates and how to format them properly into a certificate authority (CA) certificate store that `curl` can use, see SSL Certificate Verification in the `curl` documentation.

You can then specify the location of this CA certificate store using the `CURL_CA_BUNDLE` environment variable. On Windows, `curl` automatically looks for it in a file named `curl-ca-`

bundle.crt. It looks first in the same directory as curl.exe and then elsewhere on the path. For more information, see SSL Certificate Verification.

Using awscurl with temporary credentials to securely connect to a DB cluster with IAM authentication enabled

The awscurl tool uses the same syntax as curl, but needs additional information as well:

- **--access_key** – A valid access key. If not supplied using this parameter, it must be supplied in the AWS_ACCESS_KEY_ID enviroment variable, or in a configuration file.
- **--secret_key** – A valid secret key corresponding to the access key. If not supplied using this parameter, it must be supplied in the AWS_SECRET_ACCESS_KEY enviroment variable, or in a configuration file.
- **--security_token** – A valid session token. If not supplied using this parameter, it must be supplied in the AWS_SECURITY_TOKEN enviroment variable, or in a configuration file.

In the past, it was common practice to use persistent credentials with awscurl, such as IAM user credentials or even root credentials, but this is not recommended. Instead, generate temporary credentials using one of the AWS Security Token Service (STS) APIs, or one of their AWS CLI wrappers.

It is best to place the AccessKeyId, SecretAccessKey, and SessionToken values that are returned by the STS call into appropriate environment variables in your shell session rather than into a configuration file. Then, when the shell terminates, the credentials are automatically discarded, which is not the case with a configuration file. Similarly, don't request a longer duration for the temporary credentials than you are likely to need.

The following example shows the steps you might take in a Linux shell to obtain temporary credentials that are good for half an hour using sts assume-role, and then place them in environment variables where awscurl can find them:

```
aws sts assume-role \
    --duration-seconds 1800 \
    --role-arn "arn:aws:iam::(account-id):role/(rolename)" \
    --role-session-name AWSCLI-Session > $output
AccessKeyId=$(echo $output | jq '.Credentials''.AccessKeyId')
SecretAccessKey=$(echo $output | jq '.Credentials''.SecretAccessKey')
SessionToken=$(echo $output | jq '.Credentials''.SessionToken')
```

```
export AWS_ACCESS_KEY_ID=$AccessKeyId
export AWS_SECRET_ACCESS_KEY=$SecretAccessKey
export AWS_SESSION_TOKEN=$SessionToken
```

You could then use `awscurl` to make a signed request to your DB cluster something like this:

```
awscurl (your cluster endpoint):8182/status \
    --region us-east-1 \
    --service neptune-db
```

Connecting to Amazon Neptune databases using IAM authentication with Gremlin console

To connect to Amazon Neptune using the Gremlin console with Signature Version 4 authentication, you use `requestInterceptor()` to plug in a SigV4 signer to the connection established by the `:remote` command. This requires you to configure the `Cluster` object manually and then pass it to the `:remote` command.

Note that this is quite different from the typical situation where the `:remote` command takes a configuration file to form the connection. The configuration file approach won't work because `requestInterceptor()` must be set programmatically, and can't load its configuration from a file.

> **Note**
>
> The following examples use `requestInterceptor()`, which was introduced in TinkerPop 3.6.6. If you are using a TinkerPop version earlier than 3.6.6 (but 3.5.5 or higher), use `handshakeInterceptor()` instead of `requestInterceptor()` in the code examples below.

The following prerequisites are necessary:

- You must have the IAM credentials needed to sign the requests. See Using the default credential provider chain in the AWS SDK for Java Developer Guide.
- You must have installed a Gremlin console version that is compatible with the version of the Neptune engine being used by your DB cluster.

If you are using temporary credentials, they expire after a specified interval, as does the session token, so you must update your session token when you request new credentials. See Using temporary security credentials to request access to AWS resources in the IAM User Guide.

For help connecting using SSL/TLS, see SSL/TLS configuration.

Connect the Gremlin console with Sig4 signing

1. Start the Gremlin console:

```
$ bin/gremlin.sh
```

2. At the `gremlin>` prompt, install the `amazon-neptune-sigv4-signer` library (this only needs to be done once for the console):

```
:install com.amazonaws amazon-neptune-sigv4-signer 2.4.0
```

   If you encounter problems with this step, it may help to consult the TinkerPop documentation about Grape configuration.

   > **Note**
   >
   > If you are using an HTTP proxy, you may encounter errors with this step where the `:install` command does not complete. To solve this problem, run the following commands to tell the console about the proxy:
   >
   > ```
   > System.setProperty("https.proxyHost", "(the proxy IP address)")
   > System.setProperty("https.proxyPort", "(the proxy port)")
   > ```

3. Import the class required to handle the signing into `requestInterceptor()`:

```
:import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider
:import com.amazonaws.neptune.auth.NeptuneNettyHttpSigV4Signer
```

4. If you are using temporary credentials, you will also need to supply your Session Token as follows:

```
System.setProperty("aws.sessionToken","(your session token)")
```

5. If you haven't otherwise established your account credentials, you can assign them as follows:

```
System.setProperty("aws.accessKeyId","(your access key)")
System.setProperty("aws.secretKey","(your secret key)")
```

6. Manually construct the `Cluster` object to connect to Neptune:

```
cluster = Cluster.build("(host name)")  \
                 .enableSsl(true) \
                 .requestInterceptor { r ->  \
                   def sigV4Signer = new NeptuneNettyHttpSigV4Signer("(Amazon
 region)", \
                                     DefaultCredentialsProvider.create()); \
                   sigV4Signer.signRequest(r); \
                   return r; } \
                 .create()
```

   For help finding the host name of a Neptune DB instance, see Connecting to Amazon Neptune Endpoints.

7. Establish the `:remote` connection using the variable name of the `Cluster` object in the previous step:

```
:remote connect tinkerpop.server cluster
```

8. Enter the following command to switch to remote mode. This sends all Gremlin queries to the remote connection:

```
:remote console
```

Connecting to Amazon Neptune databases using IAM with Gremlin Java

Here is an example of how to connect to Neptune using the Gremlin Java API with Sig4 signing (it assumes general knowledge about using Maven). This example uses the Amazon Neptune SigV4 Signer library to assist in request signing. First, define the dependencies as part of the `pom.xml` file:

> **Note**
>
> The following examples use `requestInterceptor()`, which was introduced in TinkerPop 3.6.6. If you are using a TinkerPop version earlier than 3.6.6 (but 3.5.5 or higher), use `handshakeInterceptor()` instead of `requestInterceptor()` in the code examples below.

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>amazon-neptune-sigv4-signer</artifactId>
  <version>3.1.0</version>
</dependency>
```

The Amazon Neptune SigV4 Signer supports both versions 1.x and 2.x of the AWS Java SDK. The following examples use 2.x, where `DefaultCredentialsProvider` is a `software.amazon.awssdk.auth.credentials.AwsCredentialsProvider` instance. If you are upgrading from 1.x to 2.x, see the Credentials provider changes in the AWS SDK for Java 2.x documentation.

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import com.amazonaws.neptune.auth.NeptuneNettyHttpSigV4Signer;
import com.amazonaws.neptune.auth.NeptuneSigV4SignerException;

 ...

System.setProperty("aws.accessKeyId","your-access-key");
System.setProperty("aws.secretKey","your-secret-key");

 ...

Cluster cluster = Cluster.build((your cluster))
                 .enableSsl(true)
                 .requestInterceptor( r ->
                  {
                    try {
                      NeptuneNettyHttpSigV4Signer sigV4Signer =
                        new NeptuneNettyHttpSigV4Signer("(your region)",
 DefaultCredentialsProvider.create());
                      sigV4Signer.signRequest(r);
                    } catch (NeptuneSigV4SignerException e) {
```

```
                        throw new RuntimeException("Exception occurred while signing the
 request", e);
                      }
                      return r;
                    }
                  ).create();
try {
  Client client = cluster.connect();
  client.submit("g.V().has('code','IAD')").all().get();
} catch (Exception e) {
  throw new RuntimeException("Exception occurred while connecting to cluster", e);
}
```

Cross account IAM authentication

Amazon Neptune supports cross account IAM authentication through the use of role assumption, also sometimes referred to as role chaining. To provide access to a Neptune cluster from an application hosted in a different AWS account:

- Create a new IAM user or role in the application AWS account, with a trust policy that allows the user or role to assume another IAM role. Assign this role to the compute hosting the application (EC2 instance, Lambda function, ECS Task, etc.).

  JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "assumeRolePolicy",
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Resource": "arn:aws-eusc:iam::111122223333:role/role-name"
    }
  ]
}
```

- Create a new IAM role in the Neptune database AWS account that allows access to the Neptune database and allows role assumption from the application account IAM user/role. Use a trust policy of:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "(ARN of application account IAM user or role)"
                ]
            },
            "Action": "sts:AssumeRole",
            "Condition": {}
        }
    ]
}
```

- Use the following code example as guidance on how to use these two roles to allow the application to access Neptune. In this example, the application account role will be assumed via the DefaultCredentialProviderChain when creating the `STSclient`. The `STSclient` is then used via the `STSAssumeRoleSessionCredentialsProvider` to assume the role hosted in the Neptune database AWS account.

```
public static void main( String[] args )
  {

    /*
     * Establish an STS client from the application account.
     */
    AWSSecurityTokenService client = AWSSecurityTokenServiceClientBuilder
        .standard()
        .build();

    /*
     * Define the role ARN that you will be assuming in the database account where
 the Neptune cluster resides.
     */
    String roleArnToAssume = "arn:aws:iam::012345678901:role/
CrossAccountNeptuneRole";
    String crossAccountSessionName = "cross-account-session-" + UUID.randomUUID();
```

```
   /*
    * Change the Credentials Provider in the SigV4 Signer to use the STSAssumeRole
Provider and provide it
    * with both the role to be assumed, the original STS client, and a session name
(which can be
    * arbitrary.)
    */
   Cluster cluster = Cluster.build()
                .addContactPoint("neptune-cluster.us-west-2.neptune.amazonaws.com")
                .enableSsl(true)
                .port(8182)
                .requestInterceptor( r ->
                 {
                   try {
                     NeptuneNettyHttpSigV4Signer sigV4Signer =
                       // new NeptuneNettyHttpSigV4Signer("us-west-2",
DefaultCredentialsProvider.create());
                       new NeptuneNettyHttpSigV4Signer(
                               "us-west-2",
                                new STSAssumeRoleSessionCredentialsProvider
                                   .Builder(roleArnToAssume,
crossAccountSessionName)
                                       .withStsClient(client)
                                       .build());
                     sigV4Signer.signRequest(r);
                   } catch (NeptuneSigV4SignerException e) {
                     throw new RuntimeException("Exception occurred while signing
the request", e);
                   }
                   return r;
                 }
                ).create();

   GraphTraversalSource g =
traversal().withRemote(DriverRemoteConnection.using(cluster));

   /* whatever application code is necessary */

   cluster.close();
 }
```

Connecting to Amazon Neptune databases using IAM authentication with Java and SPARQL

This section shows how to connect to Neptune using either RDF4J or Apache Jena with Signature Version 4 authentication.

Prerequisites

- Java 8 or higher.
- Apache Maven 3.3 or higher.

  For information about installing these prerequisites on an EC2 instance running Amazon Linux, see Prerequisites for connecting Amazon Neptune databases using IAM authentication.
- IAM credentials to sign the requests. For more information, see Using the Default Credential Provider Chain in the *AWS SDK for Java Developer Guide*.

  > **Note**
  >
  > If you are using temporary credentials, they expire after a specified interval, *including the session token*.
  > You must update your session token when you request new credentials. For more information, see Using Temporary Security Credentials to Request Access to AWS Resources in the *IAM User Guide*.

- Set the `SERVICE_REGION` variable to one of the following, indicating the Region of your Neptune DB instance:
  - US East (N. Virginia): `us-east-1`
  - US East (Ohio): `us-east-2`
  - US West (N. California): `us-west-1`
  - US West (Oregon): `us-west-2`
  - Canada (Central): `ca-central-1`
  - Canada West (Calgary): `ca-west-1`
  - South America (São Paulo): `sa-east-1`
  - Europe (Stockholm): `eu-north-1`
  - Europe (Spain): `eu-south-2`

- Europe (Ireland): `eu-west-1`
- Europe (London): `eu-west-2`
- Europe (Paris): `eu-west-3`
- Europe (Frankfurt): `eu-central-1`
- Middle East (Bahrain): `me-south-1`
- Middle East (UAE): `me-central-1`
- Israel (Tel Aviv): `il-central-1`
- Africa (Cape Town): `af-south-1`
- Asia Pacific (Hong Kong): `ap-east-1`
- Asia Pacific (Tokyo): `ap-northeast-1`
- Asia Pacific (Seoul): `ap-northeast-2`
- Asia Pacific (Osaka): `ap-northeast-3`
- Asia Pacific (Singapore): `ap-southeast-1`
- Asia Pacific (Sydney): `ap-southeast-2`
- Asia Pacific (Jakarta): `ap-southeast-3`
- Asia Pacific (Melbourne): `ap-southeast-4`
- Asia Pacific (Malaysia): `ap-southeast-5`
- Asia Pacific (Mumbai): `ap-south-1`
- Asia Pacific (Hyderabad): `ap-south-2`
- China (Beijing): `cn-north-1`
- China (Ningxia): `cn-northwest-1`
- AWS GovCloud (US-West): `us-gov-west-1`
- AWS GovCloud (US-East): `us-gov-east-1`

To connect to Neptune using either RDF4J or Apache Jena with Signature Version 4 signing

1. Clone the sample repository from GitHub.

```
git clone https://github.com/aws/amazon-neptune-sparql-java-sigv4.git
```

2. Change into the cloned directory.

```
cd amazon-neptune-sparql-java-sigv4
```

3. Get the latest version of the project by checking out the branch with the latest tag.

```
git checkout $(git describe --tags `git rev-list --tags --max-count=1`)
```

4. Enter one of the following commands to compile and run the example code.

   Replace *your-neptune-endpoint* with the hostname or IP address of your Neptune DB instance. The default port is 8182.

   > **Note**
   >
   > For information about finding the hostname of your Neptune DB instance, see the Connecting to Amazon Neptune Endpoints section.

   Eclipse RDF4J

   Enter the following to run the RDF4J example.

```
mvn compile exec:java \
    -Dexec.mainClass="com.amazonaws.neptune.client.rdf4j.NeptuneRdf4JSigV4Example"
 \
    -Dexec.args="https://your-neptune-endpoint:portsparql region-name"
```

   Apache Jena

   Enter the following to run the Apache Jena example.

```
mvn compile exec:java \
    -Dexec.mainClass="com.amazonaws.neptune.client.jena.NeptuneJenaSigV4Example" \
    -Dexec.args="https://your-neptune-endpoint:port"
```

5. To view the source code for the example, see the examples in the `src/main/java/com/amazonaws/neptune/client/` directory.

To use the SigV4 signing driver in your own Java application, add the `amazon-neptune-sigv4-signer` Maven package to the `<dependencies>` section of your `pom.xml`. We recommend that you use the examples as a starting point.

Connecting to Amazon Neptune databases using IAM authentication with SPARQL and Node.js

Querying using Signature V4 signing and the AWS SDK for Javascript V3

Here is an example of how to connect to Neptune SPARQL using Node.js with Signature Version 4 authentication and the AWS SDK for Javascript V3:

```
const { HttpRequest }  = require('@smithy/protocol-http');
const { fromNodeProviderChain } = require('@aws-sdk/credential-providers');
const { SignatureV4 } = require('@smithy/signature-v4');
const { Sha256 } = require('@aws-crypto/sha256-universal');
const https = require('https');

var region = 'us-west-2'; // e.g. us-west-1
var neptune_endpoint = 'your-Neptune-cluster-endpoint';  // like: 'cluster-
id.region.neptune.amazonaws.com'
var query = `query=PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX class: <http://aws.amazon.com/neptune/csv2rdf/class/>
PREFIX resource: <http://aws.amazon.com/neptune/csv2rdf/resource/>
PREFIX prop: <http://aws.amazon.com/neptune/csv2rdf/datatypeProperty/>
PREFIX objprop: <http://aws.amazon.com/neptune/csv2rdf/objectProperty/>

SELECT ?movies ?title WHERE {
  ?jel prop:name "James Earl Jones" .
  ?movies ?p2 ?jel .
  ?movies prop:title ?title
} LIMIT 10`;

runQuery(query);

function runQuery(q) {
  var request = new HttpRequest({
    hostname: neptune_endpoint,
    port: 8182,
    path: 'sparql',
    body: encodeURI(query),
    headers: {
```

```
      'Content-Type': 'application/x-www-form-urlencoded',
      'host': neptune_endpoint + ':8182',
    },
    method: 'POST',
  });

  const credentialProvider = fromNodeProviderChain();
  let credentials = credentialProvider();
  credentials.then(
    (cred)=>{
      var signer = new SignatureV4({credentials: cred, region: region, sha256: Sha256,
 service: 'neptune-db'});
      signer.sign(request).then(
        (req)=>{
          var responseBody = '';
          var sendreq = https.request(
            {
              host: req.hostname,
              port: req.port,
              path: req.path,
              method: req.method,
              headers: req.headers,
            },
          (res) => {
            res.on('data', (chunk) => { responseBody += chunk; });
            res.on('end', () => {
                console.log(JSON.parse(responseBody));
            });
          });
          sendreq.write(req.body);
          sendreq.end();
        }
      );
    },
    (err)=>{
      console.error(err);
    }
  );
}
```

Querying using Signature V4 signing and the AWS SDK for Javascript V2

Here is an example of how to connect to Neptune SPARQL using Node.js with Signature Version 4 authentication and the AWS SDK for Javascript V2:

```
var AWS = require('aws-sdk');

var region = 'us-west-2'; // e.g. us-west-1
var neptune_endpoint = 'your-Neptune-cluster-endpoint';  // like: 'cluster-
id.region.neptune.amazonaws.com'
var query = `query=PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX class: <http://aws.amazon.com/neptune/csv2rdf/class/>
PREFIX resource: <http://aws.amazon.com/neptune/csv2rdf/resource/>
PREFIX prop: <http://aws.amazon.com/neptune/csv2rdf/datatypeProperty/>
PREFIX objprop: <http://aws.amazon.com/neptune/csv2rdf/objectProperty/>

SELECT ?movies ?title WHERE {
    ?jel prop:name "James Earl Jones" .
    ?movies ?p2 ?jel .
    ?movies prop:title ?title
} LIMIT 10`;

runQuery(query);

function runQuery(q) {

    var endpoint = new AWS.Endpoint(neptune_endpoint);
    endpoint.port = 8182;
    var request = new AWS.HttpRequest(endpoint, region);
    request.path += 'sparql';
    request.body = encodeURI(query);
    request.headers['Content-Type'] = 'application/x-www-form-urlencoded';
    request.headers['host'] = neptune_endpoint;
    request.method = 'POST';

    var credentials = new AWS.CredentialProviderChain();
    credentials.resolve((err, cred)=>{
        var signer = new AWS.Signers.V4(request, 'neptune-db');
        signer.addAuthorization(cred, new Date());
    });

    var client = new AWS.HttpClient();
    client.handleRequest(request, null, function(response) {
```

```
        console.log(response.statusCode + ' ' + response.statusMessage);
        var responseBody = '';
        response.on('data', function (chunk) {
            responseBody += chunk;
        });
        response.on('end', function (chunk) {
            console.log('Response body: ' + responseBody);
        });
    }, function(error) {
        console.log('Error: ' + error);
    });
}
```

Connecting to Amazon Neptune databases using IAM authentication with Python

The `boto3` `neptunedata` client provides the simplest way to connect to an IAM-enabled Neptune database from Python. The client handles Signature Version 4 signing automatically, so you don't need to sign requests yourself.

Prerequisites

- Python 3.x
- The `boto3` library: `pip install boto3`
- AWS credentials configured through any standard method (environment variables, AWS config file, instance profile, or Lambda execution role)
- An IAM policy that allows `neptune-db:*` actions on your Neptune cluster

Connecting with default credentials

This approach works when `boto3` can resolve credentials automatically, such as on Amazon Elastic Compute Cloud instances with instance profiles, in AWS Lambda functions, or with configured AWS profiles.

Replace the endpoint URL with your Neptune cluster endpoint.

Gremlin and openCypher

```
import boto3
```

```
from botocore.config import Config

cfg = Config(retries={"total_max_attempts": 1, "mode": "standard"}, read_timeout=None)

neptune = boto3.client('neptunedata', config=cfg,
    region_name='us-east-1',
    endpoint_url='https://your-neptune-endpoint:8182')

# Gremlin
resp = neptune.execute_gremlin_query(gremlinQuery='g.V().limit(1)')
print(resp['result'])

# openCypher
resp = neptune.execute_open_cypher_query(openCypherQuery='MATCH (n) RETURN n LIMIT 1')
print(resp['results'])
```

SPARQL

The `boto3 neptunedata` client does not currently support SPARQL. To send SPARQL queries to an IAM-enabled Neptune database, sign requests manually by using Signature Version 4 with the `botocore` request-signing utilities and the `requests` library (`pip install boto3 requests`).

```
import requests
from botocore.auth import SigV4Auth
from botocore.awsrequest import AWSRequest
from botocore.session import Session

endpoint = 'https://your-neptune-endpoint:8182'
region = 'us-east-1'

query = 'SELECT ?s ?p ?o WHERE { ?s ?p ?o } LIMIT 1'

request = AWSRequest(method='POST', url=f'{endpoint}/sparql/',
    data={'query': query})
SigV4Auth(Session().get_credentials(), 'neptune-db', region).add_auth(request)

resp = requests.post(f'{endpoint}/sparql/', headers=request.headers,
    data={'query': query})
print(resp.json())
```

> **Note**
>
> This example uses `botocore.session.Session` to resolve AWS credentials automatically from environment variables, AWS config files, instance profiles, or Lambda execution roles. You don't need to set credentials explicitly.

Using temporary credentials

For cross-account access, time-limited sessions, or as a security best practice to avoid long-lived credentials, use AWS STS to obtain temporary credentials and create a `boto3.Session`.

Gremlin and openCypher

```
import boto3
from botocore.config import Config

sts = boto3.client('sts')
creds = sts.assume_role(
    RoleArn='arn:aws:iam::123456789012:role/NeptuneRole',
    RoleSessionName='neptune-session'
)['Credentials']

session = boto3.Session(
    aws_access_key_id=creds['AccessKeyId'],
    aws_secret_access_key=creds['SecretAccessKey'],
    aws_session_token=creds['SessionToken'],
    region_name='us-east-1'
)

cfg = Config(retries={"total_max_attempts": 1, "mode": "standard"}, read_timeout=None)

neptune = session.client('neptunedata', config=cfg,
    endpoint_url='https://your-neptune-endpoint:8182')

# Gremlin
resp = neptune.execute_gremlin_query(gremlinQuery='g.V().limit(1)')
print(resp['result'])

# openCypher
resp = neptune.execute_open_cypher_query(openCypherQuery='MATCH (n) RETURN n LIMIT 1')
print(resp['results'])
```

> **Note**
>
> Temporary credentials expire after a specified interval (the default is 1 hour) and are not automatically refreshed by the client. For long-running applications, use profile-based credentials or instance profiles instead.

SPARQL

Because the `boto3 neptunedata` client does not currently support SPARQL, you must sign requests manually with the temporary credentials.

```
import boto3
import requests
from botocore.auth import SigV4Auth
from botocore.awsrequest import AWSRequest
from botocore.credentials import ReadOnlyCredentials

sts = boto3.client('sts')
creds = sts.assume_role(
    RoleArn='arn:aws:iam::123456789012:role/NeptuneRole',
    RoleSessionName='neptune-session'
)['Credentials']

endpoint = 'https://your-neptune-endpoint:8182'
region = 'us-east-1'

query = 'SELECT ?s ?p ?o WHERE { ?s ?p ?o } LIMIT 1'

request = AWSRequest(method='POST', url=f'{endpoint}/sparql/',
    data={'query': query})
SigV4Auth(ReadOnlyCredentials(
    creds['AccessKeyId'], creds['SecretAccessKey'], creds['SessionToken']
), 'neptune-db', region).add_auth(request)

resp = requests.post(f'{endpoint}/sparql/', headers=request.headers,
    data={'query': query})
print(resp.json())
```

Using with AWS Lambda

In Lambda, the execution role provides credentials automatically through `boto3`. The default credentials example works without modification.

For a complete Lambda example with connection management and retry logic, see the section called "Lambda function examples".

Connecting to Amazon Neptune databases using IAM authentication with Gremlin Python

Overview

This guide demonstrates how to connect to an Amazon Neptune database with IAM authentication enabled using the Gremlin Python driver, with Signature Version 4 authentication and the AWS SDK for Python (Boto3).

Prerequisites

- An Amazon Neptune cluster with IAM authentication enabled.
- Python 3.10 or later. The `gremlinpython` library aligned with TinkerPop 3.8.x requires this version.
- AWS credentials configured (via environment variables, shared credentials file, or IAM role).

Create a basic connection

Use the following code example as guidance on how to establish basic connection with IAM authentication using the Gremlin Python driver.

```
from boto3 import Session
from botocore.auth import SigV4Auth
from botocore.awsrequest import AWSRequest

from gremlin_python.process.anonymous_traversal import traversal
from gremlin_python.driver.driver_remote_connection import DriverRemoteConnection

def main():
    endpoint = 'your.cluster.endpoint.neptune.amazonaws.com'
    conn_string = 'wss://' + endpoint + ':8182/gremlin'
```

```
    default_region = 'us-east-1'
    service = 'neptune-db'

    credentials = Session().get_credentials()
    if credentials is None:
        raise Exception("No AWS credentials found")
    creds = credentials.get_frozen_credentials()
    # region set inside config profile or via AWS_DEFAULT_REGION environment variable
 will be loaded
    region = Session().region_name if Session().region_name else default_region

    request = AWSRequest(method='GET', url=conn_string, data=None)
    SigV4Auth(creds, service, region).add_auth(request)

    rc = DriverRemoteConnection(conn_string, 'g', headers=request.headers.items())
    g = traversal().with_remote(rc)

    # simple query to verify connection
    count = g.V().count().next()
    print('Vertex count: ' + str(count))

    # cleanup
    rc.close()

if __name__ == "__main__":
    main()
```

Connecting to Amazon Neptune databases using IAM authentication with Gremlin JavaScript

Overview

This guide demonstrates how to connect to an Amazon Neptune database with IAM authentication enabled using the Gremlin JavaScript driver, with Signature Version 4 authentication and the AWS SDK for Javascript v3.

Prerequisites

- An Amazon Neptune cluster with IAM authentication enabled.
- Node 13 or later (refer to minimal versions for Gremlin JavaScript and AWS SDK for Javascript v3).

- AWS credentials configured (via environment variables, shared credentials file, or IAM role).

Create a basic connection

Use the following code example as guidance on how to establish a basic connection with IAM authentication using the Gremlin JavaScript driver.

```
const { fromNodeProviderChain } = require('@aws-sdk/credential-providers');
const { getUrlAndHeaders } = require('gremlin-aws-sigv4/lib/utils');
const { loadConfig } = require("@smithy/node-config-provider");
const { NODE_REGION_CONFIG_FILE_OPTIONS, NODE_REGION_CONFIG_OPTIONS } =
 require("@smithy/config-resolver");

const gremlin = require('gremlin');
const DriverRemoteConnection = gremlin.driver.DriverRemoteConnection;
const traversal = gremlin.process.AnonymousTraversalSource.traversal;

const DEFAULT_REGION = 'us-east-1';

const getCredentials = async () => {
    try {
        // Loads the default credential provider chain
        return await fromNodeProviderChain();
    } catch (e) {
        console.error("No credentials found", e);
        throw e;
    }
};

(main = async () => {
    console.log('Starting');

    const credentials = await getCredentials();
    try {
        // region set inside config profile or via AWS_REGION environment variable will
 be loaded
        credentials['region'] = await loadConfig(NODE_REGION_CONFIG_OPTIONS,
 NODE_REGION_CONFIG_FILE_OPTIONS)();
    } catch (e) {
        credentials['region'] = DEFAULT_REGION
    }

    const connInfo = getUrlAndHeaders(
```

```
        'you.cluster.endpoint.neptune.amazonaws.com',
        '8182',
        credentials,
        '/gremlin',
        'wss');

    const dc = new DriverRemoteConnection(connInfo['url'], {headers:
 connInfo['headers']});
    const g = traversal().withRemote(dc);

    const query = g.V().limit(5).count();
    const count = await query.next();
    console.log("Vertex count: " + count.value);

    await dc.close();
})

main();
```

Connecting to Amazon Neptune databases using IAM authentication with Gremlin Go

Overview

This guide demonstrates how to connect to a Amazon Neptune database with IAM authentication enabled using the Gremlin Go driver, with Signature Version 4 authentication and the AWS SDK for GO v2.

Prerequisites

- An Amazon Neptune cluster with IAM authentication enabled.
- Go 1.25 or later, as required by TinkerPop 3.8.x. For information about minimum supported versions, see Gremlin Go and the AWS SDK for Go v2.
- AWS credentials configured (via environment variables, shared credentials file, or IAM role)

Create a basic connection

Use the following code example as guidance on how to establish a basic connection with IAM authentication using the Gremlin Go driver.

```
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-sdk-go-v2/config"
 "net/http"
 "strings"
 "time"

 gremlingo "github.com/apache/tinkerpop/gremlin-go/v3/driver"
 v4 "github.com/aws/aws-sdk-go-v2/aws/signer/v4"
)

const emptyStringSHA256 =
 `e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`

func main() {
 neptuneEndpoint := "you.cluster.endpoint.neptune.amazonaws.com"
 connString := "wss://" + neptuneEndpoint + ":8182/gremlin"
 service := "neptune-db"
 defaultRegion := "us-east-1"

 // Create request to sign
 req, err := http.NewRequest(http.MethodGet, connString, strings.NewReader(""))
 if err != nil {
  fmt.Println(err)
  return
 }

 // Loads the default config with default credentials provider
 // See https://github.com/aws/aws-sdk-go-v2 for additional docs on API usage
 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  fmt.Println(err)
  return
 }
 // Retrieve loaded credentials
 cr, err := cfg.Credentials.Retrieve(context.TODO())
 if err != nil {
  fmt.Println(err)
  return
 }
```

```
 region := defaultRegion
 if cfg.Region != "" {
  // region set inside config profile, or via AWS_REGION or AWS_DEFAULT_REGION
 environment variable will be loaded
  region = cfg.Region
 }

 signer := v4.NewSigner()
 // Sign request
 err = signer.SignHTTP(context.TODO(), cr, req, emptyStringSHA256, service, "us-
east-2", time.Now())
 if err != nil {
  fmt.Println(err)
  return
 }

 // Pass the signed request header into gremlingo.HeaderAuthInfo()
 driverRemoteConnection, err := gremlingo.NewDriverRemoteConnection(connString,
  func(settings *gremlingo.DriverRemoteConnectionSettings) {
   settings.TraversalSource = "g"
   settings.AuthInfo = gremlingo.HeaderAuthInfo(req.Header)
   // settings.TlsConfig = &tls.Config{InsecureSkipVerify: true} // Use this only if
 you're on a Mac running Go 1.18+ doing local dev. See https://github.com/golang/go/
issues/51991
  })
 if err != nil {
  fmt.Println(err)
  return
 }

 // Cleanup
 defer driverRemoteConnection.Close()

 // Creating graph traversal
 g := gremlingo.Traversal_().WithRemote(driverRemoteConnection)

 // Query execution
 count, err := g.V().Limit(5).Count().Next()
 if err != nil {
  fmt.Println(err)
  return
 }
 fmt.Println("Vertex count:", *count)
```

```
}
```

Gremlin Go Dynamic Credential Refresh

Gremlin Go has DynamicAuth which allows the injection of a function pointer to retrieve credentials and generate the header, which prevents header expiry with long-running connections.

```
package main

import (
 "context"
 "crypto/tls"
 "fmt"
 "github.com/aws/aws-sdk-go-v2/config"
 "net/http"
 "strings"
 "time"

 gremlingo "github.com/apache/tinkerpop/gremlin-go/v3/driver"
 v4 "github.com/aws/aws-sdk-go-v2/aws/signer/v4"
)

const emptyStringSHA256 =
 `e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`

func main() {
 neptuneEndpoint := "you.cluster.endpoint.neptune.amazonaws.com"
 connString := "wss://" + neptuneEndpoint + ":8182/gremlin"
 service := "neptune-db"
 defaultRegion := "us-east-1"

 //Create the request to sign
 req, err := http.NewRequest(http.MethodGet, connString, strings.NewReader(""))
 if err != nil {
  fmt.Println(err)
  return
 }

 // Loads the default config with default credentials provider
 // See https://github.com/aws/aws-sdk-go-v2 for additional docs on API usage
 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  fmt.Println(err)
```

```
 return
}

region := defaultRegion
if cfg.Region != "" {
 // region set inside config profile, or via AWS_REGION or AWS_DEFAULT_REGION
environment variable will be loaded
 region = cfg.Region
}

signer := v4.NewSigner()

// This is the function that will be used for dynamic refreseh of credentials and
signed headers
gen := func() gremlingo.AuthInfoProvider {
 // Retrieve loaded credentials
 cr, err := cfg.Credentials.Retrieve(context.TODO())
 fmt.Println("AWS Credentials: ", cr)
 if err != nil {
  fmt.Println(err)
  return
 }
 // Sign request
 err = signer.SignHTTP(context.TODO(), cr, req, emptyStringSHA256, service, region,
time.Now())
 if err != nil {
  fmt.Println(err)
  return
 }
 fmt.Println(req.Header)
 return gremlingo.HeaderAuthInfo(req.Header)
}

// Pass the function into gremlingo.NewDynamicAuth(), which will generate the
AuthInfoProvider to pass into gremlingo.DriverRemoteConnectionSettings below
auth := gremlingo.NewDynamicAuth(gen)

driverRemoteConnection, err := gremlingo.NewDriverRemoteConnection(connString,
 func(settings *gremlingo.DriverRemoteConnectionSettings) {
  settings.TraversalSource = "g"
  settings.AuthInfo = auth
  // settings.TlsConfig = &tls.Config{InsecureSkipVerify: true} // Use this only if
you're on a Mac running Go 1.18+ doing local dev. See https://github.com/golang/go/
issues/51991
```

```
  })
 if err != nil {
  fmt.Println(err)
  return
 }

 // Cleanup
 defer driverRemoteConnection.Close()

 // Creating graph traversal
 g := gremlingo.Traversal_().WithRemote(driverRemoteConnection)

 // Query execution
 count, err := g.V().Limit(5).Count().Next()
 if err != nil {
  fmt.Println(err)
  return
 }
 fmt.Println("Vertex count:", *count)
}
```

Connecting to Amazon Neptune databases using IAM authentication with Gremlin .NET

Overview

This guide demonstrates how to connect to a Amazon Neptune database with IAM authentication enabled using the Gremlin .NET driver, with Signature Version 4 authentication and the AWS SDK for .NET v3.

Prerequisites

- An Amazon Neptune cluster with IAM authentication enabled.
- .NET 6.0 or later. The `gremlin.net` library, version 3.8.0, requires .NET 6.0 or later.
- AWS credentials configured (through environment variables, the shared credentials file, or an IAM role).

Create a basic connection

To connect with Gremlin .NET, use the `SigV4RequestSigner` source file from the custom library https://github.com/aws/amazon-neptune-gremlin-dotnet-sigv4. An example project set-up is located in https://github.com/aws/amazon-neptune-gremlin-dotnet-sigv4/tree/main/example, and reflected below.

Project File:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp8.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="../src/SigV4RequestSigner.cs" Link="SigV4RequestSigner.cs" />
    <PackageReference Include="AWSSDK.Core" Version="3.7.402.24" />
    <PackageReference Include="gremlin.net" Version="3.8.0" />
  </ItemGroup>

</Project>
```

Example Program:

```
using System;
using System.Collections.Generic;
using System.Net.WebSockets;
using System.Linq;
using System.Net.Http;
using System.Security.Cryptography;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Web;
using Gremlin.Net;
using Gremlin.Net.Driver;
using Gremlin.Net.Driver.Remote;
using Gremlin.Net.Process;
using Gremlin.Net.Process.Traversal;
using Gremlin.Net.Structure;
using static Gremlin.Net.Process.Traversal.AnonymousTraversalSource;
```

```
using static Gremlin.Net.Process.Traversal.__;
using static Gremlin.Net.Process.Traversal.P;
using static Gremlin.Net.Process.Traversal.Order;
using static Gremlin.Net.Process.Traversal.Operator;
using static Gremlin.Net.Process.Traversal.Pop;
using static Gremlin.Net.Process.Traversal.Scope;
using static Gremlin.Net.Process.Traversal.TextP;
using static Gremlin.Net.Process.Traversal.Column;
using static Gremlin.Net.Process.Traversal.Direction;
using static Gremlin.Net.Process.Traversal.T;
using Amazon.Runtime.CredentialManagement;
using Amazon.Runtime;
using Amazon;
using Amazon.Util;
using Amazon.Neptune.Gremlin.Driver;

namespace NeptuneExample
{
    class Program
    {
        static void Main(string[] args)
        {
            /*
                Include your Neptune endpoint and port below.
            */
            var neptune_host = "neptune-endpoint"; // ex: mycluster.cluster.us-
east-1.neptune.amazonaws.com
            var neptune_port = 8182;

            var gremlinServer = new GremlinServer(neptune_host, neptune_port);
            var gremlinClient = new GremlinClient(gremlinServer,
 webSocketConfiguration: new SigV4RequestSigner().signRequest(neptune_host,
 neptune_port));
            var remoteConnection = new DriverRemoteConnection(gremlinClient);
            var g = Traversal().WithRemote(remoteConnection);

            /* Example code to pull the first 5 vertices in a graph. */
            Console.WriteLine("Get List of Node Labels:");
            Int32 limitValue = 5;
            var output = g.V().Limit<Vertex>(limitValue).ToList();
            foreach(var item in output) {
                Console.WriteLine(item);
            }
        }
```

```
    }
}
```

Managing access to Amazon Neptune databases using IAM policies

IAM policies are JSON objects that define permissions to use actions and resources.

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-Based Policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Using Service Control Policies (SCP) with AWS organizations

Service control policies (SCPs) are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for

grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see How SCPs work in the AWS Organizations User Guide.

Customers deploying Amazon Neptune in an AWS Account within an AWS Organization can leverage SCPs to control which accounts can use Neptune. To ensure access to Neptune within a member account, be sure to:

- Allow access to `rds:*` and `neptune-db:*` for Neptune database operations. Refer to Why are Amazon RDS permissions and resources required to use Neptune Database? for details on why Amazon RDS permissions are needed for Neptune database.
- Allow access to `neptune-graph:*` for Neptune Analytics operations.

Permissions Required to Use the Amazon Neptune Console

For a user to work with the Amazon Neptune console, that user must have a minimum set of permissions. These permissions allow the user to describe the Neptune resources for their AWS account and to provide other related information, including Amazon EC2 security and network information.

If you create an IAM policy that is more restrictive than the minimum required permissions, the console won't function as intended for users with that IAM policy. To ensure that those users can still use the Neptune console, also attach the `NeptuneReadOnlyAccess` managed policy to the user, as described in Using AWS managed policies to access Amazon Neptune databases.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the Amazon Neptune API.

Attaching an IAM Policy to an IAM user

To apply a managed or custom policy, you attach it to an IAM user. For a tutorial on this topic, see Create and Attach Your First Customer Managed Policy in the *IAM User Guide*.

As you work through the tutorial, you can use one of the policy examples shown in this section as a starting point and tailor it to your needs. At the end of the tutorial, you have an IAM user with an attached policy that can use the `neptune-db:*` action.

Important

- Changes to an IAM policy take up to 10 minutes to apply to the specified Neptune resources.
- IAM policies applied to a Neptune DB cluster apply to all instances in that cluster.

Using different kinds of IAM policies for controlling access to Neptune

To provide access to Neptune administrative actions or to data in a Neptune DB cluster, you attach policies to an IAM user or role. For information about how to attach an IAM policy to a user, see Attaching an IAM Policy to an IAM user. For information about attaching a policy to a role, see Adding and Removing IAM Policies in the *IAM User Guide*.

For general access to Neptune, you can use one of Neptune's managed policies. For more restricted access, you can create your own custom policy using the administrative actions and resources that Neptune supports..

In a custom IAM policy, you can use two different kinds of policy statement that control different modes of access to a Neptune DB cluster:

- Administrative policy statements – Administrative policy statements provide access to the Neptune management APIs that you use to create, configure and manage a DB cluster and its instances.

  Because Neptune shares functionality with Amazon RDS, administrative actions, resources, and condition keys in Neptune policies use an `rds:` prefix by design.
- Data-access policy statements – Data-access policy statements use data-access actions, resources, and condition keys to control access the data that a DB cluster contains.

  Neptune data-access actions, resources and condition keys use a `neptune-db:` prefix.

Using IAM condition context keys in Amazon Neptune

You can specify conditions in an IAM policy statement that controls access to Neptune. The policy statement then takes effect only when the conditions are true.

For example, you might want a policy statement to take effect only after a specific date, or allows access only when a specific value is present in the request.

To express conditions, you use predefined condition keys in the `Condition` element of a policy statement, together with IAM condition policy operators such as equals or less than.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM Policy Elements: Variables and Tags in the *IAM User Guide*.

The data type of a condition key determines which condition operators you can use to compare values in the request with the values in the policy statement. If you use a condition operator that is not compatible with that data type, the match always fails and the policy statement never applies.

Neptune supports different sets of condition keys for administrative policy statements than for data-access policy statements:

- Condition keys for administrative policy statements
- Condition keys for data-access policy statements

Support for IAM policy and access-control features in Amazon Neptune

The following table shows what IAM features Neptune supports for administrative policy statements and data-access policy statements:

IAM features you can use with Neptune

IAM feature	Administrative	Data-access
Identity-based policies	Yes	Yes
Resource-based policies	No	No
Policy actions	Yes	Yes

IAM feature	Administrative	Data-access
Policy resources	Yes	Yes
Global condition keys	Yes	(a subset)
Tag-based condition keys	Yes	No
Access Control Lists (ACLs)	No	No
Service control policies (SCPs)	Yes	Yes
Service linked roles	Yes	No

IAM Policy Limitations

Changes to an IAM policy take up to 10 minutes to apply to the specified Neptune resources.

IAM policies applied to a Neptune DB cluster apply to all instances in that cluster.

Neptune does not currently support cross-account access control at the data plane level. Cross-account access control is only supported when bulk-loading and by using role chaining. For more information, see Bulk load tutorial .

Using AWS managed policies to access Amazon Neptune databases

AWS addresses many common use cases by providing standalone IAM policies that are created and administered by AWS. Managed policies grant necessary permissions for common use cases so you can avoid having to investigate what permissions are needed. For more information, see AWS Managed Policies in the *IAM User Guide*.

The following AWS managed policies, which you can attach to users in your account, are for using Amazon Neptune management APIs:

- **NeptuneReadOnlyAccess** — Grants read-only administrative actions (such as `rds:Describe*` and `rds:ListTagsForResource`) and read-only data-access actions (`neptune-db:Read*`, `neptune-db:Get*`, and `neptune-db:List*`) on all Neptune resources. Use this policy for users who need to view cluster configurations and query data without making changes.

- **NeptuneFullAccess** — Grants all administrative actions (`rds:*` on Neptune resources) and all data-access actions (`neptune-db:*`). This policy is suitable for administrators who manage Neptune clusters through the AWS CLI or SDK but do not need AWS Management Console access.
- **NeptuneConsoleFullAccess** — Grants all administrative actions on Neptune resources plus additional permissions for Amazon EC2 (VPC), IAM, and Neptune Analytics that are needed for AWS Management Console workflows. This policy does not include data-access actions (`neptune-db:*`). Use this policy for users who manage Neptune through the AWS Management Console.
- **NeptuneGraphReadOnlyAccess** — This policy is for Neptune Analytics. For details, see NeptuneGraphReadOnlyAccess in Neptune Analytics.
- **AWSServiceRoleForNeptuneGraphPolicy** — This policy is for Neptune Analytics. For details, see AWSServiceRoleForNeptuneGraphPolicy in Neptune Analytics.

Neptune IAM roles and policies grant some access to Amazon RDS resources, because Neptune shares operational technology with Amazon RDS for certain management features. This includes administrative API permissions, which is why Neptune administrative actions have an `rds:` prefix.

Creating custom policies

If the AWS managed policies are too broad for your use case, you can create custom IAM policies that grant only the specific permissions you need. Neptune supports two categories of custom policies:

- **Administrative policies** — Control access to Neptune management operations such as creating, modifying, and deleting clusters and instances. These actions use the `rds:` prefix. For examples, see the section called “Administrative policy examples”.
- **Data-access policies** — Control access to the data in your Neptune graph database, including read, write, and delete operations. These actions use the `neptune-db:` prefix. For examples, see the section called “Data-access policy examples”.

By combining administrative and data-access policy statements, you can grant fine-grained permissions tailored to each user or role in your organization.

Validating IAM policies

When you create or edit custom IAM policies, we recommend that you validate them before applying them to users, groups, or roles.

IAM Access Analyzer policy validation — IAM Access Analyzer provides policy checks that validate your IAM policies against IAM policy grammar and AWS best practices. It identifies errors, security warnings, and suggestions to help you author policies that are functional and conform to security best practices. For more information, see Validating policies with IAM Access Analyzer in the *IAM User Guide*.

IAM Policy Simulator — The IAM Policy Simulator lets you test the effects of IAM policies before committing them to production. You can simulate API calls to AWS services to verify that your policies grant or deny the expected access. For more information, see Testing IAM policies with the IAM Policy Simulator in the *IAM User Guide*.

Updates to Neptune AWS managed policies

The following table tracks updates to Neptune managed policies starting from the time Neptune began tracking these changes:

Policy	Description	Date
AWS managed policies for Amazon Neptune - update to existing policies	The `NeptuneReadOnlyAcess` and `NeptuneFullAccess` managed policies now include `Sid` (statement ID) as an identifier in the policy statement.	2024-01-22
NeptuneGraphReadOnlyAccess (released)	Released to provide read-only access to Neptune Analytics graphs and resources.	2023-11-29
AWSServiceRoleForNeptuneGraphPolicy (released)	Released to allow Neptune Analytics graphs access to CloudWatch to publish operational and usage metrics and logs. See Using service-l	2023-11-29

Policy	Description	Date
	inked roles (SLRs) in Neptune Analytics.	
NeptuneConsoleFullAccess (added permissions)	Added permissions provide all access needed to interact with Neptune Analytics graphs.	2023-11/29
NeptuneFullAccess (added permissions)	Added data-access permissio ns, and permissions for new global database APIs.	2022-07-28
NeptuneConsoleFullAccess (added permissions)	Added permissions for new global database APIs.	2022-07-21
Neptune started tracking changes	Neptune began tracking changes to its AWS managed policies.	2022-07-21

Granting `NeptuneReadOnlyAccess` to Amazon Neptune databases using AWS managed policy

The NeptuneReadOnlyAccess managed policy below grants read-only access to all Neptune actions and resources for both administrative and data-access purposes.

> **Note**
>
> This policy was updated on 2022-07-21 to include read-only data-access permissions as well as read-only administrative permissions and to include permissions for global database actions.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
```

```
        {
            "Sid": "AllowReadOnlyPermissionsForRDS",
            "Effect": "Allow",
            "Action": [
                "rds:DescribeAccountAttributes",
                "rds:DescribeCertificates",
                "rds:DescribeDBClusterParameterGroups",
                "rds:DescribeDBClusterParameters",
                "rds:DescribeDBClusterSnapshotAttributes",
                "rds:DescribeDBClusterSnapshots",
                "rds:DescribeDBClusters",
                "rds:DescribeDBEngineVersions",
                "rds:DescribeDBInstances",
                "rds:DescribeDBLogFiles",
                "rds:DescribeDBParameterGroups",
                "rds:DescribeDBParameters",
                "rds:DescribeDBSubnetGroups",
                "rds:DescribeEventCategories",
                "rds:DescribeEventSubscriptions",
                "rds:DescribeEvents",
                "rds:DescribeGlobalClusters",
                "rds:DescribeOrderableDBInstanceOptions",
                "rds:DescribePendingMaintenanceActions",
                "rds:DownloadDBLogFilePortion",
                "rds:ListTagsForResource"
            ],
            "Resource": "*"
        },
        {
            "Sid": "AllowReadOnlyPermissionsForCloudwatch",
            "Effect": "Allow",
            "Action": [
                "cloudwatch:GetMetricStatistics",
                "cloudwatch:ListMetrics"
            ],
            "Resource": "*"
        },
        {
            "Sid": "AllowReadOnlyPermissionsForEC2",
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeAccountAttributes",
                "ec2:DescribeAvailabilityZones",
                "ec2:DescribeInternetGateways",
```

```
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcAttribute",
                "ec2:DescribeVpcs"
            ],
            "Resource": "*"
        },
        {
            "Sid": "AllowReadOnlyPermissionsForKMS",
            "Effect": "Allow",
            "Action": [
                "kms:ListKeys",
                "kms:ListRetirableGrants",
                "kms:ListAliases",
                "kms:ListKeyPolicies"
            ],
            "Resource": "*"
        },
        {
            "Sid": "AllowReadOnlyPermissionsForLogs",
            "Effect": "Allow",
            "Action": [
                "logs:DescribeLogStreams",
                "logs:GetLogEvents"
            ],
            "Resource": [
                "arn:aws-eusc:logs:*:*:log-group:/aws/rds/*:log-stream:*",
                "arn:aws-eusc:logs:*:*:log-group:/aws/neptune/*:log-stream:*"
            ]
        },
        {
            "Sid": "AllowReadOnlyPermissionsForNeptuneDB",
            "Effect": "Allow",
            "Action": [
                "neptune-db:Read*",
                "neptune-db:Get*",
                "neptune-db:List*"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
```

```
}
```

Granting NeptuneFullAccess to Amazon Neptune databases using AWS managed policy

The NeptuneFullAccess managed policy below grants full access to all Neptune actions and resources for both administrative and data-access purposes. It is recommended if you need full access from the AWS CLI or from an SDK, but not from the AWS Management Console.

> **Note**
>
> This policy was updated on 2022-07-21 to include full data-access permissions as well as full administrative permissions and to include permissions for global database actions.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowNeptuneCreate",
            "Effect": "Allow",
            "Action": [
                "rds:CreateDBCluster",
                "rds:CreateDBInstance"
            ],
            "Resource": [
                "arn:aws-eusc:rds:*:*:*"
            ],
            "Condition": {
                "StringEquals": {
                    "rds:DatabaseEngine": [
                        "graphdb",
                        "neptune"
                    ]
                }
            }
        },
        {
```

```
            "Sid": "AllowManagementPermissionsForRDS",
            "Effect": "Allow",
            "Action": [
                "rds:AddRoleToDBCluster",
                "rds:AddSourceIdentifierToSubscription",
                "rds:AddTagsToResource",
                "rds:ApplyPendingMaintenanceAction",
                "rds:CopyDBClusterParameterGroup",
                "rds:CopyDBClusterSnapshot",
                "rds:CopyDBParameterGroup",
                "rds:CreateDBClusterEndpoint",
                "rds:CreateDBClusterParameterGroup",
                "rds:CreateDBClusterSnapshot",
                "rds:CreateDBParameterGroup",
                "rds:CreateDBSubnetGroup",
                "rds:CreateEventSubscription",
                "rds:CreateGlobalCluster",
                "rds:DeleteDBCluster",
                "rds:DeleteDBClusterEndpoint",
                "rds:DeleteDBClusterParameterGroup",
                "rds:DeleteDBClusterSnapshot",
                "rds:DeleteDBInstance",
                "rds:DeleteDBParameterGroup",
                "rds:DeleteDBSubnetGroup",
                "rds:DeleteEventSubscription",
                "rds:DeleteGlobalCluster",
                "rds:DescribeDBClusterEndpoints",
                "rds:DescribeAccountAttributes",
                "rds:DescribeCertificates",
                "rds:DescribeDBClusterParameterGroups",
                "rds:DescribeDBClusterParameters",
                "rds:DescribeDBClusterSnapshotAttributes",
                "rds:DescribeDBClusterSnapshots",
                "rds:DescribeDBClusters",
                "rds:DescribeDBEngineVersions",
                "rds:DescribeDBInstances",
                "rds:DescribeDBLogFiles",
                "rds:DescribeDBParameterGroups",
                "rds:DescribeDBParameters",
                "rds:DescribeDBSecurityGroups",
                "rds:DescribeDBSubnetGroups",
                "rds:DescribeEngineDefaultClusterParameters",
                "rds:DescribeEngineDefaultParameters",
                "rds:DescribeEventCategories",
```

```
                "rds:DescribeEventSubscriptions",
                "rds:DescribeEvents",
                "rds:DescribeGlobalClusters",
                "rds:DescribeOptionGroups",
                "rds:DescribeOrderableDBInstanceOptions",
                "rds:DescribePendingMaintenanceActions",
                "rds:DescribeValidDBInstanceModifications",
                "rds:DownloadDBLogFilePortion",
                "rds:FailoverDBCluster",
                "rds:FailoverGlobalCluster",
                "rds:ListTagsForResource",
                "rds:ModifyDBCluster",
                "rds:ModifyDBClusterEndpoint",
                "rds:ModifyDBClusterParameterGroup",
                "rds:ModifyDBClusterSnapshotAttribute",
                "rds:ModifyDBInstance",
                "rds:ModifyDBParameterGroup",
                "rds:ModifyDBSubnetGroup",
                "rds:ModifyEventSubscription",
                "rds:ModifyGlobalCluster",
                "rds:PromoteReadReplicaDBCluster",
                "rds:RebootDBInstance",
                "rds:RemoveFromGlobalCluster",
                "rds:RemoveRoleFromDBCluster",
                "rds:RemoveSourceIdentifierFromSubscription",
                "rds:RemoveTagsFromResource",
                "rds:ResetDBClusterParameterGroup",
                "rds:ResetDBParameterGroup",
                "rds:RestoreDBClusterFromSnapshot",
                "rds:RestoreDBClusterToPointInTime",
                "rds:StartDBCluster",
                "rds:StopDBCluster"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Sid": "AllowOtherDepedentPermissions",
            "Effect": "Allow",
            "Action": [
                "cloudwatch:GetMetricStatistics",
                "cloudwatch:ListMetrics",
                "ec2:DescribeAccountAttributes",
```

```
                "ec2:DescribeAvailabilityZones",
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcAttribute",
                "ec2:DescribeVpcs",
                "kms:ListAliases",
                "kms:ListKeyPolicies",
                "kms:ListKeys",
                "kms:ListRetirableGrants",
                "logs:DescribeLogStreams",
                "logs:GetLogEvents",
                "sns:ListSubscriptions",
                "sns:ListTopics",
                "sns:Publish"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Sid": "AllowPassRoleForNeptune",
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:passedToService": "rds.amazonaws.com"
                }
            }
        },
        {
            "Sid": "AllowCreateSLRForNeptune",
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws-eusc:iam::*:role/aws-service-role/
rds.amazonaws.com/AWSServiceRoleForRDS",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": "rds.amazonaws.com"
                }
            }
        },
        {
            "Sid": "AllowDataAccessForNeptune",
```

```
            "Effect": "Allow",
            "Action": [
                "neptune-db:*"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Granting NeptuneConsoleFullAccess using AWS managed policy

The NeptuneConsoleFullAccess managed policy below grants full access to all Neptune actions and resources for administrative purposes, but not for data-access purposes. It also includes additional permissions to simplify Neptune access from the console, including limited IAM and Amazon EC2 (VPC) permissions.

> **Note**
>
> This policy was updated on 2023-11-29 to include permissions needed to interact with Neptune Analytics graphs.
> It was updated on 2022-07-21 to include permissions for global database actions.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowNeptuneCreate",
      "Effect": "Allow",
      "Action": [
        "rds:CreateDBCluster",
        "rds:CreateDBInstance"
      ],
      "Resource": [
        "arn:aws-eusc:rds:*:*:*"
      ],
```

```
      "Condition": {
        "StringEquals": {
          "rds:DatabaseEngine": [
            "graphdb",
            "neptune"
          ]
        }
      }
    },
    {
      "Sid": "AllowManagementPermissionsForRDS",
      "Action": [
        "rds:AddRoleToDBCluster",
        "rds:AddSourceIdentifierToSubscription",
        "rds:AddTagsToResource",
        "rds:ApplyPendingMaintenanceAction",
        "rds:CopyDBClusterParameterGroup",
        "rds:CopyDBClusterSnapshot",
        "rds:CopyDBParameterGroup",
        "rds:CreateDBClusterParameterGroup",
        "rds:CreateDBClusterSnapshot",
        "rds:CreateDBParameterGroup",
        "rds:CreateDBSubnetGroup",
        "rds:CreateEventSubscription",
        "rds:DeleteDBCluster",
        "rds:DeleteDBClusterParameterGroup",
        "rds:DeleteDBClusterSnapshot",
        "rds:DeleteDBInstance",
        "rds:DeleteDBParameterGroup",
        "rds:DeleteDBSubnetGroup",
        "rds:DeleteEventSubscription",
        "rds:DescribeAccountAttributes",
        "rds:DescribeCertificates",
        "rds:DescribeDBClusterParameterGroups",
        "rds:DescribeDBClusterParameters",
        "rds:DescribeDBClusterSnapshotAttributes",
        "rds:DescribeDBClusterSnapshots",
        "rds:DescribeDBClusters",
        "rds:DescribeDBEngineVersions",
        "rds:DescribeDBInstances",
        "rds:DescribeDBLogFiles",
        "rds:DescribeDBParameterGroups",
        "rds:DescribeDBParameters",
        "rds:DescribeDBSecurityGroups",
```

```
        "rds:DescribeDBSubnetGroups",
        "rds:DescribeEngineDefaultClusterParameters",
        "rds:DescribeEngineDefaultParameters",
        "rds:DescribeEventCategories",
        "rds:DescribeEventSubscriptions",
        "rds:DescribeEvents",
        "rds:DescribeOptionGroups",
        "rds:DescribeOrderableDBInstanceOptions",
        "rds:DescribePendingMaintenanceActions",
        "rds:DescribeValidDBInstanceModifications",
        "rds:DownloadDBLogFilePortion",
        "rds:FailoverDBCluster",
        "rds:ListTagsForResource",
        "rds:ModifyDBCluster",
        "rds:ModifyDBClusterParameterGroup",
        "rds:ModifyDBClusterSnapshotAttribute",
        "rds:ModifyDBInstance",
        "rds:ModifyDBParameterGroup",
        "rds:ModifyDBSubnetGroup",
        "rds:ModifyEventSubscription",
        "rds:PromoteReadReplicaDBCluster",
        "rds:RebootDBInstance",
        "rds:RemoveRoleFromDBCluster",
        "rds:RemoveSourceIdentifierFromSubscription",
        "rds:RemoveTagsFromResource",
        "rds:ResetDBClusterParameterGroup",
        "rds:ResetDBParameterGroup",
        "rds:RestoreDBClusterFromSnapshot",
        "rds:RestoreDBClusterToPointInTime"
      ],
      "Effect": "Allow",
      "Resource": [
        "*"
      ]
    },
    {
      "Sid": "AllowOtherDepedentPermissions",
      "Action": [
        "cloudwatch:GetMetricStatistics",
        "cloudwatch:ListMetrics",
        "ec2:AllocateAddress",
        "ec2:AssignIpv6Addresses",
        "ec2:AssignPrivateIpAddresses",
        "ec2:AssociateAddress",
```

```
            "ec2:AssociateRouteTable",
            "ec2:AssociateSubnetCidrBlock",
            "ec2:AssociateVpcCidrBlock",
            "ec2:AttachInternetGateway",
            "ec2:AttachNetworkInterface",
            "ec2:CreateCustomerGateway",
            "ec2:CreateDefaultSubnet",
            "ec2:CreateDefaultVpc",
            "ec2:CreateInternetGateway",
            "ec2:CreateNatGateway",
            "ec2:CreateNetworkInterface",
            "ec2:CreateRoute",
            "ec2:CreateRouteTable",
            "ec2:CreateSecurityGroup",
            "ec2:CreateSubnet",
            "ec2:CreateVpc",
            "ec2:CreateVpcEndpoint",
            "ec2:CreateVpcEndpoint",
            "ec2:DescribeAccountAttributes",
            "ec2:DescribeAccountAttributes",
            "ec2:DescribeAddresses",
            "ec2:DescribeAvailabilityZones",
            "ec2:DescribeAvailabilityZones",
            "ec2:DescribeCustomerGateways",
            "ec2:DescribeInstances",
            "ec2:DescribeNatGateways",
            "ec2:DescribeNetworkInterfaces",
            "ec2:DescribePrefixLists",
            "ec2:DescribeRouteTables",
            "ec2:DescribeSecurityGroupReferences",
            "ec2:DescribeSecurityGroups",
            "ec2:DescribeSecurityGroups",
            "ec2:DescribeSubnets",
            "ec2:DescribeSubnets",
            "ec2:DescribeVpcAttribute",
            "ec2:DescribeVpcAttribute",
            "ec2:DescribeVpcEndpoints",
            "ec2:DescribeVpcs",
            "ec2:DescribeVpcs",
            "ec2:ModifyNetworkInterfaceAttribute",
            "ec2:ModifySubnetAttribute",
            "ec2:ModifyVpcAttribute",
            "ec2:ModifyVpcEndpoint",
            "iam:ListRoles",
```

```
        "kms:ListAliases",
        "kms:ListKeyPolicies",
        "kms:ListKeys",
        "kms:ListRetirableGrants",
        "logs:DescribeLogStreams",
        "logs:GetLogEvents",
        "sns:ListSubscriptions",
        "sns:ListTopics",
        "sns:Publish"
      ],
      "Effect": "Allow",
      "Resource": [
        "*"
      ]
    },
    {
      "Sid": "AllowPassRoleForNeptune",
      "Action": "iam:PassRole",
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "iam:passedToService": "rds.amazonaws.com"
        }
      }
    },
    {
      "Sid": "AllowCreateSLRForNeptune",
      "Action": "iam:CreateServiceLinkedRole",
      "Effect": "Allow",
      "Resource": "arn:aws-eusc:iam::*:role/aws-service-role/rds.amazonaws.com/
AWSServiceRoleForRDS",
      "Condition": {
        "StringLike": {
          "iam:AWSServiceName": "rds.amazonaws.com"
        }
      }
    },
    {
      "Sid": "AllowManagementPermissionsForNeptuneAnalytics",
      "Effect": "Allow",
      "Action": [
        "neptune-graph:CreateGraph",
        "neptune-graph:DeleteGraph",
```

```
        "neptune-graph:GetGraph",
        "neptune-graph:ListGraphs",
        "neptune-graph:UpdateGraph",
        "neptune-graph:ResetGraph",
        "neptune-graph:CreateGraphSnapshot",
        "neptune-graph:DeleteGraphSnapshot",
        "neptune-graph:GetGraphSnapshot",
        "neptune-graph:ListGraphSnapshots",
        "neptune-graph:RestoreGraphFromSnapshot",
        "neptune-graph:CreatePrivateGraphEndpoint",
        "neptune-graph:GetPrivateGraphEndpoint",
        "neptune-graph:ListPrivateGraphEndpoints",
        "neptune-graph:DeletePrivateGraphEndpoint",
        "neptune-graph:CreateGraphUsingImportTask",
        "neptune-graph:GetImportTask",
        "neptune-graph:ListImportTasks",
        "neptune-graph:CancelImportTask"
      ],
      "Resource": [
        "arn:aws-eusc:neptune-graph:*:*:*"
      ]
    },
    {
      "Sid": "AllowPassRoleForNeptuneAnalytics",
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "iam:passedToService": "neptune-graph.amazonaws.com"
        }
      }
    },
    {
      "Sid": "AllowCreateSLRForNeptuneAnalytics",
      "Effect": "Allow",
      "Action": "iam:CreateServiceLinkedRole",
      "Resource": "arn:aws-eusc:iam::*:role/aws-service-role/neptune-
graph.amazonaws.com/AWSServiceRoleForNeptuneGraph",
      "Condition": {
        "StringLike": {
          "iam:AWSServiceName": "neptune-graph.amazonaws.com"
        }
      }
```

```
    }
  ]
}
```

Customizing access to Amazon Neptune resources using IAM condition context keys

You can specify conditions in IAM policies that control access to Neptune management actions and resources. The policy statement then takes effect only when the conditions are true.

For example, you might want a policy statement to take effect only after a specific date, or allow access only when a specific value is present in the API request.

To express conditions, you use predefined condition keys in the `Condition` element of a policy statement, together with IAM condition policy operators such as equals or less than.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM Policy Elements: Variables and Tags in the *IAM User Guide*.

The data type of a condition key determines which condition operators you can use to compare values in the request with the values in the policy statement. If you use a condition operator that is not compatible with that data type, the match always fails and the policy statement never applies.

IAM condition keys for Neptune administrative policy statements

- Global condition keys – You can use most AWS global condition keys in Neptune administrative policy statements.
- Service-specific condition keys – These are keys that are defined for specific AWS services. The ones that Neptune supports for administrative policy statements are listed in IAM condition keys for administering Amazon Neptune.

IAM condition keys for Neptune data-access policy statements

- Global condition keys – The subset of these keys that Neptune supports in data-access policy statements is listed in AWS global condition context keys supported by Neptune in data-access policy statements.
- Service-specific condition keys that Neptune defines for data-access policy statements are listed in Condition Keys.

Creating custom IAM policy statements to administer Amazon Neptune

Administrative policy statements let you control what an IAM user can do to manage a Neptune database.

A Neptune administrative policy statement grants access to one or more administrative actions and administrative resources that Neptune supports. You can also use Condition Keys to make the administrative permissions more specific.

> **Note**
>
> Because Neptune shares functionality with Amazon RDS, administrative actions, resources, and service-specific condition keys in administrative policy statements use an `rds:` prefix by design.

Topics

- IAM actions for administering Amazon Neptune
- IAM resource types for administering Amazon Neptune
- IAM condition keys for administering Amazon Neptune
- Creating IAM administrative policy statements for Amazon Neptune

IAM actions for administering Amazon Neptune

You can use the administrative actions listed below in the `Action` element of an IAM policy statement to control access to the Neptune management APIs. When you use an action in a policy, you usually allow or deny access to the API operation or CLI command with the same name. However, in some cases, a single action controls access to more than one operation. Alternatively, some operations require several different actions.

The `Resource type` field in the list below indicates whether each action supports resource-level permissions. If there is no value in this field, you must specify all resources ("*") in the `Resource` element of your policy statement. If the column includes a resource type, then you can specify a resource ARN of that type in a statement with that action. Neptune administrative resource types are listed on this page.

Required resources are indicated in the list below with an asterisk (*). If you specify a resource-level permission ARN in a statement using this action, then it must be of this type. Some actions support multiple resource types. If a resource types is optional (in other words, is not marked with an asterisk), then you do not have to include it.

For more information about the fields listed here, see action table in the IAM User Guide.

rds:AddRoleToDBCluster

`AddRoleToDBCluster` associates an IAM role with a Neptune DB cluster.

Access level: `Write`.

Dependent actions: `iam:PassRole`.

Resource type: cluster (required).

rds:AddSourceIdentifierToSubscription

`AddSourceIdentifierToSubscription` adds a source identifier to an existing Neptune event notification subscription.

Access level: `Write`.

Resource type: es (required).

rds:AddTagsToResource

`AddTagsToResource` associates an IAM role with a Neptune DB cluster.

Access level: `Write`.

Resource types:

- db
- es

- pg
- cluster-snapshot
- subgrp

Condition Keys:

- aws:RequestTag/*tag-key*
- aws:TagKeys

rds:ApplyPendingMaintenanceAction

`ApplyPendingMaintenanceAction` applies a pending maintenance action to a resource.

Access level: `Write`.

Resource type: db (required).

rds:CopyDBClusterParameterGroup

`CopyDBClusterParameterGroup` copies the specified DB cluster parameter group.

Access level: `Write`.

Resource type: cluster-pg (required).

rds:CopyDBClusterSnapshot

`CopyDBClusterSnapshot` copies a snapshot of a DB cluster.

Access level: `Write`.

Resource type: cluster-snapshot (required).

rds:CopyDBParameterGroup

`CopyDBParameterGroup` copies the specified DB parameter group.

Access level: `Write`.

Resource type: pg (required).

rds:CreateDBCluster

`CreateDBCluster` creates a new Neptune DB cluster.

Access level: `Tagging`.

Dependent actions: `iam:PassRole`.

Resource types:

- cluster (required).
- cluster-pg (required).
- subgrp (required).

Condition Keys:

- aws:RequestTag/*tag-key*
- aws:TagKeys
- neptune-rds_DatabaseEngine

rds:CreateDBClusterParameterGroup

`CreateDBClusterParameterGroup` creates a new DB cluster parameter group.

Access level: `Tagging`.

Resource type: cluster-pg (required).

Condition Keys:

- aws:RequestTag/*tag-key*
- aws:TagKeys

rds:CreateDBClusterSnapshot

`CreateDBClusterSnapshot` creates a snapshot of a DB cluster.

Access level: `Tagging`.

Resource types:

- cluster (required).
- cluster-snapshot (required).

Condition Keys:

- aws:RequestTag/*tag-key*
- aws:TagKeys

rds:CreateDBInstance

`CreateDBInstance` creates a new DB instance.

Access level: `Tagging`.

Dependent actions: `iam:PassRole`.

Resource types:

- db (required).
- pg (required).
- subgrp (required).

Condition Keys:

- aws:RequestTag/*tag-key*
- aws:TagKeys

rds:CreateDBParameterGroup

`CreateDBParameterGroup` creates a new DB parameter group.

Access level: `Tagging`.

Resource type: pg (required).

Condition Keys:

- aws:RequestTag/*tag-key*
- aws:TagKeys

rds:CreateDBSubnetGroup

`CreateDBSubnetGroup` creates a new DB subnet group.

Access level: `Tagging`.

Resource type: subgrp (required).

Condition Keys:

- aws:RequestTag/*tag-key*
- aws:TagKeys

rds:CreateEventSubscription

`CreateEventSubscription` creates a Neptune event notification subscription.

Access level: `Tagging`.

Resource type: es (required).

Condition Keys:

- aws:RequestTag/*tag-key*
- aws:TagKeys

rds:DeleteDBCluster

`DeleteDBCluster` deletes an existing Neptune DB cluster.

Access level: `Write`.

Resource types:

- cluster (required).
- cluster-snapshot (required).

rds:DeleteDBClusterParameterGroup

`DeleteDBClusterParameterGroup` deletes a specified DB cluster parameter group.

Access level: `Write`.

Resource type: cluster-pg (required).

rds:DeleteDBClusterSnapshot

`DeleteDBClusterSnapshot` deletes a DB cluster snapshot.

Access level: `Write`.

Resource type: cluster-snapshot (required).

rds:DeleteDBInstance

`DeleteDBInstance` deletes a specified DB instance.

Access level: `Write`.

Resource type: db (required).

rds:DeleteDBParameterGroup

`DeleteDBParameterGroup` deletes a specified DBParameterGroup.

Access level: `Write`.

Resource type: pg (required).

rds:DeleteDBSubnetGroup

`DeleteDBSubnetGroup` deletes a DB subnet group.

Access level: `Write`.

Resource type: subgrp (required).

rds:DeleteEventSubscription

`DeleteEventSubscription` deletes an event notification subscription.

Access level: `Write`.

Resource type: es (required).

rds:DescribeDBClusterParameterGroups

`DescribeDBClusterParameterGroups` returns a list of DBClusterParameterGroup descriptions.

Access level: `List`.

Resource type: cluster-pg (required).

rds:DescribeDBClusterParameters

`DescribeDBClusterParameters` returns the detailed parameter list for a particular DB cluster parameter group.

Access level: `List`.

Resource type: cluster-pg (required).

rds:DescribeDBClusterSnapshotAttributes

`DescribeDBClusterSnapshotAttributes` returns a list of DB cluster snapshot attribute names and values for a manual DB cluster snapshot.

Access level: `List`.

Resource type: cluster-snapshot (required).

rds:DescribeDBClusterSnapshots

`DescribeDBClusterSnapshots` returns information about DB cluster snapshots.

Access level: `Read`.

rds:DescribeDBClusters

`DescribeDBClusters` returns information about a provisioned Neptune DB cluster.

Access level: `List`.

Resource type: cluster (required).

rds:DescribeDBEngineVersions

`DescribeDBEngineVersions` returns a list of the available DB engines.

Access level: `List`.

Resource type: pg (required).

rds:DescribeDBInstances

`DescribeDBInstances` returns information about DB instances.

Access level: `List`.

Resource type: es (required).

rds:DescribeDBParameterGroups

`DescribeDBParameterGroups` returns a list of DBParameterGroup descriptions.

Access level: `List`.

Resource type: pg (required).

rds:DescribeDBParameters

`DescribeDBParameters` returns a detailed parameter list for a particular DB parameter group.

Access level: `List`.

Resource type: pg (required).

rds:DescribeDBSubnetGroups

`DescribeDBSubnetGroups` returns a list of DBSubnetGroup descriptions.

Access level: `List`.

Resource type: subgrp (required).

rds:DescribeEventCategories

`DescribeEventCategories` returns a list of categories for all event source types, or, if specified, for a specified source type.

Access level: `List`.

rds:DescribeEventSubscriptions

`DescribeEventSubscriptions` lists all the subscription descriptions for a customer account.

Access level: `List`.

Resource type: es (required).

rds:DescribeEvents

`DescribeEvents` returns events related to DB instances, DB security groups, and DB parameter groups for the past 14 days.

Access level: `List`.

Resource type: es (required).

rds:DescribeOrderableDBInstanceOptions

`DescribeOrderableDBInstanceOptions` returns a list of orderable DB instance options for the specified engine.

Access level: `List`.

rds:DescribePendingMaintenanceActions

`DescribePendingMaintenanceActions` returns a list of resources (for example, DB instances) that have at least one pending maintenance action.

Access level: `List`.

Resource type: db (required).

rds:DescribeValidDBInstanceModifications

`DescribeValidDBInstanceModifications` lists available modifications you can make to your DB instance.

Access level: `List`.

Resource type: db (required).

rds:FailoverDBCluster

`FailoverDBCluster` forces a failover for a DB cluster.

Access level: `Write`.

Resource type: cluster (required).

rds:ListTagsForResource

`ListTagsForResource` lists all tags on a Neptune resource.

Access level: `Read`.

Resource types:

- cluster-snapshot
- db
- es
- pg
- subgrp

rds:ModifyDBCluster

`ModifyDBCluster`

Modifies a setting for a Neptune DB cluster.

Access level: `Write`.

Dependent actions: `iam:PassRole`.

Resource types:

- cluster (required).
- cluster-pg (required).

rds:ModifyDBClusterParameterGroup

`ModifyDBClusterParameterGroup` modifies the parameters of a DB cluster parameter group.

Access level: `Write`.

Resource type: cluster-pg (required).

rds:ModifyDBClusterSnapshotAttribute

`ModifyDBClusterSnapshotAttribute` adds an attribute and values to, or removes an attribute and values from, a manual DB cluster snapshot.

Access level: `Write`.

Resource type: cluster-snapshot (required).

rds:ModifyDBInstance

`ModifyDBInstance` modifies settings for a DB instance.

Access level: `Write`.

Dependent actions: `iam:PassRole`.

Resource types:

- db (required).
- pg (required).

rds:ModifyDBParameterGroup

`ModifyDBParameterGroup` modifies the parameters of a DB parameter group.

Access level: `Write`.

Resource type: pg (required).

rds:ModifyDBSubnetGroup

`ModifyDBSubnetGroup` modifies an existing DB subnet group.

Access level: `Write`.

Resource type: subgrp (required).

rds:ModifyEventSubscription

`ModifyEventSubscription` modifies an existing Neptune event notification subscription.

Access level: `Write`.

Resource type: es (required).

rds:RebootDBInstance

`RebootDBInstance` restarts the database engine service for the instance.

Access level: `Write`.

Resource type: db (required).

rds:RemoveRoleFromDBCluster

`RemoveRoleFromDBCluster` disassociates an AWS Identity and Access Management (IAM) role from an Amazon Neptune DB cluster.

Access level: `Write`.

Dependent actions: `iam:PassRole`.

Resource type: cluster (required).

rds:RemoveSourceIdentifierFromSubscription

`RemoveSourceIdentifierFromSubscription` removes a source identifier from an existing Neptune event notification subscription.

Access level: `Write`.

Resource type: es (required).

rds:RemoveTagsFromResource

`RemoveTagsFromResource` removes metadata tags from a Neptune resource.

Access level: `Tagging`.

Resource types:

- cluster-snapshot
- db
- es
- pg

- subgrp

Condition Keys:

- aws:RequestTag/*tag-key*
- aws:TagKeys

rds:ResetDBClusterParameterGroup

`ResetDBClusterParameterGroup` modifies the parameters of a DB cluster parameter group to the default value.

Access level: `Write`.

Resource type: cluster-pg (required).

rds:ResetDBParameterGroup

`ResetDBParameterGroup` modifies the parameters of a DB parameter group to the engine/system default value.

Access level: `Write`.

Resource type: pg (required).

rds:RestoreDBClusterFromSnapshot

`RestoreDBClusterFromSnapshot` creates a new DB cluster from a DB cluster snapshot.

Access level: `Write`.

Dependent actions: `iam:PassRole`.

Resource types:

- cluster (required).
- cluster-snapshot (required).

Condition Keys:

- aws:RequestTag/*tag-key*

- aws:TagKeys

rds:RestoreDBClusterToPointInTime

`RestoreDBClusterToPointInTime` restores a DB cluster to an arbitrary point in time.

Access level: `Write`.

Dependent actions: `iam:PassRole`.

Resource types:

- cluster (required).
- subgrp (required).

Condition Keys:

- aws:RequestTag/*tag-key*
- aws:TagKeys

rds:StartDBCluster

`StartDBCluster` starts the specified DB cluster.

Access level: `Write`.

Resource type: cluster (required).

rds:StopDBCluster

`StopDBCluster` stops the specified DB cluster.

Access level: `Write`.

Resource type: cluster (required).

IAM resource types for administering Amazon Neptune

Neptune supports the resource types in the following table for use in the `Resource` element of IAM administration policy statements. For more information about the `Resource` element, see IAM JSON Policy Elements: Resource.

The list of Neptune administration actions identifies the resource types that can be specified with each action. A resource type also determines which condition keys you can include in a policy, as specified in the last column of the table below.

The ARN column in the table below specifies the Amazon Resource Name (ARN) format that you must use to reference resources of this type. The portions that are preceded by a `$` must be replaced by the actual values for your scenario. For example, if you see `$user-name` in an ARN, you must replace that string either with the actual IAM user's name or with a policy variable that contains an IAM user name. For more information about ARNs, see IAM ARNs, and Working with administrative ARNs in Amazon Neptune.

The `Condition Keys` column specifies condition context keys that you can include in an IAM policy statement only when both this resource and a compatible supporting action are included in the statement.

Resource Types	ARN	Condition Keys
`cluster` (a DB cluster)	`arn:`*`partition`*` :rds:`*`region`*`:`*`account-id`*` :cluster: `*`instance-name`*	aws:ResourceTag/*tag-key* rds:cluster-tag/*tag-key*
`cluster-pg` (a DB cluster parameter group)	`arn:`*`partition`*` :rds:`*`region`*`:`*`account-id`*` :cluster-pg: `*`neptune-DBClusterParameterGroupName`*	aws:ResourceTag/*tag-key*
`cluster-snapshot` (a DB cluster snapshot)	`arn:`*`partition`*` :rds:`*`region`*`:`*`account-id`*` :cluster-snapshot: `*`neptune-DBClusterSnapshotName`*	aws:ResourceTag/*tag-key* rds:cluster-snapshot-tag/*tag-key*
db	`arn:`*`partition`*` :rds:`*`region`*`:`*`account-id`*` :db:`*`neptune-DbInstanceName`*	aws:ResourceTag/*tag-key*

Resource Types	ARN	Condition Keys
(a DB instance)		rds:DatabaseClass rds:DatabaseEngine rds:db-tag/*tag-key*
es (an event subscription)	arn:*partition* :rds:*region*:*account-id* :es:*neptune-CustSubscriptionId*	aws:ResourceTag/*tag-key* rds:es-tag/*tag-key*
pg (a DB parameter group)	arn:*partition* :rds:*region*:*account-id* :pg:*neptune-ParameterGroupName*	aws:ResourceTag/*tag-key* rds:pg-tag/*tag-key*
subgrp (a DB subnet group)	arn:*partition* :rds:*region*:*account-id* :subgrp:*neptune-DBSubnetGroupName* }	aws:ResourceTag/*tag-key* rds:subgrp-tag/*tag-key*

IAM condition keys for administering Amazon Neptune

Using condition keys, you can specify conditions in an IAM policy statement so that the statement takes effect only when the conditions are true. The condition keys that you can use in Neptune administrative policy statements fall into the following categories:

- Global condition keys – These are defined by AWS for general use with AWS services. Most can be used in Neptune administrative policy statements.
- Administrative resource property condition keys – These keys, listed below, are based on properties of administrative resources.
- Tag-based access condition keys – These keys, listed below, are based on AWS tags attached to administrative resources.

Neptune administrative resource property condition keys

Condition keys	Description	Type
`rds:DatabaseClass`	Filters access by the type of DB instance class.	String
`rds:DatabaseEngine`	Filters access by the database engine. For possible values refer to the engine parameter in CreateDBInstance API	String
`rds:DatabaseName`	Filters access by the user-defined name of the database on the DB instance	String
`rds:EndpointType`	Filters access by the type of the endpoint. One of: READER, WRITER, CUSTOM	String
`rds:Vpc`	Filters access by the value that specifies whether the DB instance runs in an Amazon Virtual Private Cloud (Amazon VPC). To indicate that the DB instance runs in an Amazon VPC, specify `true`.	Boolean

Administrative tag-based condition keys

Amazon Neptune supports specifying conditions in an IAM policy using custom tags, to control access to Neptune through the Management API reference.

For example, if you add a tag named `environment` to your DB instances, with values such as `beta`, `staging`, and `production`, you can then create a policy that restricts access to the instances based on the value of that tag.

> **Important**
>
> If you manage access to your Neptune resources using tagging, be sure to secure access to the tags. You can restrict access to the tags by creating policies for the `AddTagsToResource` and `RemoveTagsFromResource` actions.
> For example, you could use the following policy to deny users the ability to add or remove tags for all resources. Then, you could create policies to allow specific users to add or remove tags.

JSON

```
{ "Version":"2012-10-17",
  "Statement":[
    { "Sid": "DenyTagUpdates",
      "Effect": "Deny",
      "Action": [
        "rds:AddTagsToResource",
        "rds:RemoveTagsFromResource"
      ],
      "Resource":"*"
    }
  ]
}
```

The following tag-based condition keys only work with administrative resources in administrative policy statements.

Tag-based administrative condition keys

Condition keys	Description	Type
aws:RequestTag/${TagKey}	Filters access based on the presence of tag key-value pairs in the request.	String
aws:ResourceTag/${TagKey}	Filters access based on tag key-value pairs attached to the resource.	String
aws:TagKeys	Filters access based on the presence of tag keys in the request.	String
rds:cluster-pg-tag/${TagKey}	Filters access by the tag attached to a DB cluster parameter group.	String

Condition keys	Description	Type
`rds:cluster-snapshot-tag/${TagKey}`	Filters access by the tag attached to a DB cluster snapshot.	String
`rds:cluster-tag/${TagKey}`	Filters access by the tag attached to a DB cluster.	String
`rds:db-tag/${TagKey}`	Filters access by the tag attached to a DB instance.	String
`rds:es-tag/${TagKey}`	Filters access by the tag attached to an event subscription.	String
`rds:pg-tag/${TagKey}`	Filters access by the tag attached to a DB parameter group.	String
`rds:req-tag/${TagKey}`	Filters access by the set of tag keys and values that can be used to tag a resource.	String
`rds:secgrp-tag/${TagKey}`	Filters access by the tag attached to a DB security group.	String
`rds:snapshot-tag/${TagKey}`	Filters access by the tag attached to a DB snapshot.	String
`rds:subgrp-tag/${TagKey}`	Filters access by the tag attached to a DB subnet group	String

Creating IAM administrative policy statements for Amazon Neptune

General administrative policy examples

The following examples show how to create Neptune administrative policies that grant permissions to take various management actions on a DB cluster.

Policy that prevents an IAM user from deleting a specified DB instance

The following is an example policy that prevents an IAM user from deleting a specified Neptune DB instance:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "DenyDeleteOneInstance",
      "Effect": "Deny",
      "Action": "rds:DeleteDBInstance",
      "Resource": "arn:aws-eusc:rds:us-west-2:123456789012:db:my-instance-name"
    }
  ]
}
```

Policy that grants permission to create new DB instances

The following is an example policy that allows an IAM user to create DB instances in a specified Neptune DB cluster:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowCreateInstance",
      "Effect": "Allow",
      "Action": "rds:CreateDBInstance",
      "Resource": "arn:aws-eusc:rds:us-west-2:123456789012:cluster:my-cluster"
```

```
    }
  ]
}
```

Policy that grants permission to create new DB instances that use a specific DB parameter group

The following is an example policy that allows an IAM user to create DB instances in a specified DB cluster (here `us-west-2`) in a specified Neptune DB cluster using only a specified DB parameter group.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowCreateInstanceWithPG",
      "Effect": "Allow",
      "Action": "rds:CreateDBInstance",
      "Resource": [
        "arn:aws-eusc:rds:us-west-2:123456789012:cluster:my-cluster",
        "arn:aws-eusc:rds:us-west-2:123456789012:pg:my-instance-pg"
      ]
    }
  ]
}
```

Policy that grants permission to describe any resource

The following is an example policy that allows an IAM user to describe any Neptune resource.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowDescribe",
```

```
            "Effect": "Allow",
            "Action": "rds:Describe*",
            "Resource": "*"
        }
    ]
}
```

Tag-based administrative policy examples

The following examples show how to create Neptune administrative policies that tags to filter permissions for various management actions on a DB cluster.

Example 1: Grant permission for actions on a resource using a custom tag that can take multiple values

The policy below allows use of the `ModifyDBInstance`, `CreateDBInstance` or `DeleteDBInstance` API on any DB instance that has the `env` tag set to either `dev` or `test`:

JSON

```
{ "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowDevTestAccess",
      "Effect": "Allow",
      "Action": [
        "rds:ModifyDBInstance",
        "rds:CreateDBInstance",
        "rds:DeleteDBInstance"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "rds:db-tag/env": [
              "dev",
              "test"
          ],
          "rds:DatabaseEngine": "neptune"
        }
      }
    }
```

```
  ]
}
```

Example 2: Limit the set of tag keys and values that can be used to tag a resource

This policy uses a `Condition` key to allow a tag that has the key `env` and a value of `test`, `qa`, or `dev` to be added to a resource:

JSON

```
{ "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowTagAccessForDevResources",
      "Effect": "Allow",
      "Action": [
        "rds:AddTagsToResource",
        "rds:RemoveTagsFromResource"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "rds:req-tag/env": [
            "test",
            "qa",
            "dev"
          ],
          "rds:DatabaseEngine": "neptune"
        }
      }
    }
  ]
}
```

Example 3: Allow full access to Neptune resources based on `aws:ResourceTag`

The following policy is similar to the first example above, but uses the `aws:ResourceTag` instead:

JSON

```
{ "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowFullAccessToDev",
      "Effect": "Allow",
      "Action": [
        "rds:*"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/env": "dev",
          "rds:DatabaseEngine": "neptune"
        }
      }
    }
  ]
}
```

Creating custom IAM policy statements to access data in Amazon Neptune

Neptune data-access policy statements use data-access actions, resources, and condition keys, all of which are preceded by a `neptune-db:` prefix.

Topics

- Using query actions in Neptune data-access policy statements
- IAM actions for data access in Amazon Neptune
- IAM resource types for accessing data in Amazon Neptune
- IAM condition keys for accessing data in Amazon Neptune
- Creating IAM data-access policies in Amazon Neptune
- Tag-based access control for Amazon Neptune data-plane operations

Using query actions in Neptune data-access policy statements

There are three Neptune query actions that can be used in data-access policy statements, namely `ReadDataViaQuery`, `WriteDataViaQuery`, and `DeleteDataViaQuery`. A particular query may need permissions to perform more than one of these actions, and it may not always be obvious what combination of these actions must be permitted in order to run a query.

Before running a query, Neptune determines the permissions needed to run each step of the query, and combines these into the full set of permissions that the query requires. Note that this full set of permissions includes all actions that the query *might* perform, which is not necessarily the set of actions that the query actually will perform when it runs over your data.

This means that to permit a given query to run, you must provide permissions for every action that the query could possibly perform, whether or not it actually performs them.

Here are some sample Gremlin queries where this is explained in more detail:

- ```
  g.V().count()
  ```

  `g.V()` and `count()` only require read access, so the query as a whole only requires `ReadDataViaQuery` access.
- ```
  g.addV()
  ```

  `addV()` needs to check whether a vertex with a given ID exists or not before inserting a new one. This means that it requires both `ReadDataViaQuery` and `WriteDataViaQuery` access.
- ```
  g.V('1').as('a').out('created').addE('createdBy').to('a')
  ```

  `g.V('1').as('a')` and `out('created')` only require read access, but `addE().from('a')` requires both read and write access because `addE()` needs to read the `from` and `to` vertices and check whether an edge with the same ID already exists before adding a new one. The query as a whole therefore needs both `ReadDataViaQuery` and `WriteDataViaQuery` access.
- ```
  g.V().drop()
  ```

  `g.V()` only requires read access. `drop()` needs both read and delete access because it needs to read a vertex or edge before deleting it, so the query as a whole requires both `ReadDataViaQuery` and `DeleteDataViaQuery` access.

- ```
  g.V('1').property(single, 'key1', 'value1')
  ```

  `g.V('1')` only requires read access, but `property(single, 'key1', 'value1')` requires read, write, and delete access. Here, the `property()` step inserts the key and value if they do not already exist in the vertex, but if they do already exist, it deletes the existing property value and inserts a new value in its place. Therefore, the query as a whole requires `ReadDataViaQuery`, `WriteDataViaQuery`, and `DeleteDataViaQuery` access.

  Any query that contains a `property()` step will need `ReadDataViaQuery`, `WriteDataViaQuery`, and `DeleteDataViaQuery` permissions.

Here are some openCypher examples:

- ```
  MATCH (n)
  RETURN n
  ```

  This query reads all nodes in the database and returns them, which only requires `ReadDataViaQuery` access.

- ```
  MATCH (n:Person)
  SET n.dept = 'AWS'
  ```

  This query requires `ReadDataViaQuery`, `WriteDataViaQuery`, and `DeleteDataViaQuery` access. It reads all nodes with the label 'Person' and either adds a new property with the key `dept` and value `AWS` to them, or if the `dept` property already exists, it deletes the old value and inserts `AWS` instead. Also, if the value to be set is `null`, `SET` deletes the property altogether.

  Because the `SET` clause may in some cases need to delete an existing value, it **always** needs `DeleteDataViaQuery` permissions as well as `ReadDataViaQuery` and `WriteDataViaQuery` permissions.

- ```
  MATCH (n:Person)
  DETACH DELETE n
  ```

  This query needs `ReadDataViaQuery` and `DeleteDataViaQuery` permissions. It finds all the nodes with The label `Person` and deletes them along with the edges connected to those nodes and any associated labels and properties.

- ```
  MERGE (n:Person {name: 'John'})-[:knows]->(:Person {name: 'Peter'})
  RETURN n
  ```

  This query needs `ReadDataViaQuery` and `WriteDataViaQuery` permissions. The `MERGE` clause either matches a specified pattern or creates it. Since, a write can occur if the pattern is not matched, write permissions are needed as well as read permissions.

IAM actions for data access in Amazon Neptune

Note that Neptune data-access actions have the prefix `neptune-db:`, whereas administrative actions in Neptune have the prefix `rds:`.

The Amazon Resource Name (ARN) for a data resource in IAM is not the same as the ARN assigned to a cluster on creation. You must construct the ARN as shown in Specifying data resources. Such data resource ARNs can use wildcards to include multiple resources.

Data-access policy statements can also include the neptune-db:QueryLanguage condition key to restrict access by query language.

Starting with Release: 1.2.0.0 (2022-07-21), Neptune supports restricting permissions to one or more specific Neptune actions. This provides more granular access control than was previously possible.

> **Important**
>
> - Changes to an IAM policy take up to 10 minutes to apply to the specified Neptune resources.
> - IAM policies that are applied to a Neptune DB cluster apply to all instances in that cluster.

Query-based data-access actions

> **Note**
>
> It isn't always obvious what permissions are needed to run a given query, because queries can potentially take more than one action depending on the data that they process. See Using query actions for more information.
  ```

`neptune-db:ReadDataViaQuery`

`ReadDataViaQuery` allows the user to read data from the Neptune database by submitting queries.

Action groups: read-only, read-write.

Action context keys: `neptune-db:QueryLanguage`.

Required resources: database.

`neptune-db:WriteDataViaQuery`

`WriteDataViaQuery` allows the user to write data to the Neptune database by submitting queries.

Action groups: read-write.

Action context keys: `neptune-db:QueryLanguage`.

Required resources: database.

`neptune-db:DeleteDataViaQuery`

`DeleteDataViaQuery` allows the user to delete data from the Neptune database by submitting queries.

Action groups: read-write.

Action context keys: `neptune-db:QueryLanguage`.

Required resources: database.

`neptune-db:GetQueryStatus`

`GetQueryStatus` allows the user to check the status of all active queries.

Action groups: read-only, read-write.

Action context keys: `neptune-db:QueryLanguage`.

Required resources: database.

`neptune-db:GetStreamRecords`

`GetStreamRecords` allows the user to fetch stream records from Neptune.

Action groups: read-write.

Action context keys: `neptune-db:QueryLanguage`.

Required resources: database.

`neptune-db:CancelQuery`

`CancelQuery` allows the user to to cancel a query.

Action groups: read-write.

Required resources: database.

General data-access actions

`neptune-db:GetEngineStatus`

`GetEngineStatus` allows the user to check the status of the Neptune engine.

Action groups: read-only, read-write.

Required resources: database.

`neptune-db:GetStatisticsStatus`

`GetStatisticsStatus` allows the user to check the status of statistics being collected for the database.

Action groups: read-only, read-write.

Required resources: database.

`neptune-db:GetGraphSummary`

`GetGraphSummary` The graph summary API enables you to retrieve a read-only summary of your graph.

Action groups: read-only, read-write.

Required resources: database.

`neptune-db:ManageStatistics`

`ManageStatistics` allows the user to to manage the collection of statistics for the database.

Action groups: read-write.

Required resources: database.

`neptune-db:DeleteStatistics`

`DeleteStatistics` allows the user to delete all the statistics in the database.

Action groups: read-write.

Required resources: database.

`neptune-db:ResetDatabase`

`ResetDatabase` allows the user to get the token needed for a reset and to reset the Neptune database.

Action groups: read-write.

Required resources: database.

Bulk-loader data-access actions

`neptune-db:StartLoaderJob`

`StartLoaderJob` allows the user to start a bulk-loader job.

Action groups: read-write.

Required resources: database.

`neptune-db:GetLoaderJobStatus`

`GetLoaderJobStatus` allows the user to check the status of a bulk-loader job.

Action groups: read-only, read-write.

Required resources: database.

`neptune-db:ListLoaderJobs`

`ListLoaderJobs` allows the user to list all the bulk-loader jobs.

Action groups: list-only, read-only, read-write.

Required resources: database.

`neptune-db:CancelLoaderJob`

`CancelLoaderJob` allows the user to cancel a loader job.

Action groups: read-write.

Required resources: database.

Machine-learning data-access actions

`neptune-db:StartMLDataProcessingJob`

`StartMLDataProcessingJob` allows a user to start a Neptune ML data processing job.

Action groups: read-write.

Required resources: database.

`neptune-db:StartMLModelTrainingJob`

`StartMLModelTrainingJob` allows a user to start an ML model training job.

Action groups: read-write.

Required resources: database.

`neptune-db:StartMLModelTransformJob`

`StartMLModelTransformJob` allows a user to start an ML model transform job.

Action groups: read-write.

Required resources: database.

`neptune-db:CreateMLEndpoint`

`CreateMLEndpoint` allows a user to create a Neptune ML endpoint.

Action groups: read-write.

Required resources: database.

`neptune-db:GetMLDataProcessingJobStatus`

`GetMLDataProcessingJobStatus` allows a user to check the status of a Neptune ML data processing job.

Action groups: read-only, read-write.

Required resources: database.

`neptune-db:GetMLModelTrainingJobStatus`

`GetMLModelTrainingJobStatus` allows a user to check the status of a Neptune ML model training job.

Action groups: read-only, read-write.

Required resources: database.

`neptune-db:GetMLModelTransformJobStatus`

`GetMLModelTransformJobStatus` allows a user to check the status of a Neptune ML model transform job.

Action groups: read-only, read-write.

Required resources: database.

`neptune-db:GetMLEndpointStatus`

`GetMLEndpointStatus` allows a user to check the status of a Neptune ML endpoint.

Action groups: read-only, read-write.

Required resources: database.

`neptune-db:ListMLDataProcessingJobs`

`ListMLDataProcessingJobs` allows a user to list all the Neptune ML data processing jobs.

Action groups: list-only, read-only, read-write.

Required resources: database.

`neptune-db:ListMLModelTrainingJobs`

`ListMLModelTrainingJobs` allows a user to list all the Neptune ML model training jobs.

Action groups: list-only, read-only, read-write.

Required resources: database.

`neptune-db:ListMLModelTransformJobs`

`ListMLModelTransformJobs` allows a user to list all the ML model transform jobs.

Action groups: list-only, read-only, read-write.

Required resources: database.

`neptune-db:ListMLEndpoints`

`ListMLEndpoints` allows a user to list all the Neptune ML endpoints.

Action groups: list-only, read-only, read-write.

Required resources: database.

`neptune-db:CancelMLDataProcessingJob`

`CancelMLDataProcessingJob` allows a user to cancel a Neptune ML data processing job.

Action groups: read-write.

Required resources: database.

`neptune-db:CancelMLModelTrainingJob`

`CancelMLModelTrainingJob` allows a user to cancel a Neptune ML model training job.

Action groups: read-write.

Required resources: database.

`neptune-db:CancelMLModelTransformJob`

`CancelMLModelTransformJob` allows a user to cancel a Neptune ML model transform job.

Action groups: read-write.

Required resources: database.

`neptune-db:DeleteMLEndpoint`

`DeleteMLEndpoint` allows a user to delete a Neptune ML endpoint.

Action groups: read-write.

Required resources: database.

Export data-access actions

`neptune-db:StartExportJob`

`StartExportJob` allows the user to start an export job.

Action groups: read-write.

Required resources: database.

`neptune-db:GetExportJobStatus`

`GetExportJobStatus` allows the user to get the status of an export job.

Action groups: read-only, read-write.

Required resources: database.

`neptune-db:ListExportJobs`

`ListExportJobs` allows the user to list all export jobs.

Action groups: list-only, read-only, read-write.

Required resources: database.

`neptune-db:CancelExportJob`

`CancelExportJob` allows the user to cancel an export job.

Action groups: read-write.

Required resources: database.

IAM resource types for accessing data in Amazon Neptune

Data resources, like data actions, have a `neptune-db:` prefix.

In a Neptune data-access policy, you specify the DB cluster that you are giving access to in an ARN with the following format:

```
arn:aws:neptune-db:region:account-id:cluster-resource-id/*
```

Such a resource ARN contains the following parts:

- *`region`* is the AWS Region for the Amazon Neptune DB cluster.
- *`account-id`* is the AWS account number for the DB cluster.
- *`cluster-resource-id`* is a resource id for the DB cluster.

> **Important**
>
> The `cluster-resource-id` is different from the cluster identifier. To find a cluster resource ID in the Neptune AWS Management Console, look in the **Configuration** section for the DB cluster in question.

IAM condition keys for accessing data in Amazon Neptune

Using condition keys, you can specify conditions in an IAM policy statement so that the statement takes effect only when the conditions are true.

The condition keys that you can use in Neptune data-access policy statements fall into the following categories:

- Global condition keys – The subset of AWS global condition keys that Neptune supports in data-access policy statements is listed below.
- Service-specific condition keys – These are keys defined by Neptune specifically for use in data-access policy statements. At present there is only one, neptune-db:QueryLanguage, which grants access only if a specific query language is being used.

AWS global condition context keys supported by Neptune in data-access policy statements

The following table lists the subset of AWS global condition context keys that Amazon Neptune supports for use in data-access policy statements:

Global condition keys that you can use in data-access policy statements

Condition Keys	Description	Type
aws:CurrentTime	Filters access by the current date and time of the request.	String
aws:EpochTime	Filters access by date and time of the request expressed as a UNIX epoch value.	Numeric
aws:PrincipalAccount	Filters access by the account to which the requesting principal belongs.	String
aws:PrincipalArn	Filters access by the ARN of the principal that made the request.	String
aws:PrincipalIsAWSService	Allows access only if the call is being made directly by an AWS service principal.	Boolean
aws:PrincipalOrgID	Filters access by the identifier of the organization in AWS Organizations to which the requesting principal belongs.	String
aws:PrincipalOrgPaths	Filters access by the AWS Organizations path for the principal who is making the request.	String
aws:PrincipalTag	Filters access by a tag attached to the principal making the request.	String
aws:PrincipalType	Filters access by the type of principal making the request.	String

Condition Keys	Description	Type
aws:RequestedRegion	Filters access by the AWS Region that was called in the request.	String
aws:RequestTag	Filters access based on the presence of tag key-value pairs in the request.	String
aws:ResourceTag	Filters access based on tag key-value pairs attached to the resource.	String
aws:SecureTransport	Allows access only if the request was sent using SSL.	Boolean
aws:SourceIp	Filters access by the requester's IP address.	String
aws:TagKeys	(Multivalue key) Filters access based on the presence of tag keys in the request.	ArrayOfString
aws:TokenIssueTime	Filters access by the date and time that temporary security credentials were issued.	String
aws:UserAgent	Filters access by the requester's client application.	String
aws:userid	Filters access by the requester's principal identifier.	String
aws:ViaAWSService	Allows access only if an AWS service made the request on your behalf.	Boolean

Neptune service-specific condition keys

Neptune supports the following service-specific condition key for IAM policies:

Neptune service-specific condition keys

Condition Keys	Description	Type
`neptune-db:QueryLanguage`	Filters data access by the query language being used. Valid values are: `Gremlin`, `OpenCypher` , and `Sparql`. Supported actions are `ReadDataViaQuery` , `WriteDataViaQuery` , `DeleteDataViaQuery` , `GetQueryStatus` , and `CancelQuery` .	`String`

Creating IAM data-access policies in Amazon Neptune

The following examples show how to create custom IAM policies that use fine-grained access control of data-plane APIs and actions, introduced in Neptune engine release version 1.2.0.0.

Policy example allowing unrestricted access to the data in a Neptune DB cluster

The following example policy allows an IAM user to connect to a Neptune DB cluster using IAM database authentication, and uses the "*" character to match all available actions.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "neptune-db:*",
      "Resource": "arn:aws-eusc:neptune-db:eusc-de-east-1:123456789012:cluster-ABCD1234EFGH5678IJKL90MNOP/*"
    }
  ]
}
```

The preceding example includes a resource ARN in a format that is particular to Neptune IAM authentication. To construct the ARN, see Specifying data resources. Note that the ARN used for an IAM authorization `Resource` is not the same as the ARN assigned to the cluster on creation.

Policy example allowing read-only access to a Neptune DB cluster

The following policy grants permission for full read-only access to data in a Neptune DB cluster:

JSON

```
{
  "Version":"2012-10-17",
  "Statement":[
    {
      "Effect": "Allow",
      "Action": [
        "neptune-db:Read*",
        "neptune-db:Get*",
        "neptune-db:List*"
      ],
      "Resource": "arn:aws-eusc:neptune-db:eusc-de-east-1:123456789012:cluster-
ABCD1234EFGH5678IJKL90MNOP/*"
    }
  ]
}
```

Policy example denying all access to a Neptune DB cluster

The default IAM action is to deny access to a DB cluster unless an `Allow` *Effect* is granted. However, the following policy denies all access to a DB cluster for a particular AWS account and Region, which then takes precedence over any `Allow` effect.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": "neptune-db:*",
      "Resource": "arn:aws-eusc:neptune-db:eusc-de-east-1:123456789012:cluster-
ABCD1234EFGH5678IJKL90MNOP/*"
    }
```

```
  ]
}
```

Policy example granting read access through queries

The following policy only grants permission to read from a Neptune DB cluster using a query:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "neptune-db:ReadDataViaQuery",
      "Resource": "arn:aws-eusc:neptune-db:eusc-de-east-1:123456789012:cluster-
ABCD1234EFGH5678IJKL90MNOP/*"
    }
  ]
}
```

Policy example that only allows Gremlin queries

The following policy uses the `neptune-db:QueryLanguage` condition key to grant permission to query Neptune only using the Gremlin query language:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "neptune-db:ReadDataViaQuery",
        "neptune-db:WriteDataViaQuery",
        "neptune-db:DeleteDataViaQuery"
      ],
      "Resource": "*",
```

```
      "Condition": {
        "StringEquals": {
          "neptune-db:QueryLanguage": "Gremlin"
        }
      }
    }
  ]
}
```

Policy example allowing all access except to Neptune ML model management

The following policy grants full access to Neptune graph operations except for the Neptune ML model-management features:

JSON

```
{
  "Version":"2012-10-17",
  "Statement":[
    {
      "Effect": "Allow",
      "Action": [
        "neptune-db:CancelLoaderJob",
        "neptune-db:CancelQuery",
        "neptune-db:DeleteDataViaQuery",
        "neptune-db:DeleteStatistics",
        "neptune-db:GetEngineStatus",
        "neptune-db:GetLoaderJobStatus",
        "neptune-db:GetQueryStatus",
        "neptune-db:GetStatisticsStatus",
        "neptune-db:GetStreamRecords",
        "neptune-db:ListLoaderJobs",
        "neptune-db:ManageStatistics",
        "neptune-db:ReadDataViaQuery",
        "neptune-db:ResetDatabase",
        "neptune-db:StartLoaderJob",
        "neptune-db:WriteDataViaQuery"
      ],
      "Resource": "arn:aws-eusc:neptune-db:eusc-de-east-1:123456789012:cluster-
ABCD1234EFGH5678IJKL90MNOP/*"
    }
  ]
```

```
}
```

Policy example allowing access to Neptune ML model management

This policy grants access to the Neptune ML model-management features:

JSON

```
{
  "Version":"2012-10-17",
  "Statement":[
    {
      "Effect": "Allow",
      "Action": [
        "neptune-db:CancelMLDataProcessingJob",
        "neptune-db:CancelMLModelTrainingJob",
        "neptune-db:CancelMLModelTransformJob",
        "neptune-db:CreateMLEndpoint",
        "neptune-db:DeleteMLEndpoint",
        "neptune-db:GetMLDataProcessingJobStatus",
        "neptune-db:GetMLEndpointStatus",
        "neptune-db:GetMLModelTrainingJobStatus",
        "neptune-db:GetMLModelTransformJobStatus",
        "neptune-db:ListMLDataProcessingJobs",
        "neptune-db:ListMLEndpoints",
        "neptune-db:ListMLModelTrainingJobs",
        "neptune-db:ListMLModelTransformJobs",
        "neptune-db:StartMLDataProcessingJob",
        "neptune-db:StartMLModelTrainingJob",
        "neptune-db:StartMLModelTransformJob"
      ],
      "Resource": "arn:aws-eusc:neptune-db:eusc-de-east-1:123456789012:cluster-
ABCD1234EFGH5678IJKL90MNOP/*"
    }
  ]
}
```

Policy example granting full query access

The following policy grants full access to Neptune graph query operations, but not to features like fast reset, streams, the bulk loader, Neptune ML model management, and so forth:

JSON

```
{
  "Version":"2012-10-17",
  "Statement":[
    {
      "Effect": "Allow",
      "Action": [
        "neptune-db:ReadDataViaQuery",
        "neptune-db:WriteDataViaQuery",
        "neptune-db:DeleteDataViaQuery",
        "neptune-db:GetEngineStatus",
        "neptune-db:GetQueryStatus",
        "neptune-db:CancelQuery"
      ],
      "Resource": "arn:aws-eusc:neptune-db:eusc-de-east-1:123456789012:cluster-
ABCD1234EFGH5678IJKL90MNOP/*"
    }
  ]
}
```

Policy example granting full access for Gremlin queries only

The following policy grants full access to Neptune graph query operations using the Gremlin query language, but not to queries in other languages, and not to features like fast reset, streams, the bulk loader, Neptune ML model management, and so on:

JSON

```
{
  "Version":"2012-10-17",
  "Statement":[
    {
      "Effect": "Allow",
      "Action": [
```

```
        "neptune-db:ReadDataViaQuery",
        "neptune-db:WriteDataViaQuery",
        "neptune-db:DeleteDataViaQuery",
        "neptune-db:GetEngineStatus",
        "neptune-db:GetQueryStatus",
        "neptune-db:CancelQuery"
      ],
      "Resource": [
        "arn:aws-eusc:neptune-db:eusc-de-east-1:123456789012:cluster-
ABCD1234EFGH5678IJKL90MNOP/*"
      ],
      "Condition": {
        "StringEquals": {
           "neptune-db:QueryLanguage":"Gremlin"
        }
      }
    }
  ]
}
```

Policy example granting full access except for fast reset

The following policy grants full access to a Neptune DB cluster except for using fast reset:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "neptune-db:*",
      "Resource": "arn:aws-eusc:neptune-db:eusc-de-east-1:123456789012:cluster-
ABCD1234EFGH5678IJKL90MNOP/*"
    },
    {
      "Effect": "Deny",
      "Action": "neptune-db:ResetDatabase",
      "Resource": "arn:aws-eusc:neptune-db:eusc-de-east-1:123456789012:cluster-
ABCD1234EFGH5678IJKL90MNOP/*"
    }
  ]
```

```
}
```

Tag-based access control for Amazon Neptune data-plane operations

Tag-based access control (TBAC) lets you use AWS resource tags and IAM principal tags as conditions in IAM policies and Service Control Policies (SCPs) to control access to Amazon Neptune data-plane operations. With TBAC, you can enforce that only principals whose tags match the tags on a Neptune DB cluster can perform `neptune-db:*` actions against that cluster, without enumerating specific cluster Amazon Resource Names (ARNs) in every policy.

TBAC builds on Neptune's existing security model and complements the action-based access control data-plane actions.

How TBAC fits into Neptune's security layers

Neptune protects your data through multiple, overlapping security mechanisms. TBAC adds an attribute-based authorization layer that works alongside all of them:

Neptune security layers and how TBAC complements them

Layer	Mechanism	Scope
Network isolation	Virtual private cloud (VPC), security groups, VPC endpoints (PrivateLink)	Controls which hosts can reach Neptune endpoints
Encryption	Transport Layer Security (TLS) 1.3 in transit; AWS KMS-managed encryption at rest	Protects data confidentiality
IAM authentication	AWS Signature Version 4 (SigV4)-signed requests to the Neptune data endpoint	Authenticates the caller
Action-based access control	`neptune-db:` actions (`ReadDataViaQuery`, `WriteDataViaQuery`, etc.)	Controls what operations a principal can perform

Layer	Mechanism	Scope
Condition keys	`neptune-db:QueryLanguage` , global context keys	Adds contextual constraints to policies
TBAC	`aws:ResourceTag/${TagKey}` evaluated against `aws:PrincipalTag/${TagKey}`	Restricts access based on tag alignment between principal and resource
Administrative tag-based access	`aws:ResourceTag` , `rds:cluster-tag` , etc. on management-plane actions	Controls who can manage Neptune infrastructure

Key TBAC concepts

Principal tags

Tags attached to IAM users, roles, or federated session principals. You can set these via the IAM console, AWS CLI, or identity provider (IdP) Security Assertion Markup Language (SAML) / OpenID Connect (OIDC) attribute mappings.

Resource tags

Tags attached to Neptune DB clusters using `AddTagsToResource`. These propagate to all instances in the cluster for data-plane policy evaluation.

Condition key variables

- `aws:PrincipalTag/`*`TagKey`* — resolves to the tag value on the calling principal.
- `aws:ResourceTag/`*`TagKey`* — resolves to the tag value on the target Neptune resource.

Supported policy types

- **IAM identity policies** — attached to users, groups, or roles.
- **SCPs** — applied at the AWS Organizations organizational unit (OU) or account level to set permission guardrails.

Prerequisites for using TBAC

Before you can use TBAC with Neptune data-plane operations, you must have the following in place:

1. **Neptune engine version 1.2.0.0 or later** — required for data-plane TBAC support.
2. **IAM authentication enabled** on the Neptune DB cluster.
3. **Tags applied to Neptune DB clusters** — the resource tags that policies will evaluate.
4. **Tags applied to IAM principals** — the principal tags that will be compared against resource tags.

TBAC policy patterns

The following patterns show common ways to use TBAC in IAM policies for Neptune data-plane operations.

Deny access when principal and resource tags do not match

This is the most common TBAC pattern. It denies all Neptune data-plane actions unless the principal's tags match the resource's tags. You can apply this as an SCP for organization-wide enforcement or as an IAM policy for targeted control.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DenyNeptuneProjectMismatch",
      "Effect": "Deny",
      "Action": "neptune-db:*",
      "Resource": "*",
      "Condition": {
        "StringNotEquals": {
          "aws:ResourceTag/Project": "${aws:PrincipalTag/Project}"
        }
      }
    },
    {
      "Sid": "DenyNeptuneDepartmentMismatch",
      "Effect": "Deny",
      "Action": "neptune-db:*",
      "Resource": "*",
```

```
      "Condition": {
        "StringNotEquals": {
          "aws:ResourceTag/Department": "${aws:PrincipalTag/Department}"
        }
      }
    }
  ]
}
```

How it works: Each statement uses a separate `StringNotEquals` condition for a single tag key. The Deny triggers independently for each tag—if the resource's `Project` tag does not match the principal's `Project` tag, access is denied regardless of the `Department` tag. This ensures a principal tagged with `Project=FraudDetection` can only access Neptune clusters also tagged `Project=FraudDetection`, and similarly for `Department`.

Deny access when required resource tags are missing

This pattern prevents access to Neptune clusters that have not been properly tagged, ensuring all clusters are enrolled in the TBAC scheme:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DenyNeptuneMissingProjectTag",
      "Effect": "Deny",
      "Action": "neptune-db:*",
      "Resource": "*",
      "Condition": {
        "Null": {
          "aws:ResourceTag/Project": "true"
        }
      }
    },
    {
      "Sid": "DenyNeptuneMissingDepartmentTag",
      "Effect": "Deny",
      "Action": "neptune-db:*",
      "Resource": "*",
      "Condition": {
        "Null": {
          "aws:ResourceTag/Department": "true"
        }
```

```
      }
    }
  ]
}
```

How it works: The `Null` condition evaluates to true when the specified tag key does not exist on the resource. This forces all Neptune clusters to carry the required classification tags before any principal can access them.

Combining TBAC with action-based access control

TBAC can be combined with specific `neptune-db:` actions to create fine-grained, tag-aware policies:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowReadOnlyForMatchingTags",
      "Effect": "Allow",
      "Action": [
        "neptune-db:ReadDataViaQuery",
        "neptune-db:GetQueryStatus",
        "neptune-db:GetEngineStatus"
      ],
      "Resource": "arn:aws:neptune-db:*:*:*/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Project": "${aws:PrincipalTag/Project}"
        }
      }
    }
  ]
}
```

Query language restriction with TBAC

Combine TBAC with the `neptune-db:QueryLanguage` condition key to restrict both which clusters a principal can access and which query languages they can use:

```
{
  "Version": "2012-10-17",
  "Statement": [
```

```
    {
      "Sid": "AllowOpenCypherOnlyForMatchingProject",
      "Effect": "Allow",
      "Action": [
        "neptune-db:ReadDataViaQuery",
        "neptune-db:WriteDataViaQuery"
      ],
      "Resource": "arn:aws:neptune-db:*:*:*/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Project": "${aws:PrincipalTag/Project}",
          "neptune-db:QueryLanguage": "OpenCypher"
        }
      }
    }
  ]
}
```

Using TBAC with service control policies

SCPs are ideal for enforcing TBAC because they set permission boundaries across an entire organizational unit (OU) or account without requiring changes to individual IAM policies.

We recommend the following SCP strategy:

1. Apply a Deny-based SCP at the OU level that blocks `neptune-db:*` when tags don't match.
2. Apply a second statement denying access to untagged resources.
3. Your individual accounts can retain their Allow policies for specific `neptune-db:` actions—the SCP acts as a guardrail.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DenyNeptuneProjectMismatch",
      "Effect": "Deny",
      "Action": "neptune-db:*",
      "Resource": "*",
      "Condition": {
        "StringNotEquals": {
          "aws:ResourceTag/Project": "${aws:PrincipalTag/Project}"
```

```
        }
      }
    },
    {
      "Sid": "DenyNeptuneDepartmentMismatch",
      "Effect": "Deny",
      "Action": "neptune-db:*",
      "Resource": "*",
      "Condition": {
        "StringNotEquals": {
          "aws:ResourceTag/Department": "${aws:PrincipalTag/Department}"
        }
      }
    },
    {
      "Sid": "DenyNeptuneMissingProjectTag",
      "Effect": "Deny",
      "Action": "neptune-db:*",
      "Resource": "*",
      "Condition": {
        "Null": {
          "aws:ResourceTag/Project": "true"
        }
      }
    },
    {
      "Sid": "DenyNeptuneMissingDepartmentTag",
      "Effect": "Deny",
      "Action": "neptune-db:*",
      "Resource": "*",
      "Condition": {
        "Null": {
          "aws:ResourceTag/Department": "true"
        }
      }
    }
  ]
}
```

Implementing TBAC for Neptune

Step 1: Define your tag taxonomy

Choose tag keys that represent your organizational boundaries. Common patterns:

Example tag taxonomy

Tag key	Purpose	Example values
`Project`	Application or workload identifier	`FraudDetection` , `RecommendationEngine`
`Department`	Business unit or cost center	`Engineering` , `Finance`, `Analytics`
`Environment`	Deployment stage	`production` , `staging`, `development`
`Team`	Owning team	`graph-platform` , `data-science`

Step 2: Tag your Neptune DB clusters

Use the AWS CLI to add the required classification tags to your Neptune DB clusters:

```
aws neptune add-tags-to-resource \
  --resource-name arn:aws:rds:us-east-1:123456789012:cluster:my-neptune-cluster \
  --tags Key=Project,Value=FraudDetection Key=Department,Value=Engineering
```

Step 3: Tag your IAM principals

Use the AWS CLI to tag IAM roles with the same keys and values used on your Neptune clusters. For IAM roles:

```
aws iam tag-role \
  --role-name NeptuneAppRole \
  --tags Key=Project,Value=FraudDetection Key=Department,Value=Engineering
```

For federated users, pass tags via SAML/OIDC session tags using `aws:PrincipalTag` attributes from your identity provider.

Step 4: Deploy the TBAC policy

Attach as an SCP for organization-wide enforcement, or as an IAM policy for targeted control.

Step 5: Protect tag integrity

Restrict who can modify tags on Neptune resources and IAM principals:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DenyTagModification",
      "Effect": "Deny",
      "Action": [
        "rds:AddTagsToResource",
        "rds:RemoveTagsFromResource"
      ],
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
          "aws:TagKeys": ["Project", "Department"]
        }
      }
    }
  ]
}
```

Important considerations for TBAC

- **Propagation delay** — Changes to IAM policies take up to 10 minutes to apply to Neptune resources. Changes to cluster tags (adding, modifying, or removing tags) take approximately 5 minutes to propagate to data-plane policy evaluation. Plan for this delay when updating tags on active clusters.
- **Cluster-level granularity** — You apply tags to Neptune DB clusters at the cluster level. All instances in a cluster share the same policy evaluation. TBAC does not provide sub-graph or vertex/edge-level access control.
- **IAM authentication required** — TBAC only applies when IAM authentication is enabled on the cluster. Connections without IAM auth bypass these policies entirely.
- **Tag immutability** — Secure your tagging operations. If a principal can modify their own tags or the resource tags, they can bypass TBAC controls. Use SCPs or permission boundaries to restrict `iam:TagRole`, `iam:TagUser`, `rds:AddTagsToResource`, and `rds:RemoveTagsFromResource`.

- **Null tag handling** — If a principal lacks a tag that the policy references via `${aws:PrincipalTag/Key}`, the variable resolves to an empty string. Design your policies to handle this case (the "missing tags" deny pattern above addresses this for resource tags).
- **Multiple condition keys** — When multiple condition keys appear in the same `Condition` block, they are evaluated with AND logic. For `StringNotEquals`, a Deny only triggers when *all* specified conditions are true simultaneously. To deny on *any* single tag mismatch, use separate policy statements for each tag key (as shown in the patterns above).

Relationship to existing Neptune security features

How TBAC complements existing Neptune security features

Existing feature	What it controls	How TBAC complements it
VPC / Security Groups	Network-level access to port 8182	TBAC adds identity-aware authorization on top of network controls
IAM Authentication (SigV4)	Verifies caller identity	TBAC uses the authenticated identity's tags for authorization decisions
Action-based access control	Which operations (read/write/delete/load) a principal can perform	TBAC adds which clusters a principal can target, based on tag alignment
`neptune-db:QueryLanguage` condition key	Which query languages (Gremlin, openCypher, SPARQL) are permitted	Can be combined with TBAC in the same policy statement
Administrative tag-based access (`rds:*` actions)	Who can manage Neptune infrastructure	TBAC extends the same tag-based pattern to data-plane (`neptune-db:*`) actions
AWS KMS encryption	Data confidentiality at rest	Orthogonal—TBAC controls authorization, not encryption

Using service-linked roles for Amazon Neptune

Amazon Neptune uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to Neptune. Service-linked roles are predefined by Neptune and include all the permissions that the service requires to call other AWS services on your behalf.

> **Important**
>
> For certain management features, Amazon Neptune uses operational technology that is shared with Amazon RDS. This includes the *service-linked role* and management API permissions.

A service-linked role makes using Neptune easier because you don't have to manually add the necessary permissions. Neptune defines the permissions of its service-linked roles, and unless defined otherwise, only Neptune can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

You can delete the roles only after first deleting their related resources. This protects your Neptune resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see AWS Services That Work with IAM, and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-Linked Role Permissions for Neptune

Neptune uses the `AWSServiceRoleForRDS` service-linked role to allow Neptune and Amazon RDS to call AWS services on behalf of your database instances. The `AWSServiceRoleForRDS` service-linked role trusts the `rds.amazonaws.com` service to assume the role.

The role permissions policy allows Neptune to complete the following actions on the specified resources:

- Actions on `ec2`:
  - `AssignPrivateIpAddresses`
  - `AuthorizeSecurityGroupIngress`

- `CreateNetworkInterface`
- `CreateSecurityGroup`
- `DeleteNetworkInterface`
- `DeleteSecurityGroup`
- `DescribeAvailabilityZones`
- `DescribeInternetGateways`
- `DescribeSecurityGroups`
- `DescribeSubnets`
- `DescribeVpcAttribute`
- `DescribeVpcs`
- `ModifyNetworkInterfaceAttribute`
- `RevokeSecurityGroupIngress`
- `UnassignPrivateIpAddresses`

- Actions on `sns`:
  - `ListTopic`
  - `Publish`
- Actions on `cloudwatch`:
  - `PutMetricData`
  - `GetMetricData`
  - `CreateLogStream`
  - `PullLogEvents`
  - `DescribeLogStreams`
  - `CreateLogGroup`

> **Note**
>
> You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. You might encounter the following error message:
> Unable to create the resource. Verify that you have permission to create service linked role. Otherwise wait and try again later.

If you see this message, make sure that you have the following permissions enabled:

```
{
    "Action": "iam:CreateServiceLinkedRole",
    "Effect": "Allow",
    "Resource": "arn:aws:iam::*:role/aws-service-role/rds.amazonaws.com/
AWSServiceRoleForRDS",
    "Condition": {
        "StringLike": {
            "iam:AWSServiceName":"rds.amazonaws.com"
        }
    }
}
```

For more information, see Service-Linked Role Permissions in the *IAM User Guide*.

Creating a Service-Linked Role for Neptune

You don't need to manually create a service-linked role. When you create an instance or a cluster, Neptune creates the service-linked role for you.

Important

To learn more, see A New Role Appeared in My IAM Account in the *IAM User Guide*.

If you delete this service-linked role and then need to create it again, you can use the same process to re-create the role in your account. When you create an instance or a cluster, Neptune creates the service-linked role for you again.

Editing a Service-Linked Role for Neptune

Neptune does not allow you to edit the `AWSServiceRoleForRDS` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a Service-Linked Role in the *IAM User Guide*.

Deleting a Service-Linked Role for Neptune

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must delete all of your instances and clusters before you can delete the associated service-linked role.

Cleaning Up a Service-Linked Role Before Deleting

Before you can use IAM to delete a service-linked role, you must first confirm that the role has no active sessions and remove any resources used by the role.

To check whether the service-linked role has an active session in the IAM console

1. Sign in to the AWS Management Console and open the IAM console at https://eusc-de-east-1.console.amazonaws-eusc.eu/iam/.
2. In the navigation pane of the IAM console, choose **Roles**. Then choose the name (not the check box) of the `AWSServiceRoleForRDS` role.
3. On the **Summary** page for the selected role, choose the **Access Advisor** tab.
4. On the **Access Advisor** tab, review recent activity for the service-linked role.

> **Note**
>
> If you are unsure whether Neptune is using the `AWSServiceRoleForRDS` role, you can try to delete the role. If the service is using the role, then the deletion fails and you can view the Regions where the role is being used. If the role is being used, then you must wait for the session to end before you can delete the role. You cannot revoke the session for a service-linked role.

If you want to remove the `AWSServiceRoleForRDS` role, you must first delete *all* of your instances and clusters.

Deleting All of Your Instances

Use one of these procedures to delete each of your instances.

To delete an instance (console)

1. Open the Amazon RDS console at https://eusc-de-east-1.console.amazonaws-eusc.eu/rds/.
2. In the navigation pane, choose **Instances**.
3. In the **Instances** list, choose the instance that you want to delete.
4. Choose **Instance actions**, and then choose **Delete**.
5. If you are prompted for **Create final Snapshot?**, choose **Yes** or **No**.
6. If you chose **Yes** in the previous step, for **Final snapshot name** enter the name of your final snapshot.
7. Choose **Delete**.

To delete an instance (AWS CLI)

See `delete-db-instance` in the *AWS CLI Command Reference*.

To delete an instance (API)

See `DeleteDBInstance`.

Deleting All of Your Clusters

Use one of the following procedures to delete a single cluster, and then repeat the procedure for each of your clusters.

To delete a cluster (console)

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the **Clusters** list, choose the cluster that you want to delete.
3. Choose **Cluster Actions**, and then choose **Delete**.
4. Choose **Delete**.

To delete a cluster (CLI)

See `delete-db-cluster` in the *AWS CLI Command Reference*.

To delete a cluster (API)

See `DeleteDBCluster`

You can use the IAM console, the IAM CLI, or the IAM API to delete the `AWSServiceRoleForRDS` service-linked role. For more information, see Deleting a Service-Linked Role in the *IAM User Guide*.

Using temporary credentials to connect to Amazon Neptune

Amazon Neptune supports IAM authentication using temporary credentials.

You can use an assumed role to authenticate using an IAM authentication policy, like one of the example policies in the previous sections.

If you are using temporary credentials, you must specify `AWS_SESSION_TOKEN` in addition to `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `SERVICE_REGION`.

> **Note**
>
> The temporary credentials expire after a specified interval, *including the session token*. You must update your session token when you request new credentials. For more information, see Using Temporary Security Credentials to Request Access to AWS Resources.

The following sections describe how to allow access and retrieve temporary credentials.

To authenticate using temporary credentials

1. Create an IAM role with permission to access a Neptune cluster. For information about creating this role, see the section called “Types of IAM policies”.
2. Add a trust relationship to the role that allows access to the credentials.

   Retrieve the temporary credentials, including the `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN`.
3. Connect to the Neptune cluster and sign the requests using the temporary credentials. For more information about connecting and signing requests, see the section called “Connecting with IAM authentication”.

There are various methods for retrieving temporary credentials depending on the environment.

Topics

- Getting Temporary Credentials Using the AWS CLI
- Setting Up AWS Lambda for Neptune IAM Authentication
- Setting Up Amazon EC2 for Neptune IAM Authentication

Getting Temporary Credentials Using the AWS CLI

To get credentials using the AWS Command Line Interface (AWS CLI), first you need to add a trust relationship that grants permission to assume the role to the AWS user that will run the AWS CLI command.

Add the following trust relationship to the Neptune IAM authentication role. If you don't have a Neptune IAM authentication role, see the section called “Types of IAM policies”.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws-eusc:iam::123456789012:user/test"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

For information about adding the trust relationship to the role, see Editing the Trust Relationship for an Existing Role in the *AWS Directory Service Administration Guide*.

If the Neptune policy is not yet attached to a role, create a new role. Attach the Neptune IAM authentication policy, and then add the trust policy. For information about creating a new role, see Creating a New Role.

> **Note**
>
> The following sections assume that you have the AWS CLI installed.

To run the AWS CLI manually

1. Enter the following command to request the credentials using the AWS CLI. Replace the role ARN, session name, and profile with your own values.

```
aws sts assume-role  --role-arn  arn:aws:iam::123456789012:role/NeptuneIAMAuthRole
 --role-session-name test --profile testprofile
```

2. The following is example output from the command. The `Credentials` section contains the values that you need.

> **Note**
>
> Record the `Expiration` value because you need to get new credentials after this time.

```
{
    "AssumedRoleUser": {
        "AssumedRoleId": "AROA3XFRBF535PLBIFPI4:s3-access-example",
        "Arn": "arn:aws:sts::123456789012:assumed-role/xaccounts3access/s3-access-
example"
    },
    "Credentials": {
        "SecretAccessKey": "9drTJvcXLB89EXAMPLELB8923FB892xMFI",
        "SessionToken": "AQoXdzELDDY//////////
wEaoAK1wvxJY12r2IrDFT2IvAzTCn3zHoZ7YNtpiQLF0MqZye/qwjzP2iEXAMPLEbw/
m3hsj8VBTkPORGvr9jM5sgP+w9IZWZnU+LWhmg
+a5fDi2oTGUYcdg9uexQ4mtCHIHfi4citgqZTgco40Yqr4lIlo4V2b2Dyauk0eYFNebHtYlFVgAUj
+7Indz3LU0aTWk1WKIjHmmMCIoTkyYp/k7kUG7moeEYKSitwQIi6Gjn+nyzM
+PtoA3685ixzv0R7i5rjQi0YE0lf1oeie3bDiNHncmzosRM6SFiPzSvp6h/32xQuZsjcypmwsPSDtTPYcs0+YN/8BRi
IcrxSpnWEXAMPLEXSDFTAQAM6Dl9zR0tXoybnlrZIwMLlMi1Kcgo5OytwU=",
        "Expiration": "2016-03-15T00:05:07Z",
        "AccessKeyId": "ASIAJEXAMPLEXEG2JICEA"
    }
}
```

3. Set the environment variables using the returned credentials.

```
export AWS_ACCESS_KEY_ID=ASIAJEXAMPLEXEG2JICEA
export AWS_SECRET_ACCESS_KEY=9drTJvcXLB89EXAMPLELB8923FB892xMFI
```

```
export AWS_SESSION_TOKEN=AQoXdzELDDY//////////
wEaoAK1wvxJY12r2IrDFT2IvAzTCn3zHoZ7YNtpiQLF0MqZye/qwjzP2iEXAMPLEbw/
m3hsj8VBTkPORGvr9jM5sgP+w9IZWZnU+LWhmg
+a5fDi2oTGUYcdg9uexQ4mtCHIHfi4citgqZTgco40Yqr4lIlo4V2b2Dyauk0eYFNebHtYlFVgAUj
+7Indz3LU0aTWk1WKIjHmmMCIoTkyYp/k7kUG7moeEYKSitwQIi6Gjn+nyzM
+PtoA3685ixzv0R7i5rjQi0YE0lf1oeie3bDiNHncmzosRM6SFiPzSvp6h/32xQuZsjcypmwsPSDtTPYcs0+YN/8BRi
IcrxSpnWEXAMPLEXSDFTAQAM6Dl9zR0tXoybnlrZIwMLlMi1Kcgo5OytwU=

export SERVICE_REGION=us-east-1 or us-east-2 or us-west-1 or us-west-2 or ca-
central-1 or
                      ca-west-1 or sa-east-1 or eu-north-1 or eu-south-2 or eu-
west-1 or eu-west-2 or eu-west-3 or
                      eu-central-1 or me-south-1 or me-central-1 or il-central-1 or
 af-south-1 or ap-east-1 or
                      ap-northeast-1 or ap-northeast-2 or ap-northeast-3 or ap-
southeast-1 or ap-southeast-2 or
                      ap-southeast-3 or ap-southeast-4 or ap-southeast-5 or ap-
south-1 or ap-south-2 or
                      cn-north-1 or cn-northwest-1 or
                      us-gov-east-1 or us-gov-west-1
```

4. Connect using one of the following methods.

   - the section called “Connecting from Gremlin Console”
   - the section called “Connecting using Gremlin Java”
   - the section called “Connecting with SPARQL Java”
   - the section called “Connecting with Python”

To use a script to get the credentials

1. Run the following command to install the **jq** command. The script uses this command to parse the output of the AWS CLI command.

```
sudo yum -y install jq
```

2. Create a file named `credentials.sh` in a text editor and add the following text. Replace the service Region, role ARN, session name, and profile with your own values.

```
#!/bin/bash
```

```
creds_json=$(aws sts assume-role  --role-arn  arn:aws:iam::123456789012:role/
NeptuneIAMAuthRole --role-session-name test --profile testprofile)

export AWS_ACCESS_KEY_ID=$(echo "$creds_json" | jq .Credentials.AccessKeyId |tr -d
 '"')
export AWS_SECRET_ACCESS_KEY=$(echo "$creds_json" |
 jq .Credentials.SecretAccessKey| tr -d '"')
export AWS_SESSION_TOKEN=$(echo "$creds_json" | jq .Credentials.SessionToken|tr -d
 '"')

export SERVICE_REGION=us-east-1 or us-east-2 or us-west-1 or us-west-2 or ca-
central-1 or
                      ca-west-1 or sa-east-1 or eu-north-1 or eu-south-2 or eu-
west-1 or eu-west-2 or eu-west-3 or
                      eu-central-1 or me-south-1 or me-central-1 or il-central-1 or
 af-south-1 or ap-east-1 or
                      ap-northeast-1 or ap-northeast-2 or ap-northeast-3 or ap-
southeast-1 or ap-southeast-2 or
                      ap-southeast-3 or ap-southeast-4 or ap-southeast-5 or ap-
south-1 or ap-south-2 or
                      cn-north-1 or cn-northwest-1 or
                      us-gov-east-1 or us-gov-west-1
```

3. Connect using one of the following methods.

   - the section called “Connecting from Gremlin Console”
   - the section called “Connecting using Gremlin Java”
   - the section called “Connecting with SPARQL Java”
   - the section called “Connecting with Python”

Setting Up AWS Lambda for Neptune IAM Authentication

AWS Lambda includes credentials automatically each time the Lambda function is run.

First you add a trust relationship that grants permission to assume the role to the Lambda service.

Add the following trust relationship to the Neptune IAM authentication role. If you don't have a Neptune IAM authentication role, see the section called “Types of IAM policies”.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

For information about adding the trust relationship to the role, see Editing the Trust Relationship for an Existing Role in the *AWS Directory ServiceAdministration Guide*.

If the Neptune policy is not yet attached to a role, create a new role. Attach the Neptune IAM authentication policy, and then add the trust policy. For information about creating a new role, see Creating a New Role in the *AWS Directory Service Administration Guide*.

To access Neptune from Lambda

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://eusc-de-east-1.console.amazonaws-eusc.eu/lambda/.
2. Create a new Lambda function for Python version 3.6.
3. Assign the `AWSLambdaVPCAccessExecutionRole` role to the Lambda function. This is required to access Neptune resources, which are VPC only.
4. Assign the Neptune authentication IAM role to the Lambda function.

   For more information, see AWS Lambda Permissions in the *AWS Lambda Developer Guide*.
5. Copy the IAM authentication Python sample into the Lambda function code.

   For more information about the sample and the sample code, see the section called "Connecting with Python".

Setting Up Amazon EC2 for Neptune IAM Authentication

Amazon EC2 allows you to use instance profiles to automatically provide credentials. For more information, see Using Instance Profiles in the *IAM User Guide*.

First you add a trust relationship that grants permission to assume the role to the Amazon EC2 service.

Add the following trust relationship to the Neptune IAM authentication role. If you don't have a Neptune IAM authentication role, see the section called “Types of IAM policies”.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

For information about adding the trust relationship to the role, see Editing the Trust Relationship for an Existing Role in the *AWS Directory Service Administration Guide*.

If the Neptune policy is not yet attached to a role, create a new role. Attach the Neptune IAM authentication policy, and then add the trust policy. For information about creating a new role, see Creating a New Role in the *AWS Directory Service Administration Guide*.

To use a script to get the credentials

1. Run the following command to install the **jq** command. The script uses this command to parse the output of the **curl** command.

   ```
   sudo yum -y install jq
   ```

2. Create a file named `credentials.sh` in a text editor and add the following text. Replace the service Region with your own value.

```
TOKEN=$( curl -X PUT "http://169.254.169.254/latest/api/token" -H "X-aws-ec2-
metadata-token-ttl-seconds: 21600" )
role_name=$( curl -H "X-aws-ec2-metadata-token: $TOKEN" http://169.254.169.254/
latest/meta-data/iam/security-credentials/ )
creds_json=$( curl -H "X-aws-ec2-metadata-token: $TOKEN" http://169.254.169.254/
latest/meta-data/iam/security-credentials/${role_name} )

export AWS_ACCESS_KEY_ID=$(echo "$creds_json" | jq .AccessKeyId |tr -d '"')
export AWS_SECRET_ACCESS_KEY=$(echo "$creds_json" | jq .SecretAccessKey| tr -d '"')
export AWS_SESSION_TOKEN=$(echo "$creds_json" | jq .Token|tr -d '"')

export SERVICE_REGION=us-east-1 or us-east-2 or us-west-1 or us-west-2 or ca-
central-1 or
                      ca-west-1 or sa-east-1 or eu-north-1 or eu-south-2 or eu-
west-1 or eu-west-2 or eu-west-3 or
                      eu-central-1 or me-south-1 or me-central-1 or il-central-1 or
 af-south-1 or ap-east-1 or
                      ap-northeast-1 or ap-northeast-2 or ap-northeast-3 or ap-
southeast-1 or ap-southeast-2 or
                      ap-southeast-3 or ap-southeast-4 or ap-southeast-5 or ap-
south-1 or ap-south-2 or
                      cn-north-1 or cn-northwest-1 or
                      us-gov-east-1 or us-gov-west-1
```

3. Run the script in the `bash` shell using the `source` command:

```
source credentials.sh
```

   Even better is to add the commands in this script to the `.bashrc` file on your EC2 instance so that they will be invoked automatically when you log in, making temporary credentials available to the Gremlin console.

4. Connect using one of the following methods.

   - the section called “Connecting from Gremlin Console”
   - the section called “Connecting using Gremlin Java”
   - the section called “Connecting with SPARQL Java”
   - the section called “Connecting with Python”

Logging and monitoring usage and performance in Amazon Neptune

Amazon Neptune supports various methods for monitoring performance and usage:

- **Cluster status** – Check the health of a Neptune cluster's graph database engine. For more information, see the section called "Instance Status".
- **Amazon CloudWatch** – Neptune automatically sends metrics to CloudWatch and also supports CloudWatch Alarms. For more information, see the section called "Using CloudWatch".
- **Audit log files** – View, download, or watch database log files using the Neptune console. For more information, see the section called "Audit Logs with Neptune".
- **Publishing logs to Amazon CloudWatch Logs** – You can configure a Neptune DB cluster to publish audit log data to a log group in Amazon CloudWatch Logs. With CloudWatch Logs, you can perform real-time analysis of the log data, use CloudWatch to create alarms and view metrics, and use CloudWatch Logs to store your log records in highly durable storage. For more information, see Neptune CloudWatch Logs.
- **AWS CloudTrail** – Neptune supports API logging using CloudTrail. For more information, see the section called "Logging Neptune API Calls with AWS CloudTrail".
- **Tagging** – Use tags to add metadata to your Neptune resources and track usage based on tags. For more information, see the section called "Tagging Neptune resources".

Amazon Neptune and interface VPC endpoints (AWS PrivateLink)

You can use AWS PrivateLink to create a private connection between your VPC and Amazon Neptune API endpoints. You can access Amazon Neptune API operations as though they were in your VPC, without the use of an internet gateway, NAT device, VPN connection, or connection. Traffic between your VPC and Amazon Neptune stays within the Amazon network.

Amazon Neptune shares API infrastructure with Amazon Relational Database Service. To set up interface VPC endpoints for Amazon Neptune, follow the instructions in Amazon RDS API and interface VPC endpoints (AWS PrivateLink) in the *Amazon Aurora User Guide*.

Compliance considerations for Amazon Neptune

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. For more information about your compliance responsibility when using AWS services, see AWS Security Documentation.

Building resilient and disaster-tolerant Amazon Neptune deployments

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

An Amazon Neptune DB cluster can only be created in an Amazon VPC that has at least two subnets in at least two Availability Zones. By distributing your cluster instances across at least two Availability Zones, Neptune helps ensure that there are instances available in your DB cluster in the unlikely event of an Availability Zone failure. The cluster volume for your Neptune DB cluster always spans three Availability Zones to provide durable storage with less possibility of data loss.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

Migrating an existing graph to Amazon Neptune

There are a number of tools and techniques that can help you migrate existing graph data into Amazon Neptune from another data store.

A simple migration workflow involves the following steps:

- Export the data from its existing store into Amazon Simple Storage Service (Amazon S3).
- Clean and format it for import.
- Load it into a Neptune DB cluster using Neptune Bulk Loader.
- Configure your Gremlin or SPARQL application to use the corresponding endpoint that Neptune provides.

> **Note**
>
> Your Neptune cluster must be running in a VPC that your application can access.

There are ways to simplify and automate some of these steps, depending on where the data is stored:

Topics

- Migrating from Neo4j to Amazon Neptune
- Migrating an existing graph from an Apache TinkerPop Gremlin server to Amazon Neptune
- Migrating an existing graph from an RDF triple store to Amazon Neptune
- Using AWS Database Migration Service (AWS DMS) to migrate from a relational or NoSQL database to Amazon Neptune
- Migrating from Blazegraph to Amazon Neptune

Migrating from Neo4j to Amazon Neptune

Neo4j and Amazon Neptune are both graph databases, designed for online, transactional graph workloads that support the labeled property graph data model. These similarities make Neptune a common choice for customers seeking to migrate their current Neo4j applications. However, these migrations are not simply lift and shift, because there are differences in languages and feature support, operational characteristics, server architecture, and storage capabilities between the two databases.

This page frames the migration process and brings up things to consider before migrating a Neo4j graph application to Neptune. These considerations apply generally to any Neo4j graph application, whether powered by a Community, Enterprise, or Aura database. Although each solution is unique and may require additional procedures, all migrations follow the same general pattern.

Each of the steps described in the following sections includes considerations and recommendations to simplify the migration process. Additionally, there are open-source tools, and blog posts that describe the process, and a feature compatibility section with recommended architectural options.

Topics

- General information about migrating from Neo4j to Neptune
- Preparing to migrate from Neo4j to Neptune
- Provisioning infrastructure when migrating from Neo4j to Neptune
- Data migration from Neo4j to Neptune
- Application migration from Neo4j to Neptune
- Neptune compatibility with Neo4j
- Rewriting Cypher queries to run in openCypher on Neptune
- Resources for migrating from Neo4j to Neptune

General information about migrating from Neo4j to Neptune

With Neptune support for the openCypher query language, you can move most Neo4j workloads that use the Bolt protocol or HTTPS to Neptune. However, openCypher is an open-source specification that contains most but not all of the functionality supported by other databases such as Neo4j.

In spite of being compatible in many ways, Neptune is not a drop-in replacement for Neo4j. Neptune is a fully managed graph database service with enterprise features like high availability and high durability that is architecturally different from Neo4j. Neptune is instance-based, with a single primary writer instance and up to 15 read replica instances that let you scale read capacity horizontally. Using Neptune Serverless, you can automatically scale your compute capacity up and down depending on query volume. This is independent of Neptune storage, which scales automatically as you add data.

Neptune supports the open-source openCypher standard specification, version 9. At AWS, we believe that open source is good for everyone and we are committed both to bringing the value of open source to our customers, and to bringing the operational excellence of AWS to open source communities.

However, many applications running on Neo4j also use proprietary features that are not open-sourced and that Neptune doesn't support. For example, Neptune doesn't support APOC procedures, some Cypher-specific clauses and functions, and `Char`, `Date`, or `Duration` data types. Neptune does auto-cast the missing data types to data types that are supported.

In addition to openCypher, Neptune also supports the Apache TinkerPop Gremlin query language for property graphs (as well as SPARQL for RDF data). Gremlin can interoperate with openCypher on the same property graph, and in many cases you can use Gremlin to supply functionality that openCypher does not provide. Below is a quick comparison of the two languages:

	openCypher	**Gremlin**
Style	Declarative	Imperative

Syntax	Pattern matching
	`Match p=(a)-[:route]->(d)`
	`WHERE a.code='ANC'`
	`RETURN p`
		Traversal based
		`g.V().has('code', 'ANC').`
		`out('route').path().`
		`by(elementMap())`

Ease of use	SQL-inspired, readable by non-programmers	Steeper learning curve, similar to programming languages like Java
Flexibility	Low	High

	openCypher	Gremlin
Query support	String-based queries	String-based queries or in-line code supported by client libraries
Clients	HTTPS and Bolt	HTTPS and Websockets

In general, it isn't necessary to change your data model to migrate from Neo4j to Neptune, because both Neo4j and Neptune support labeled property graph (LPG) data. However, Neptune has some architectural and data model differences that you can take advantage of to optimize performance. For example:

- Neptune IDs are treated as first-class citizens.
- Neptune uses AWS Identity and Access Management (IAM) policies to secure access to your graph data in flexible and granular ways.
- Neptune provides several ways to use Jupyter notebooks to run queries and visualize the results. Neptune also works with third-party visualization tools.
- >Although Neptune has no drop-in replacement for the Neo4j Graph Data Science (GDS) library, Neptune supports graph analytics today through a variety of solutions. For example, several sample notebooks demonstrate how to leverage the Neptune integration with the AWS Pandas SDK within Python environments to run analytics on graph data.

Please reach out to AWS support or engage your AWS account team if you have questions. We use your feedback to prioritize new features that will meet your needs.

Preparing to migrate from Neo4j to Neptune

Migrating from the Neo4j graph database to the Neptune graph database service can be approached in one of two main ways: re-platforming or refactoring/re-architecting. The re-platforming approach involves modifying the existing data model and application architecture to best leverage Neptune's capabilities, while the refactoring approach focuses on finding equivalent components in Neptune to build a comparable implementation. In practice, a combination of these strategies is often used, as the migration process involves balancing the target Neptune architecture with constraints and requirements from the existing Neo4j implementation. Regardless of the approach, the key is to work backwards from the application's use cases to design the data model, queries, and overall architecture that best address your needs.

Approaches to migrating

When migrating a Neo4j application to Neptune, we recommend one of two strategies: either re-platforming, or refactoring/re-architecting. For more information about migration strategies, see 6 Strategies for Migrating Applications to the Cloud, a blog post by Stephen Orban.

The *re-platforming approach*, sometimes called *lift-tinker-and-shift*, involves the following steps:

- Identify the use cases your application is intended to satisfy.
- Modify the existing graph data model and application architecture to best address these workload needs using Neptune's capabilities.
- Determine how to migrate data, queries and other parts of the source application into the target model and architecture.

This working-backwards approach lets you migrate your application to the kind of Neptune solution you might design if this were a brand-new project.

The *refactoring approach*, by contrast, involves:

- Identifying the components of the existing implementation, including infrastructure, data, queries, and application capabilities.
- Finding equivalents in Neptune that can be used to build a comparable implementation.

This working-forwards approach seeks to swap one implementation for another.

In practice, you're likely to adopt a mix of these two approaches. You might start with a use case, design the target Neptune architecture, but then turn to the existing Neo4j implementation to identify constraints and invariants you'll have to maintain. For example, you might have to continue integrating with other external systems, or continue offering specific APIs to consumers of your graph application. With this information, you can determine what data already exists to move to your target model, and what must be sourced elsewhere.

At other points, you might start by analyzing a specific piece of your Neo4j implementation as the best source of information about the job your application is intended to do. That kind of archaeology in the existing application can help define a use case that you can then design towards using Neptune's capabilities.

Whether you're building a new application using Neptune or migrating an existing application from Neo4j, we recommend working backwards from use cases to design a data model, a set of queries, and an application architecture that address your business needs.

Architectural differences between Neptune and Neo4j

When customers first consider migrating an application from Neo4j to Neptune, it is often tempting to perform a like-to-like comparison based on instance size. However, the architectures of Neo4j and Neptune have fundamental differences. Neo4j is based on an all-in-one approach where data loading, data ETL, application queries, data storage, and management operations all happen in the same set of compute resources, such as EC2 instances.

Neptune, by contrast, is an OLTP focused graph database where the architecture separates responsibilities and where resources are decoupled so they can scale dynamically and independently.

When migrating from Neo4j to Neptune, determine the data durability, availability and scalability requirements of your application. Neptune's cluster architecture simplifies the design of applications that require high durability, availability and scalability. With an understanding of Neptune's cluster architecture, you can then design a Neptune cluster topology to satisfy these requirements.

Neo4j's cluster architecture

Many production applications use Neo4j's causal clustering to provide data durability, high availability and scalability. Neo4j's clustering architecture uses core-server and read-replica instances:

- Core servers provide for data durability and fault tolerance by replicating data using the Raft protocol.
- Read replicas use transaction log shipping to asynchronously replicate data for high read throughput workloads.

Every instance in a cluster, whether core server or read replica, contains a full copy of the graph data.

Neptune's cluster architecture

A Neptune cluster is made up of a primary writer instance and up to 15 read replica instances. All the instances in the cluster share the same underlying distributed storage service that is separate from the instances.

- The primary writer instance coordinates all write operations to the database and is vertically scalable to provide flexible support for different write workloads. It also supports read operations.
- Read replica instances support read operations from the underlying storage volume, and allow you to scale horizontally to support high read workloads. They also provide for high availability by serving as failover targets for the primary instance.

> **Note**
>
> For heavy write workloads, it is best to scale the read replica instances to the same size as the writer instance, to ensure that the readers can stay consistent with the data changes.

- The underlying storage volume scales storage capacity automatically as the data in your database increases, up to 128 tebibytes (TiB) of storage.

Instance sizes are dynamic and independent. Each instance can be resized while the cluster is running, and read replicas can be added or removed while the cluster is running.

The Neptune Serverless feature can scale your compute capacity up and down automatically as demand rises and falls. Not only can this decrease your administrative overhead, it also lets you configure the database to handle large spikes in demand without degrading performance or requiring you to over-provision.

You can stop a Neptune cluster for up to 7 days.

Neptune also supports auto-scaling, to adjust the reader instance sizes automatically based on workload.

Using Neptune's global database feature, you can mirror a cluster in up to 5 other regions.

Neptune is also fault tolerant by design:

- The cluster volume that provides data storage to all the instances in the cluster spans multiple Availability Zones (AZs) in a single AWS Region. Each AZ contains a full copy of the cluster data.
- If the primary instance becomes unavailable, Neptune automatically fails over to an existing read replica with zero data loss, typically in under 30 seconds. If there are no existing read replicas in the cluster, Neptune automatically provisions a new primary instance – again, with zero data loss.

What all this means is that when migrating from a Neo4j causal cluster to Neptune, you don't have to architect the cluster topology explicitly for high data durability and high availability. This leaves you to size your cluster for expected read and write workloads, and any increased availability requirements you may have, in just a few ways:

- To scale read operations, add read replica instances or enable Neptune Serverless functionality.
- To improve availability, distribute the primary instance and read replicas in your cluster over multiple Availability Zones (AZs).
- To reduce any failover time, provision at least one read replica instance that can serve as a failover target for the primary. You can determine the order in which read replica instances are promoted to primary after a failure by assigning each replica a priority. It’s a best practice to ensure that a failover target has an instance class capable of handling your application’s write workload if promoted to primary.

Data storage differences between Neptune and Neo4j

Neptune uses a graph data model based on a native quad model. When migrating your data to Neptune, there are several differences in the architecture of the data model and storage layer that you should be aware of to make optimal use of the distributed and scalable shared storage that Neptune provides:

- Neptune doesn't use any explicitly defined schema or constraints. It lets you add nodes, edges, and properties dynamically without having to define the schema ahead of time. Neptune doesn't limit the values and types of data stored, except as noted in Neptune limits. As part of Neptune's

storage architecture, data is also automatically indexed in a way that handles many of the most common access patterns. This storage architecture removes the operational overhead of creation and management of database schema and index optimization.

- Neptune provides a unique distributed and shared storage architecture that automatically scales in 10 GB chunks as the storage needs of your database grow, up to 128 tebibytes (TiB). This storage layer is reliable, durable, and fault-tolerant, with data copied 6 times, twice in each of 3 Availability Zones. It provides all Neptune clusters with a highly available and fault-tolerant data storage layer by default. Neptune's storage architecture reduces costs and removes the need to provision or over-provision storage to handle future data growth.

Before migrating your data to Neptune, it's good to familiarize yourself with Neptune's property graph data model and transaction semantics.

Operational differences between Neptune and Neo4j

Neptune is a fully managed service that automates many of the normal operational tasks you would have to do when using on-premise or self-managed databases such as Neo4j Enterprise or Community Edition:

- **Automated backups** – Neptune backs up your cluster volume automatically and retains the backup for a retention period that you specify (from 1 to 35 days). These backups are continuous and incremental, so you can quickly restore to any point within the retention period. No performance impact or interruption of database service occurs as backup data is being written.
- **Manual Snapshots** – Neptune lets you make a storage-volume snapshot of your DB cluster to back up the entire DB cluster. This kind of snapshot can then be used to restore the database, make a copy of it, and share it across accounts.
- **Cloning** – Neptune supports a cloning feature that lets you create cost-effective clones of a database quickly. The clones use a copy-on-write protocol to require only minimal additional space after they are created. Database cloning is an effective way to try out new Neptune features or upgrades with no disruption to the originating cluster.
- **Monitoring** – Neptune provides various methods to monitor the performance and usage of your cluster, including:
  - Instance status
  - Integration with Amazon CloudWatch and AWS CloudTrail
  - Audit log capabilities

- Event notifications
- Tagging
- **Security** – Neptune provides a secure environment by default. A cluster resides within a private VPC that provides network isolation from other resources. All traffic is encrypted via SSL, and all data is encrypted at rest using AWS KMS.

  In addition, Neptune integrates with AWS Identity and Access Management (IAM) to provide authentication. By specifying IAM condition keys, you can use IAM policies to provide fine-grained access control over data actions.

Tooling and integration differences between Neptune and Neo4j

Neptune has a different architecture for integrations and tools than Neo4j, which may impact the architecture of your application. Neptune uses the compute resources of the cluster to process queries, but leverages other best-in-class AWS services for functionality like full-text search (using OpenSearch), ETL (using Glue), and so forth. For a full listing of these integrations, see Neptune integrations.

Provisioning infrastructure when migrating from Neo4j to Neptune

Amazon Neptune clusters are built to scale in three dimensions: storage, write capacity, and read capacity. The sections below discuss specific options to consider when migrating.

Provisioning storage

The storage for any Neptune cluster is automatically provisioned, without any administrative overhead on your part. It resizes dynamically in 10 GB chunks as the storage needs of the cluster increase. As a result, there is no need to estimate and provision or over-provision storage to handle future data growth.

Provisioning write capacity

Neptune provides a single writer instance that can be scaled vertically to any instance size available on the Neptune pricing page. When reading and writing data to a writer instance, all transactions are ACID compliant, with data isolation as defined in Transaction Isolation Levels in Neptune.

Choosing an optimal size for a writer instance requires running load tests to determine the optimal instance size for your workload. Any instance within Neptune can be resized at any time by modifying the DB instance class. You can estimate a starting instance size based on concurrency and average query latency as described below in Estimating optimal instance size when provisioning your cluster.

Provisioning read capacity

Neptune is built to scale read-replica instances both horizontally, by adding up to 15 of them within a cluster (or more in a Neptune global database), and vertically to any instance size available on the Neptune pricing page. All Neptune read-replica instances uses the same underlying storage volume, enabling transparent replication of data with minimal lag.

In addition to enabling horizontal scaling of read requests within a Neptune cluster, read replicas also act as failover targets for the writer instance to enable high availability. See Amazon Neptune basic operational guidelines for suggestions about how to determine the appropriate number and placement of read replicas in your cluster.

For applications where connectivity and workload are unpredictable, Neptune also supports an auto-scaling feature that can automatically adjust the number of Neptune replicas based on criteria that you specify.

To determine an optimal size and number of read replica instances requires running load tests to determine the characteristics of the read workload they must support. Any instance within Neptune can be resized at any time by modifying the DB instance class. You can estimate a starting instance size based on concurrency and average query latency, as described in the next section.

Use Neptune Serverless to scale reader and writer instances automatically as needed

While it is often helpful to be able to estimate the compute capacity that your anticipated workloads will require, you can configure the Neptune Serverless feature to scale read and write capacity up and down automatically. This can help you meet peak requirements while also scaling back automatically when demand decreases.

Estimating optimal instance size when provisioning your cluster

Estimation of optimal instance size requires knowing the average query latency in Neptune, when your workload is running, as well as the number of concurrent queries being processed. A rough estimate of instance size can be calculated as the average query latency multiplied by the number of concurrent queries. This gives you the average number of concurrent threads needed to handle the workload.

Each vCPU in a Neptune instance can support two concurrent query threads, so dividing the threads by 2 provides the number of vCPUs required, which can then be correlated to the appropriate instance size on the Neptune pricing page. For example:

```
Average Query Latency:         30ms (0.03s)
Number of concurrent queries:  1000/second

Number of threads needed:      0.03 x 1000 = 30 threads
Number of vCPUs needed:        30 / 2 = 15 vCPUs
```

Correlating this to the number of vCPUs in an instance, we see that we get a rough estimate that a `r5.4xlarge` would be the recommended instance to try for this workload. This estimate is rough and only meant to provide initial guidance on instance-size selection. Any application should go through a right-sizing exercise to determine the appropriate number and type(s) of instances that are appropriate for the workload.

Memory requirements should also be taken into account, as well as processing requirements. Neptune is most performant when the data being accessed by queries is available in the main-memory buffer pool cache. Provisioning sufficient memory can also reduce I/O costs significantly.

Additional details and guidance on sizing of instances in a Neptune cluster can be found on the Sizing DB instances in a Neptune DB cluster page.

Data migration from Neo4j to Neptune

When performing a migration from Neo4j to Amazon Neptune, migrating the data is a major step in the process. There are multiple approaches to migrating data. The correct approach is determined by the needs of the application, the data size, and the type of migration desired. However, many of these migrations all require assessment of the same considerations, of which several are highlighted below.

> **Note**
>
> See the Migrating a Neo4j graph database to Neptune with a fully automated utility in the AWS Database Blog for a complete, step-by-step walkthrough of one example of how to perform an offline data migration.

Assessing data migration from Neo4j to Neptune

The first step when assessing any data migration is to determine how you will migrate the data. The options depend on the architecture of the application being migrated, the data size, and the availability needs during the migration. In general, migrations tend to fall into one of two categories: online or offline.

Offline migrations tend to be the simplest to accomplish, because the application doesn't accept read or write traffic during the migration. After the application stops accepting traffic, the data can be exported, optimized, imported, and the application tested before the application is re-enabled.

Online migrations are more complex, because the application still needs to accept read and write traffic while the data is being migrated. The exact needs of each online migration may differ, but the general architecture would generally be similar to the following:

- A feed of ongoing changes to the database needs to be enabled in Neo4j by configuring Neo4j Streams as a source to a Kafka cluster.
- Once this is completed, an export of the running system can be taken, following the instructions in Exporting data from Neo4j when migrating to Neptune, and the time noted for later correlation to the Kafka topic.
- The exported data is then imported into Neptune, following instructions in Importing data from Neo4j when migrating to Neptune.

- Changed data from the Kafka stream can then be copied to the Neptune cluster using an architecture similar to the one described in Writing to Amazon Neptune from Amazon Kinesis Data Streams. Note that the changes replication can be run in parallel to validate the new application architecture and performance.
- After the data migration is validated, then the application traffic can be redirected to the Neptune cluster and the Neo4j instance can be decommissioned.

Data-model optimizations for migrating from Neo4j to Neptune

Both Neptune and Neo4j support labeled property graphs (LPG). However, Neptune has some architectural and data-model differences that you can take advantage of to optimize performance:

Optimizing node and edge IDs

Neo4j automatically generates numeric long IDs. Using Cypher you can refer to nodes by ID, but this is generally discouraged in favor of looking up nodes by an indexed property.

Neptune allows you to supply your own string-based IDs for vertices and edges. If you don't supply your own IDs, Neptune automatically generates string representations of UUIDs for new edges and vertices.

If you migrate data from Neo4j to Neptune by exporting from Neo4j and then bulk importing into Neptune, you can preserve Neo4j's IDs. The numeric values generated by Neo4j can act as user-supplied IDs when importing into Neptune, where they are represented as strings rather than numeric values.

However, there are circumstances in which you may want to promote a vertex property to become a vertex ID. Just as looking up a node using an indexed property is the fastest way to find a node in Neo4j, looking up a vertex by ID is the fastest way to find a vertex in Neptune. Therefore, if you can identify a suitable vertex property that contains unique values, you should consider replacing the vertex ~id with the nominated property value in your bulk load CSV files. If you do this, you will also have to rewrite any corresponding ~from and ~to edge values in your CSV files.

Schema constraints when migrating data from Neo4j to Neptune

Within Neptune, the only schema constraint available is the uniqueness of the ID of a node or edge. Applications that need to leverage a uniqueness constraint are encouraged to look at this approach for achieving a uniqueness constraint through specifying the node or edge ID. If the application used multiple columns as a uniqueness constraint, the ID may be set to a combination of these

values. For instance, `id=123, code='SEA'` could be represented as `ID='123_SEA'`) to achieve a complex uniqueness constraint.

Edge direction optimization when migrating data from Neo4j to Neptune

When nodes, edges, or properties are added to Neptune,they are automatically indexed in three different ways, with an optional fourth index. Because of how Neptune builds and uses the indices, queries that follow outgoing edges are more efficient than ones that use incoming edges. In terms of Neptune's graph data storage model, these are subject-based searches that use the SPOG index.

If, in migrating your data model and queries to Neptune, you find that your most important queries rely on traversing incoming edges where there is a high degree of fan out, you may want to consider altering your model so that these traversals follow outgoing edges instead, especially when you cannot specify which edge labels to traverse. To do so, reverse the direction of the relevant edges and update the edge labels to reflect the semantics of this direction change. For example, you might change:

```
person_A — parent_of — person_B
   to:
person_B — child_of — person_A
```

To make this change in a bulk-load edge CSV file, simply swap the `~from` and `~to` column headings, and update the values of the `~label` column.

As an alternative to reversing edge direction, you can enable a fourth Neptune index, the OSGP index, which makes traversing incoming edges, or object-based searches, much more efficient. However, enabling this fourth index will lower insert rates and require more storage.

Filtering optimization when migrating data from Neo4j to Neptune

Neptune is optimized to work best when properties are filtered to the most selective property available. When multiple filters are used, the set of matching items is found for each and then the overlap of all matching sets is calculated. When possible, combining multiple properties into a single property minimizes the number of index lookups and decreases the latency of a query.

For example, this query uses two index look-ups and a join:

```
MATCH (n) WHERE n.first_name='John' AND n.last_name='Doe' RETURN n
```

This query retrieves the same information using a single index look-up:

```
MATCH (n) WHERE n.name='John Doe' RETURN n
```

Neptune supports different data types than Neo4j does.

Neo4j data-type mappings into data types that Neptune supports

- **Logical**: `Boolean`

  Map this in Neptune to `Bool` or `Boolean`.

- **Numeric**: `Number`

  Map this in Neptune to the narrowest of the following Neptune openCypher types that can support all values of the numeric property in question:

  ```
  Byte
  Short
  Integer
  Long
  Float
  Double
  ```

- **Text**: `String`

  Map this in Neptune to `String`.

- **Point in time**:

  ```
  Date
  Time
  LocalTime
  DateTime
  LocalDateTime
  ```

  Map these in Neptune to `Date` as UTC, using one of the following ISO-8601 formats that Neptune supports:

  ```
  yyyy-MM-dd
  yyyy-MM-ddTHH:mm
  yyyy-MM-ddTHH:mm:ss
  yyyy-MM-ddTHH:mm:ssZ
  ```

- **Time duration**: `Duration`

  Map this in Neptune to a numeric value for date arithmetic, if necessary.

- **Spatial**: `Point`

  Map this in Neptune into component numeric values, each of which then becomes a separate property, or express as a String value to be interpreted by the client application. Note that Neptune's full-text search integration using OpenSearch lets you index geolocation properties.

Migrating multivalued properties from Neo4j to Neptune

Neo4j allows homogeneous lists of simple types to be stored as properties of both nodes and edges. These lists can contain duplicate values.

Neptune, however, allows only set or single cardinality for vertex properties, and single cardinality for edge properties in property graph data. As a result, there is no straightforward migration of Neo4j node list properties that contain duplicate values into Neptune vertex properties, or of Neo4j relationship-list properties into Neptune edge properties.

Some possible strategies for migrating Neo4j multivalued node properties with duplicate values into Neptune are as follows:

- Discard the duplicate values and convert the multivalued Neo4j node property to a set cardinality Neptune vertex property. Note that the Neptune set may not then reflect the order of items in the original Neo4j multivalued property.
- Convert the multivalued Neo4j node property to a string representation of a JSON-formatted list in a Neptune vertex string property.
- Extract each of the multivalued property values into a separate vertex with a value property, and connect those vertices to the parent vertex using an edge labelled with the property name.

Similarly, possible strategies for migrating Neo4j multivalued relationship properties into Neptune are as follows:

- Convert the multivalued Neo4j relationship property to a string representation of a JSON-formatted list and store it as a Neptune edge string property.
- Refactor the Neo4j relationship into incoming and outgoing Neptune edges attached to an intermediate vertex. Extract each of the multivalued relationship property values into a separate

vertex with a value property and those vertices to this intermediate vertex using an edge labelled with the property name.

Note that a string representation of a JSON-formatted list is opaque to the openCypher query language, although openCypher includes a `CONTAINS` predicate that allows for simple searches inside string values.

Exporting data from Neo4j when migrating to Neptune

When exporting data from Neo4j, use the APOC procedures to export either to CSV or to GraphML. Although it's possible to export to other formats, there are open-source tools for converting CSV data exported from Neo4j to Neptune bulk-load format, and also open-source tools for converting GraphML data exported from Neo4j to Neptune bulk-load format.

You can also export data directly into Amazon S3 using the various APOC procedures. Exporting to an Amazon S3 bucket is disabled by default, but it can be enabled using the procedures highlighted in Exporting to Amazon S3 in the Neo4j APOC documentation.

Importing data from Neo4j when migrating to Neptune

You can import data into Neptune either by using the Neptune bulk loader or by using application logic in a supported query language such as openCypher.

The Neptune bulk loader is the preferred approach to importing large amounts of data because it provides optimized import performance if you follow best practices. The bulk loader supports two different CSV formats, to which data exported from Neo4j can be converted using the the open-source utilities mentioned above in the Exporting data section.

You can also use openCypher to import data with custom logic for parsing, transforming, and importing. You can submit the openCypher queries either through the HTTPS endpoint (which is recommended) or by using the bolt driver.

Application migration from Neo4j to Neptune

After you have migrated your data from Neo4j to Neptune, the next step is to migrate the application itself. As with data, there are multiple approaches to migrating your application based on the tools your use, requirements, architectural differences, and so on. Things that you usually need to consider in this process are outlined below.

Migrating connections when moving from Neo4j to Neptune

If you don't currently use the Bolt drivers, or would like to use an alternative, you can connect to the HTTPS endpoint which provides full access to the data returned.

If you do have an application that uses the Bolt protocol, you can migrate these connections to Neptune and let your applications connect using the same drivers as you did in Neo4j. To connect to Neptune, you may need to make one or more of the following changes to your application:

- The URL and port will need to be updated to use the cluster endpoints and cluster port (the default is 8182).
- Neptune requires all connections to use SSL, so you need to specify for each connection that it is encrypted.
- Neptune manages authentication through the assignment of IAM policies and roles. IAM policies and roles provide an extremely flexible level of user management within the application, so it is important to read and understand the information in the IAM overview before configuring your cluster.
- Bolt connections behave differently in Neptune than in Neo4j in several ways, as explained in Bolt connection behavior in Neptune.
- You can find more information and suggestions in Neptune Best Practices Using openCypher and Bolt.

There are code samples for commonly used language such as Java, Python, .NET, and NodeJS, and for connection scenarios such as using IAM authentication, in Using the Bolt protocol to make openCypher queries to Neptune.

Routing queries to cluster instances when moving from Neo4j to Neptune

Neo4j client applications use a routing driver and specify an access mode to route read and write requests to an appropriate server in a causal cluster.

When migrating a client application to Neptune, use Neptune endpoints to route queries efficiently to an appropriate instance in your cluster:

- All connections to Neptune should use `bolt://` rather than `bolt+routing://` or `neo4j://` in the URL.
- The cluster endpoint connects to the current primary instance in your cluster. Use the cluster endpoint to route write requests to the primary.
- The reader endpoint distributes connections across read-replica instances in your cluster. If you have a single-instance cluster with no read-replica instances, the reader endpoint connects to the primary instance, which supports write operations. If the cluster does contain one or more read-replica instances, sending a write request to the reader endpoint generates an exception.
- Each instance in your cluster can also have its own instance endpoint. Use an instance endpoint if your client application needs to send a request to a specific instance in the cluster.

For more information, see Neptune endpoint considerations.

Data consistency in Neptune

When using Neo4j causal clusters, read replicas are eventually consistent with core servers, but client applications can ensure causal consistency by using causal chaining. Causal chaining entails passing bookmarks between transactions, which allows a client application to write to a core server and then read its own write from a read-replica.

In Neptune, read-replica instances are eventually consistent with the writer, with replica lag that is usually less than 100 milliseconds. However, until a change has been replicated, updates to existing edges and vertices and additions of new edges and vertices are not visible on a replica instance. Therefore, if your application needs immediate consistency on Neptune by reading each write, use the cluster endpoint for the read-after-write operation. This is the only time to use the cluster endpoint for read operations. In all other circumstances, use the reader endpoint for reads.

Migrating queries from Neo4j to Neptune

Although Neptune's support for openCypher dramatically reduces the amount of work required to migrate queries from Neo4j, there are still some differences to assess when migrating:

- As discussed in Data-model optimizations above, there may be modifications to your data model that you need to make so as to create an optimized graph data model for Neptune, which in turn will require changes to your queries and testing.

- Neo4j offers a variety of Cypher-specific language extensions that are not included in the openCypher specification implemented by Neptune. Depending on the use case and feature used, there may be workarounds within the openCypher language, or using the Gremlin language, or through other mechanisms as described in Rewriting Cypher queries to run in openCypher on Neptune.
- Applications often use other middleware components to interact with the database instead of the Bolt drivers themselves. Please check Neptune compatibility with Neo4j to see if tools or middleware that you're using are supported.
- In the case of a failover, the Bolt driver might continue to connect to the previous writer or reader instance because the cluster endpoint provided to the connection has resolved to an IP address. Proper error handling in your application should handle this, as described in Create a new connection after failover.
- When transactions are canceled because of unresolvable conflicts or lock-wait timeouts, Neptune responds with a `ConcurrentModificationException`. For more information, see Engine Error Codes. As a best practice, clients should always catch and handle these exceptions.

  A `ConcurrentModificationException` occurs occasionally when multiple threads or multiple applications are writing to the system simultaneously. Because of transaction isolation levels, these conflicts may sometimes be unavoidable.
- Neptune supports running both Gremlin and openCypher queries on the same data. This means that in some scenarios you may need to consider using Gremlin, with its more powerful querying capabilities, to perform some of the functionality of your queries.

As discussed in Provisioning infrastructure above, each application should go through a right-sizing exercise to ensure that the number of instances, the instance sizes, and the cluster topology are all optimized for the specific workload of the application.

The considerations discussed here for migrating your application are the most common ones, but this is not an exhaustive list. Each application is unique. Please reach out to AWS support or engage your account team if you have further questions.

Migrating features and tools that are specific to Neo4j

Neo4j has a variety of custom features and add-ons with funtionality that your application may rely on. When evaluating the need to migrate this functionality, it often helps to investigate whether there is a better approach within AWS to achieve the same goal. Considering the

architectural differences between Neo4j and Neptune, you can often find effective alternatives that take advantage of other AWS services or integrations.

See Neptune compatibility with Neo4j for a list of Neo4j-specific features and suggested workarounds.

Neptune compatibility with Neo4j

Neo4j's has an all-in-one architectural approach, where data loading, data ETL, application queries, data storage, and management operations all occur in the same set of compute resources, such as EC2 instances. Amazon Neptune is an OLTP-focused open-specifications graph database where the architecture separates operations and decouples resources so they can scale dynamically.

There are a variety of features and tooling in Neo4j, including third-party tooling, that are not part of the openCypher specification, are incompatible with openCypher, or are incompatible with Neptune's implementation of openCypher. Below are listed some of the most common of these.

Neo4j-specific features not present in Neptune

- **`LOAD CSV`** – Neptune has a different architectural approach to loading data than Neo4j. To allow for better scaling and cost optimization, Neptune implements a separation of concerns around resources, and recommends using one of the AWS service integrations such as AWS Glue to perform the required ETL processes to prepare data in a format supported by the Neptune bulk loader.

  Another option is to do the same thing using application code running on AWS compute resources such as Amazon EC2 instances, Lambda functions, Amazon Elastic Container Service, AWS Batch jobs, and so on. The code could use either Neptune's HTTPS endpoint or Bolt endpoint.
- **Fine-grained access control** – Neptune supports granular access control over data-access actions using IAM condition keys. Additional fine-grained access control can be implemented at the application layer.
- **Neo4j Fabric** – Neptune does support query federation across databases for RDF workloads using the SPARQL `SERVICE` keyword. Because there is not currently an open standard or specification for query federation for property graph workloads, that functionality would need to be implemented at the application layer.
- **Role-based access control (RBAC)** – Neptune manages authentication through the assignment of IAM policies and roles. IAM policies and roles provide an extremely flexible level of user management within an application, so it is worth reading and understanding the information in the IAM overview before configuring your cluster.
- **Bookmarking** – Neptune clusters consist of a single writer instance and up to 15 read-replica instances. Data written to the writer instance is ACID compliant and provides a strong

consistency guarantee on subsequent reads. Read-replicas use the same storage volume as the writer instance and are eventually consistent, usually in less than 100ms from the time data is written. If your use case has an immediate need to guarantee read consistency of new writes, these reads should be directed to the cluster endpoint instead of the reader endpoint.

- **APOC procedures** – Because APOC procedures are not included in the openCypher specification, Neptune does not provide direct support for external procedures. Instead, Neptune relies on integrations with other AWS services to achieve similar end user functionality in a scalable, secure, and robust manner. Sometimes APOC procedures can be rewritten in openCypher or Gremlin, and some are not relevant to Neptune applications.

  In general, APOC procedures fall into the categories below:

  - **Import** – Neptune supports importing data using a variety of formats using query languages, the Neptune bulk loader, or as an target of AWS Database Migration Service. ETL operations on data may be performed using AWS Glue and the `neptune-python-utils` open-source package.
  - **Export** – Neptune supports exporting data using the `neptune-export` utility, which supports a variety of common export formats and methods.
  - **Database Integration** – Neptune supports integration with other databases using ETL tools such as AWS Glue or migrations tools such as the AWS Database Migration Service.
  - **Graph Updates** – Neptune supports a rich set of features for updating property-graph data through its support for both the openCypher and the Gremlin query languages. See Cypher rewrites for examples of rewrites of commonly used procedures.
  - **Data Structures** – Neptune supports a rich set of features for updating property-graph data through its support for both the openCypher and the Gremlin query languages. See Cypher rewrites for examples of rewrites of commonly used procedures.
  - **Temporal (Date Time)** – Neptune supports a rich set of features for updating property-graph data through its support for both the openCypher and the Gremlin query languages. See Cypher rewrites for examples of rewrites of commonly used procedures.
  - **Mathematical** – Neptune supports a rich set of features for updating property-graph data through its support for both the openCypher and the Gremlin query languages. See Cypher rewrites for examples of rewrites of commonly used procedures.
  - **Advanced Graph Querying** – Neptune supports a rich set of features for updating property-graph data through its support for both the openCypher and the Gremlin query languages. See Cypher rewrites for examples of rewrites of commonly used procedures.

- **Comparing Graphs** – Neptune supports a rich set of features for updating property-graph data through its support for both the openCypher and the Gremlin query languages. See Cypher rewrites for examples of rewrites of commonly used procedures.
- **Cypher Execution** – Neptune supports a rich set of features for updating property-graph data through its support for both the openCypher and the Gremlin query languages. See Cypher rewrites for examples of rewrites of commonly used procedures.

- **Custom procedures** – Neptune does not support custom procedures created by users. This functionality would have to be implemented at the application layer.
- **Geospatial** – Although Neptune doesn't provide native support for geospatial features, similar functionality can be achieved through integration with other AWS services, as shown in this blog post: Combine Amazon Neptune and Amazon OpenSearch Service for geospatial queries by Ross Gabay and Abhilash Vinod (1 February 2022).
- **Graph Data Science** – Neptune supports graph analytics today through Neptune Analytics, a memory-optimized engine that supports a library of graph analytic algorithms.

  Neptune also provides an integration with the AWS Pandas SDK and several sample notebooks that show how to leverage this integration within Python environments to run analytics on graph data.

- **Schema Constraints** – Within Neptune, the only schema constraint available is the uniqueness of the ID of a node or edge. There is no feature to specify any additional schema constraints, or any additional uniqueness or value constraints on an element in the graph. ID values in Neptune are strings and may be set using Gremlin, like this:

  ```
  g.addV('person').property(id, '1') )
  ```

  Applications that need to leverage the ID as a uniqueness constraint are encouraged to try this approach for achieving a uniqueness constraint. If the application used multiple columns as a uniqueness constraint, the ID may be set to a combination of these values. For example `id=123, code='SEA'` could be represented as `ID='123_SEA'` to achieve a complex uniqueness constraint.
- **Multi-tenancy** – Neptune only supports a single graph per cluster. To build a multi-tenant system using Neptune, either use multiple clusters or logically partition the tenants within a single graph and use application-side logic to enforce separation. For example, add a property `tenantId` and include it in each query, like this:

```
MATCH p=(n {tenantId:1})-[]->({tenantId:1}) RETURN p LIMIT 5)
```

Neptune Serverless makes it relatively easy to implement multi-tenancy using multiple DB clusters, each of which is scaled independently and automatically as needed.

Neptune support for Neo4j tools

Neptune provides the following alternatives to Neo4j tools:

- **Neo4j Browser** – Neptune provides open-source graph notebooks that provide a developer-focused IDE for running queries and visualizing the results.
- **Neo4j Bloom** – Neptune supports rich graph visualizations using third-party visualization solutions such as Graph-explorer, Tom Sawyer, Cambridge Intelligence, Graphistry, metaphacts, and G.V().
- **GraphQL** – Neptune currently supports GraphQL though custom AWS AppSync integrations. See the Build a graph application with Amazon Neptune and AWS Amplify blog post, and the example project, Building Serverless Calorie tracker application with AWS AppSync and Amazon Neptune.
- **NeoSemantics** – Neptune natively supports the RDF data model, so customers wishing to run RDF workloads are advised to use Neptune's RDF model support.
- **Arrows.app** – The Cypher created when exporting the model using the export command is compatible with Neptune.
- **Linkurious Ogma** – A sample integration with Linkurious Ogma is available here.
- **Spring Data Neo4j** – This is not currently compatible with Neptune.
- **Neo4j Spark Connector** – The Neo4j spark connector can be used within a Spark Job to connect to Neptune using openCypher. Here is some sample code and application configuration:

  Sample code:

```
SparkSession spark = SparkSession
            .builder()
            .config("encryption.enabled", "true")
            .appName("Simple Application").config("spark.master",
 "local").getOrCreate();

Dataset<Row> df = spark.read().format("org.neo4j.spark.DataSource")
```

```
            .option("url", "bolt://(your cluster endpoint):8182")
            .option("encryption.enabled", "true")
            .option("query", "MATCH (n:airport) RETURN n")
            .load();

System.out.println("TOTAL RECORD COUNT: " + df.count());
spark.stop();
```

Application configuration:

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-connector-apache-spark_2.12-4.1.0</artifactId>
    <version>4.0.1_for_spark_3</version>
</dependency>
```

Neo4j features and tools not listed here

If you are using a tool or feature that is not listed here, we are unsure of its compatibility with Neptune or other services within AWS. Please reach out to AWS support or engage your account team if you have further questions.

Rewriting Cypher queries to run in openCypher on Neptune

The openCypher language is a declarative query language for property graphs that was originally developed by Neo4j, then open-sourced in 2015, and contributed to the openCypher project under an Apache 2 open-source license. At AWS, we believe that open source is good for everyone and we are committed to bringing the value of open source to our customers, and the operational excellence of AWS to open source communities.

OpenCypher syntax is documented in the Cypher Query Language Reference, Version 9.

Because openCypher contains a subset of the syntax and features of the Cypher query language, some migration scenarios require either rewriting queries in openCypher-compliant forms or examining alternative methods to achieve the desired functionality.

This section contains recommendations for handling common differences, but they are by no means exhaustive. You should test any application using these rewrites thoroughly to ensure that the results are what you expect.

Rewriting None, `All`, and Any predicate functions

> **Note**
>
> Starting with Neptune engine release 1.5.0.0, the `none()`, `all()`, `any()`, and `single()` predicate functions are natively supported in openCypher. If you are using engine release 1.5.0.0 or later, you can use these functions directly without the rewrites described below.

In engine releases before 1.5.0.0, these functions are not part of the openCypher specification. Comparable results can be achieved in openCypher using List Comprehension.

For example, find all the paths that go from node `Start` to node `End`, but no journey is allowed to pass through a node with a class property of `D`:

```
# Neo4J Cypher code
match p=(a:Start)-[:HOP*1..]->(z:End)
where none(node IN nodes(p) where node.class ='D')
return p

# Neptune openCypher code
match p=(a:Start)-[:HOP*1..]->(z:End)
where size([node IN nodes(p) where node.class = 'D']) = 0
```

```
return p
```

List comprehension can achieve these results as follows:

```
all  => size(list_comprehension(list)) = size(list)
any  => size(list_comprehension(list)) >= 1
none => size(list_comprehension(list)) = 0
```

Rewriting the Cypher `reduce()` function in openCypher

The `reduce()` function is not part of the openCypher specification. It is often used to create an aggregation of data from elements within a list. In many cases, you can use a combination of List Comprehension and the `UNWIND` clause to achieve similar results.

For example, the following Cypher query finds all airports on paths having one to three stops between Anchorage (ANC) and Austin (AUS), and returns the total distance of each path:

```
MATCH p=(a:airport {code: 'ANC'})-[r:route*1..3]->(z:airport {code: 'AUS'})
RETURN p, reduce(totalDist=0, r in relationships(p) | totalDist + r.dist) AS totalDist
ORDER BY totalDist LIMIT 5
```

You can write the same query in openCypher for Neptune as follows:

```
MATCH p=(a:airport {code: 'ANC'})-[r:route*1..3]->(z:airport {code: 'AUS'})
UNWIND [i in relationships(p) | i.dist] AS di
RETURN p, sum(di) AS totalDist
ORDER BY totalDist
LIMIT 5
```

Rewriting the Cypher FOREACH clause in openCypher

The FOREACH clause is not part of the openCypher specification. It is often used to update data in the middle of a query, often from aggregations or elements within a path.

As a path example, find all airports on a path with no more than two stops between Anchorage (ANC) and Austin (AUS) and set a property of visited on each of them:

```
# Neo4J Example
MATCH p=(:airport {code: 'ANC'})-[*1..2]->({code: 'AUS'})
FOREACH (n IN nodes(p) | SET n.visited = true)
```

```
# Neptune openCypher
MATCH p=(:airport {code: 'ANC'})-[*1..2]->({code: 'AUS'})
WITH nodes(p) as airports
UNWIND airports as a
SET a.visited=true
```

Another example is:

```
# Neo4J Example
MATCH p=(start)-[*]->(finish)
WHERE start.name = 'A' AND finish.name = 'D'
FOREACH (n IN nodes(p) | SET n.marked = true)

# Neptune openCypher
MATCH p=(start)-[*]->(finish)
WHERE start.name = 'A' AND finish.name = 'D'
UNWIND nodes(p) AS n
SET n.marked = true
```

Rewriting Neo4j APOC procedures in Neptune

The examples below use openCypher to replace some of the most commonly used APOC procedures. These examples are for reference only, and are intended to provide some suggestions about how to handle common scenarios. In practice, each application is different, and you'll have to come up with your own strategies for providing all the functionality you need.

Rewriting `apoc.export` procedures

Neptune provides an array of options for both full graph and query-based exports in various output formats such as CSV and JSON, using the neptune-export utility (see Exporting data from a Neptune DB cluster).

Rewriting `apoc.schema` procedures

Neptune does not have explicitly defined schema, indices, or constraints, so many `apoc.schema` procedures are no longer required. Examples are:

- `apoc.schema.assert`
- `apoc.schema.node.constraintExists`
- `apoc.schema.node.indexExists`,
- `apoc.schema.relationship.constraintExists`

- `apoc.schema.relationship.indexExists`
- `apoc.schema.nodes`
- `apoc.schema.relationships`

Neptune openCypher does support retrieving similar values to those that the procedures do, as shown below, but can run into performance issues on larger graphs as doing so requires scanning a large portion of the graph to return the answer.

```
# openCypher replacement for apoc.schema.properties.distinct
MATCH (n:airport)
RETURN DISTINCT n.runways
```

```
# openCypher replacement for apoc.schema.properties.distinctCount
MATCH (n:airport)
RETURN DISTINCT n.runways, count(n.runways)
```

Alternatives to apoc.do procedures

These procedures are used to provide conditional query execution that is easy to implement using other openCypher clauses. In Neptune there are at least two ways to achieve similar behavior:

- One way is to combine openCypher's List Comprehension capabilities with the `UNWIND` clause.
- Another way is to use the choose() and coalesce() steps in Gremlin.

Examples of these approaches are shown below.

Alternatives to apoc.do.when

```
# Neo4J Example
MATCH (n:airport {region: 'US-AK'})
CALL apoc.do.when(
 n.runways>=3,
 'SET n.is_large_airport=true RETURN n',
 'SET n.is_large_airport=false RETURN n',
 {n:n}
) YIELD value
WITH collect(value.n) as airports
RETURN size([a in airports where a.is_large_airport]) as large_airport_count,
size([a in airports where NOT a.is_large_airport]) as small_airport_count
```

```
# Neptune openCypher
MATCH (n:airport {region: 'US-AK'})
WITH n.region as region, collect(n) as airports
WITH [a IN airports where a.runways >= 3] as large_airports,
[a IN airports where a.runways < 3] as small_airports, airports
UNWIND large_airports as la
SET la.is_large_airport=true
WITH DISTINCT small_airports, airports
UNWIND small_airports as la
    SET la.small_airports=true
WITH DISTINCT airports
RETURN size([a in airports where a.is_large_airport]) as large_airport_count,
size([a in airports where NOT a.is_large_airport]) as small_airport_count

#Neptune Gremlin using choose()
g.V().
  has('airport', 'region', 'US-AK').
  choose(
    values('runways').is(lt(3)),
    property(single, 'is_large_airport', false),
    property(single, 'is_large_airport', true)).
  fold().
  project('large_airport_count', 'small_airport_count').
    by(unfold().has('is_large_airport', true).count()).
    by(unfold().has('is_large_airport', false).count())

 #Neptune Gremlin using coalesce()
g.V().
  has('airport', 'region', 'US-AK').
  coalesce(
    where(values('runways').is(lt(3))).
    property(single, 'is_large_airport', false),
    property(single, 'is_large_airport', true)).
  fold().
  project('large_airport_count', 'small_airport_count').
    by(unfold().has('is_large_airport', true).count()).
    by(unfold().has('is_large_airport', false).count())
```

Alternatives to apoc.do.case

```
# Neo4J Example
```

```
MATCH (n:airport {region: 'US-AK'})
CALL apoc.case([
 n.runways=1, 'RETURN "Has one runway" as b',
 n.runways=2, 'RETURN "Has two runways" as b'
 ],
 'RETURN "Has more than 2 runways" as b'
) YIELD value
RETURN {type: value.b,airport: n}

# Neptune openCypher
MATCH (n:airport {region: 'US-AK'})
WITH n.region as region, collect(n) as airports
WITH [a IN airports where a.runways =1] as single_runway,
[a IN airports where a.runways =2] as double_runway,
[a IN airports where a.runways >2] as many_runway
UNWIND single_runway as sr
    WITH {type: "Has one runway",airport: sr} as res, double_runway, many_runway
WITH DISTINCT double_runway as double_runway, collect(res) as res, many_runway
UNWIND double_runway as dr
    WITH {type: "Has two runways",airport: dr} as two_runways, res, many_runway
WITH collect(two_runways)+res as res, many_runway
UNWIND many_runway as mr
    WITH {type: "Has more than 2 runways",airport: mr} as res2, res, many_runway
WITH collect(res2)+res as res
UNWIND res as r
RETURN r

#Neptune Gremlin using choose()
g.V().
  has('airport', 'region', 'US-AK').
  project('type', 'airport').
    by(
      choose(values('runways')).
        option(1, constant("Has one runway")).
        option(2, constant("Has two runways")).
        option(none, constant("Has more than 2 runways"))).
    by(elementMap())

 #Neptune Gremlin using coalesce()
 g.V().
  has('airport', 'region', 'US-AK').
  project('type', 'airport').
    by(
      coalesce(
```

```
        has('runways', 1).constant("Has one runway"),
        has('runways', 2).constant("Has two runways"),
        constant("Has more than 2 runways"))).
    by(elementMap())
```

Alternatives to List-based properties

Neptune does not currently support storing List-based properties. However, similar results can be obtained by storing list values as a comma separated string and then using the `join()` and `split()` functions to construct and deconstruct the list property.

For example, if we wanted to save a list of tags as a property, we could use the example rewrite which shows how to retrieve a comma separated property and then use the `split()` and `join()` functions with List Comprehension to achieve comparable results:

```
# Neo4j Example (In this example, tags is a durable list of string.
MATCH (person:person {name: "TeeMan"})
WITH person, [tag in person.tags WHERE NOT (tag IN ['test1', 'test2', 'test3'])] AS
 newTags
SET person.tags = newTags
RETURN person

# Neptune openCypher
MATCH (person:person {name: "TeeMan"})
WITH person, [tag in split(person.tags, ',') WHERE NOT (tag IN ['test1', 'test2',
 'test3'])] AS newTags
SET person.tags = join(newTags,',')
RETURN person
```

Rewriting CALL subqueries

Neptune `CALL` subqueries do not support the syntax `CALL (friend) { ... }` for importing variables into the subquery scope (`friend`, in this example). Please use the `WITH` clause inside the subquery for the same, e.g., `CALL { WITH friend ... }`.

Optional `CALL` subqueries are not supported at this moment.

Other differences between Neptune openCypher and Cypher

- Neptune only supports TCP connections for the Bolt protocol. WebSockets connections for Bolt are not supported.

- Neptune openCypher removes whitespace as defined by Unicode in the `trim()`, `ltrim()` and `rtrim()` functions.
- In Neptune openCypher, `tostring(double)` does not automatically switch to E notation for large values of the double.

Resources for migrating from Neo4j to Neptune

Neptune provides several tools and resources that can assist in the migration process.

Tools to help migrate from Neo4j to Neptune

- The openCypher CheatSheet.
- neo4j-to-neptune – A command-line utility for migrating data from Neo4j to Neptune. This tool includes the ability to:
  - Export the data from a properly configured Neo4j graph.
  - Convert that data into Neptune format.
  - Bulk load that data into Neptune.
  - Perform some basic conversions of data during the conversion to Neptune format, such as renaming vertex or edge labels and generating elements.
  - Generate properties for nodes and edges using templates (for example, create an `~id` value using a template such as `Person_{personid}` for situations where you need to create the unique identifier for an element).
- openCypher Query Compatibility Checker – This tool takes an input of openCypher queries and will:
  - Check for compatibility with the selected version of Neptune.
  - Identify specific unsupported functions and clauses with their positions.
  - Suggest replacements if available.
  - Provide error descriptions of any other syntax errors.

Migrating an existing graph from an Apache TinkerPop Gremlin server to Amazon Neptune

If you have graph data in an Apache TinkerPop Gremlin Server that you would like to migrate to Amazon Neptune, you would take the following steps:

1. Export the data from Gremlin server into Amazon Simple Storage Service (Amazon S3).
2. Convert the exported data to a CSV format that the Neptune bulk loader can import.
3. Using Neptune Bulk Loader, import the data into a Neptune DB cluster that you have prepared.
4. Modify your existing application to connect to Neptune's Gremlin endpoint, and make any changes necessary to conform with Neptune Gremlin implementation differences.

Migrating an existing graph from an RDF triple store to Amazon Neptune

If you have graph data in an RDF/SPARQL to migrate to Amazon Neptune, you would take the following steps:

1. Export the data from your RDF triple store.
2. Convert the exported data to a format that the Neptune bulk loader can import.
3. Store the data to be imported in Amazon Simple Storage Service (Amazon S3).
4. Using Neptune Bulk Loader, import the data from Amazon S3 into a Neptune DB cluster that you have prepared.
5. Modify your existing application to connect to Neptune's SPARQL endpoint.

If you want to try out migrating property-graph CSV data into RDF, you can use the Amazon Neptune CSV to RDF converter.

Using AWS Database Migration Service (AWS DMS) to migrate from a relational or NoSQL database to Amazon Neptune

AWS Database Migration Service (AWS DMS) is a cloud service that makes it easy to migrate relational databases, data warehouses, NoSQL databases, and other types of data stores. If you have graph data stored in one of the relational or NoSQL databases that AWS DMS supports, AWS DMS can help you migrate to Neptune quickly and securely, without requiring downtime from your current database. See Using AWS Database Migration Service to load data into Amazon Neptune from a different data store for details.

The migration dataflow using AWS DMS is as follows:

- Create a AWS DMS table-mapping object. This JSON object specifies which tables should be read from your source database and in what order, and how their columns are named. It can also filter the rows being copied and provide simple value transformations such as converting to lower case or rounding.
- Create a Neptune `GraphMappingConfig` to specify how the data extracted from the source database should be loaded into Neptune.
  - For RDF data (queried using SPARQL), the `GraphMappingConfig` is written in the W3's standard R2RML mapping language.
  - For property graph data (queried using Gremlin), the `GraphMappingConfig` is a JSON object, as described in GraphMappingConfig Layout for Property-Graph/Gremlin Data
- Create an AWS DMS replication instance in the same VPC as your Neptune DB cluster, to perform the migration.
- Create an Amazon S3 bucket to be used as intermediary storage for staging the data being migrated.
- Run the AWS DMS migration task.

See Using AWS Database Migration Service to load data into Amazon Neptune from a different data store for the details, and also Chris Smith's four-piece blog post, "Populating your graph in Amazon Neptune from a relational database using AWS Database Migration Service (DMS):"

- Part 1: Setting the stage
- Part 2: Designing the property graph model
- Part 3: Designing the RDF Model

- Part 4: Putting it all together

Migrating from Blazegraph to Amazon Neptune

If you have a graph in the open-source Blazegraph RDF triplestore, you can migrate to your graph data to Amazon Neptune using the following steps:

- *Provision AWS infrastructure.* Begin by provisioning the required Neptune infrastructure using an AWS CloudFormation template (see Create Neptune cluster).
- *Export data from Blazegraph.* There are two main methods for exporting data from Blazegraph, namely using SPARQL CONSTRUCT queries or using the Blazegraph Export utility.
-

This approach is also generally applicable for migrating from other RDF triplestore databases.

Blazegraph to Neptune compatibility

Before migrating your graph data to Neptune, there are several significant differences between Blazegraph and Neptune that you should be aware of. These differences can require changes to queries, the application architecture, or both, or even make migration impractical:

- **`Full-text search`** – In Blazegraph, you can either use internal full-text search or external full-text search capabilities through an integration with Apache Solr. If you use either of these features, stay informed about the latest updates on the full-text search features that Neptune supports. See Neptune full text search.
- **`Query hints`** – Both Blazegraph and Neptune extend SPARQL using the concept of query hints. During a migration, you need to migrate any query hints you use. For information about the latest query hints Neptune supports, see SPARQL query hints.
- **Inference** – Blazegraph supports inference as a configurable option in triples mode, but not in quads mode. Neptune does not yet support inference.
- **Geospatial search** – Blazegraph supports the configuration of namespaces that enable geospatial support. This feature is not yet available in Neptune.
- **Multi-tenancy** – Blazegraph supports multi-tenancy within a single database. In Neptune, multi-tenancy is supported either by storing data in named graphs and using the USING NAMED clauses for SPARQL queries, or by creating a separate database cluster for each tenant.
- **Federation** – Neptune currently supports SPARQL 1.1 federation to locations accessible to the Neptune instance, such as within the private VPC, across VPCs, or to external internet

endpoints. Depending on the specific setup and required federation endpoints, you may need some additional network configuration.

- **Blazegraph standards extensions** – Blazegraph includes multiple extensions to both the SPARQL and REST API standards, whereas Neptune is only compatible with the standards specifications themselves. This may require changes to your application, or make migration difficult.

Provisioning AWS infrastructure for Neptune

Although you can construct the required AWS infrastructure manually through the AWS Management Console or AWS CLI, it's often more convenient to use a CloudFormation template instead, as described below:

Provisioning Neptune using a CloudFormation template:

1. Navigate to Creating an Amazon Neptune cluster using AWS CloudFormation.
2. Choose **Launch Stack** in your preferred region.
3. Set the required parameters (stack name and `EC2SSHKeyPairName`). Also set the following optional parameters to ease the migration process:
   - Set `AttachBulkloadIAMRoleToNeptuneCluster` to true. This parameter allows for creating and attaching the appropriate IAM role to your cluster to allow for bulk loading data.
   - Set `NotebookInstanceType` to your preferred instance type. This parameter creates a Neptune workbook that you use to run the bulk load into Neptune and validate the migration.
4. Choose **Next**.
5. Set any other stack options you want.
6. Choose **Next**.
7. Review your options and select both check boxes to acknowledge that AWS CloudFormation may require additional capabilities.
8. Choose **Create stack**.

The stack creation process can take a few minutes.

Exporting data from Blazegraph

The next step is to export data out of Blazegraph in a format that is compatible with the Neptune bulk loader.

Depending on how the data is stored in Blazegraph (triples or quads) and how many named graphs are in use, Blazegraph may require that you perform the export process multiple times and generate multiple data files:

- If the data is stored as triples, you need to run one export for each named graph.
- If the data is stored as quads, you may choose to either export data in N-Quads format or export each named graph in a triples format.

Below we assume that you export a single namespace as N-Quads, but you can repeat the process for additional namespaces or desired export formats.

If you need Blazegraph to be online and available during the migration, use SPARQL CONSTRUCT queries. This requires that you install, configure, and run a Blazegraph instance with an accessible SPARQL endpoint.

If you don't need Blazegraph to be online, use the BlazeGraph Export utility. To do this you must download Blazegraph, and the data file and configuration files need to be accessible, but the server doesn’t need to be running.

Exporting data from Blazegraph using SPARQL CONSTRUCT

SPARQL CONSTRUCT is a feature of SPARQL that returns an RDF graph matching the a specified query template. For this use case, you use it to export your data one namespace at a time, using a query like the following:

```
CONSTRUCT WHERE { hint:Query hint:analytic "true" . hint:Query
 hint:constructDistinctSPO "false" . ?s ?p ?o }
```

Although other RDF tools exist to export this data, the easiest way to run this query is by using the REST API endpoint provided by Blazegraph. The following script demonstrates how to use a Python (3.6+) script to export data as N-Quads:

```
import requests

# Configure the URL here: e.g. http://localhost:9999/sparql
```

```
url = "http://localhost:9999/sparql"
payload = {'query': 'CONSTRUCT WHERE { hint:Query hint:analytic "true" . hint:Query
 hint:constructDistinctSPO "false" . ?s ?p ?o }'}
# Set the export format to be n-quads
headers = {
'Accept': 'text/x-nquads'
}
# Run the http request
response = requests.request("POST", url, headers=headers, data = payload, files = [])
#open the file in write mode, write the results, and close the file handler
f = open("export.nq", "w")
f.write(response.text)
f.close()
```

If the data is stored as triples, you need to change the `Accept` header parameter to export data in an appropriate format (N-Triples, RDF/XML, or Turtle) using the values specified on the Blazegraph GitHub repo.

Using the Blazegraph export utility to export data

Blazegraph contains a utility method to export data, namely the `ExportKB` class. `ExportKB` facilitates exporting data from Blazegraph, but unlike the previous method, requires that the server be offline while the export is running. This makes it the ideal method to use when you can take Blazegraph offline during migration, or the migration can occur from a backup of the data.

You run the utility from a Java command line on a machine that has Blazegraph installed but not running. The easiest way to run this command is to download the latest blazegraph.jar release located on GitHub. Running this command requires several parameters:

- **`log4j.primary.configuration`** – The location of the log4j properties file.
- **`log4j.configuration`** – The location of the log4j properties file.
- **`output`** – The output directory for the exported data. Files are located as a `tar.gz` in a subdirectory named as documented in the knowledge base.
- **`format`** – The desired output format followed by the location of the `RWStore.properties` file. If you're working with triples, you need to change the `-format` parameter to `N-Triples`, `Turtle`, or `RDF/XML`.

For example, if you have the Blazegraph journal file and properties files, export data as N-Quads using the following code:

```
java -cp blazegraph.jar \
    com.bigdata.rdf.sail.ExportKB \
    -outdir ~/temp/ \
    -format N-Quads \
    ./RWStore.properties
```

If the export is successful, you see output like this:

```
Exporting kb as N-Quads on /home/ec2-user/temp/kb
Effective output directory: /home/ec2-user/temp/kb
Writing /home/ec2-user/temp/kb/kb.properties
Writing /home/ec2-user/temp/kb/data.nq.gz
Done
```

Create an Amazon Simple Storage Service (Amazon S3) bucket and copy the exported data into it

Once you have exported your data from Blazegraph, create an Amazon Simple Storage Service (Amazon S3) bucket in the same Region as the target Neptune DB cluster for the Neptune bulk loader to use to import the data from.

For instructions on how to create an Amazon S3 bucket, see How do I create an S3 Bucket? in the Amazon Simple Storage Service User Guide, and Examples of creating a bucket in the Amazon Simple Storage Service User Guide.

For instructions about how to copy the data files you have exported into the new Amazon S3 bucket, see Uploading an object to a bucket in the Amazon Simple Storage Service User Guide, or Using high-level (s3) commands with the AWS CLI. You can also use Python code like the following to copy the files one by one:

```
import boto3

region = 'region name'
bucket_name = 'bucket name'
s3 = boto3.resource('s3')
s3.meta.client.upload_file('export.nq', bucket_name, 'export.nq')
```

Loading data into Amazon Neptune

There are several different ways to load graph data into Amazon Neptune:

- If you only need to load a relatively small amount of data, you can use queries such as SPARQL `INSERT` statements or Gremlin `mergeV` and `mergeE` steps. openCypher also includes `CREATE` and `MERGE` clauses.
- You can take advantage of Neptune Bulk Loader to ingest large amounts of data that reside in external files. The bulk loader command is faster and has less overhead than the query-language commands. It is optimized for large datasets, and supports both RDF (Resource Description Framework) data and Gremlin data.
- You can use AWS Database Migration Service (AWS DMS) to import data from other data stores (see Using AWS Database Migration Service to load data into Amazon Neptune from a different data store, and AWS Database Migration Service User Guide).
- For smaller datasets in one or a few Amazon S3 files, you can use query-based loading functions to read and process data directly within your queries. See Loading data into Amazon Neptune using queries for more details.

Topics

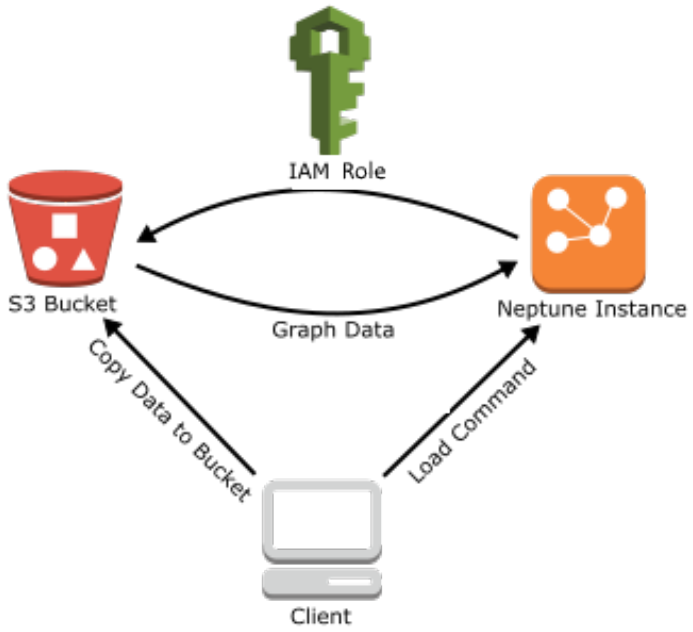
- Using the Amazon Neptune bulk loader to ingest data
- Using AWS Database Migration Service to load data into Amazon Neptune from a different data store
- Loading data into Amazon Neptune using queries

Using the Amazon Neptune bulk loader to ingest data

Amazon Neptune provides a `Loader` command for loading data from external files directly into a Neptune DB cluster. You can use this command instead of executing a large number of `INSERT` statements, `addV` and `addE` steps, or other API calls.

The Neptune **Loader** command is faster, has less overhead, is optimized for large datasets, and supports both Gremlin data and the RDF (Resource Description Framework) data used by SPARQL.

The following diagram shows an overview of the load process:



Here are the steps of the loading process:

1. Copy the data files to an Amazon Simple Storage Service (Amazon S3) bucket.
2. Create an IAM role with Read and List access to the bucket.
3. Create an Amazon S3 VPC endpoint.
4. Start the Neptune loader by sending a request via HTTP to the Neptune DB instance.
5. The Neptune DB instance assumes the IAM role to load the data from the bucket.

> **Note**
>
> You can load encrypted data from Amazon S3 if it was encrypted using either the Amazon S3 `SSE-S3` or the `SSE-KMS` mode, provided that the role you use for bulk load has access to the Amazon S3 object, and also in the case of SSE-KMS, to `kms:decrypt`. Neptune can then impersonate your credentials and issue `s3:getObject` calls on your behalf. However, Neptune does not currently support loading data encrypted using the `SSE-C` mode.

The following sections provide instructions for preparing and loading data into Neptune.

Topics

- Prerequisites: IAM Role and Amazon S3 Access
- Load Data Formats
- Example: Loading Data into a Neptune DB Instance
- Optimizing an Amazon Neptune bulk load
- Fast bulk load (offline loader)
- Neptune Loader Reference

Prerequisites: IAM Role and Amazon S3 Access

Loading data from an Amazon Simple Storage Service (Amazon S3) bucket requires an AWS Identity and Access Management (IAM) role that has access to the bucket. Amazon Neptune assumes this role to load the data.

> **Note**
>
> You can load encrypted data from Amazon S3 if it was encrypted using the Amazon S3 `SSE-S3` mode. In that case, Neptune is able to impersonate your credentials and issue `s3:getObject` calls on your behalf.
>
> You can also load encrypted data from Amazon S3 that was encrypted using the `SSE-KMS` mode, as long as your IAM role includes the necessary permissions to access AWS KMS. Without proper AWS KMS permissions, the bulk load operation fails and returns a `LOAD_FAILED` response.
>
> Neptune does not currently support loading Amazon S3 data encrypted using the `SSE-C` mode.

The following sections show how to use a managed IAM policy to create an IAM role for accessing Amazon S3 resources, and then attach the role to your Neptune cluster.

Topics

- Creating an IAM role to allow Amazon Neptune to access Amazon S3 resources
- Adding the IAM Role to an Amazon Neptune Cluster
- Creating the Amazon S3 VPC Endpoint
- Chaining IAM roles in Amazon Neptune

> **Note**
>
> These instructions require that you have access to the IAM console and permissions to manage IAM roles and policies. For more information, see Permissions for Working in the AWS Management Console in the *IAM User Guide*.
> The Amazon Neptune console requires the user to have the following IAM permissions to attach the role to the Neptune cluster:
>
> ```
> iam:GetAccountSummary on resource: *
> iam:ListAccountAliases on resource: *
> iam:PassRole on resource: * with iam:PassedToService restricted to
 rds.amazonaws.com
> ```

Creating an IAM role to allow Amazon Neptune to access Amazon S3 resources

Use the `AmazonS3ReadOnlyAccess` managed IAM policy to create a new IAM role that will allow Amazon Neptune access to Amazon S3 resources.

To create a new IAM role that allows Neptune access to Amazon S3

1. Open the IAM console at https://eusc-de-east-1.console.amazonaws-eusc.eu/iam/.
2. In the navigation pane, choose **Roles**.
3. Choose **Create role**.
4. Under **AWS service**, choose **S3**.
5. Choose **Next: Permissions**.
6. Use the filter box to filter by the term **S3** and check the box next to **AmazonS3ReadOnlyAccess**.

> **Note**
>
> This policy grants `s3:Get*` and `s3:List*` permissions to all buckets. Later steps restrict access to the role using the trust policy.
> The loader only requires `s3:Get*` and `s3:List*` permissions to the bucket you are loading from, so you can also restrict these permissions by the Amazon S3 resource.
> If your S3 bucket is encrypted, you need to add `kms:Decrypt` permissions

7. Choose **Next: Review**.
8. Set **Role Name** to a name for your IAM role, for example: `NeptuneLoadFromS3`. You can also add an optional **Role Description** value, such as "Allows Neptune to access Amazon S3 resources on your behalf."
9. Choose **Create Role**.
10. In the navigation pane, choose **Roles**.
11. In the **Search** field, enter the name of the role you created, and choose the role when it appears in the list.
12. On the **Trust Relationships** tab, choose **Edit trust relationship**.
13. In the text field, paste the following trust policy.

    JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "rds.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

14. Choose **Update trust policy**.
15. Complete the steps in Adding the IAM Role to an Amazon Neptune Cluster.

Adding the IAM Role to an Amazon Neptune Cluster

Use the console to add the IAM role to an Amazon Neptune cluster. This allows any Neptune DB instance in the cluster to assume the role and load from Amazon S3.

> **Note**
>
> The Amazon Neptune console requires the user to have the following IAM permissions to attach the role to the Neptune cluster:
>
> ```
> iam:GetAccountSummary on resource: *
> iam:ListAccountAliases on resource: *
> iam:PassRole on resource: * with iam:PassedToService restricted to
 rds.amazonaws.com
> ```

To add an IAM role to an Amazon Neptune cluster

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Databases**.
3. Choose the cluster identifier for the cluster that you want to modify.
4. Choose the **Connectivity & Security** tab.
5. In the IAM Roles section, choose the role you created in the previous section.
6. Choose **Add role**.
7. Wait until the IAM role becomes accessible to the cluster before you use it.

Creating the Amazon S3 VPC Endpoint

The Neptune loader requires a VPC endpoint of type Gateway for Amazon S3.

To set up access for Amazon S3

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://eusc-de-east-1.console.amazonaws-eusc.eu/vpc/.
2. In the navigation pane, choose **Endpoints**.
3. Choose **Create Endpoint**.
4. Choose the **Service Name** `com.amazonaws.`*`region`*`.s3` for the Gateway type endpoint.

> **Note**
>
> If the Region here is incorrect, make sure that the console Region is correct.

5. Choose the VPC that contains your Neptune DB instance (it is listed for your DB instance in the Neptune console).
6. Select the check box next to the route tables that are associated with the subnets related to your cluster. If you only have one route table, you must select that box.
7. Choose **Create Endpoint**.

For information about creating the endpoint, see VPC Endpoints in the *Amazon VPC User Guide*. For information about the limitations of VPC endpoints, VPC Endpoints for Amazon S3.

Next Steps

Now that you have granted access to the Amazon S3 bucket, you can prepare to load data. For information about supported formats, see Load Data Formats.

Chaining IAM roles in Amazon Neptune

> **Important**
>
> The new bulk load cross-account feature introduced in engine release 1.2.1.0.R3 that takes advantage of chaining IAM roles may in some cases cause you to observe degraded bulk load performance. As a result, upgrades to engine releases that support this feature have been temporarily suspended until this problem is resolved.

When you attach a role to your cluster, your cluster can assume that role to gain access to data stored in Amazon S3. Starting with engine release 1.2.1.0.R3, if that role doesn't have access to all the resources you need, you can chain one or more additional roles that your cluster can assume to gain access to other resources. Each role in the chain assumes the next role in the chain, until your cluster has assumed the role at the end of chain.

To chain roles, you establish a trust relationship between them. For example, to chain `RoleB` onto `RoleA`, `RoleA` must have a permissions policy that allows it to assume `RoleB`, and `RoleB` must

have a trust policy that allows it to pass its permissions back to `RoleA`. For more information, see Using IAM roles.

The first role in a chain must be attached to the cluster that is loading data.

The first role, and each subsequent role that assumes the following role in the chain, must have:

- A policy that includes a specific statement with the `Allow` effect on the `sts:AssumeRole` action.
- The Amazon Resource Name (ARN) of the next role in a `Resource` element.

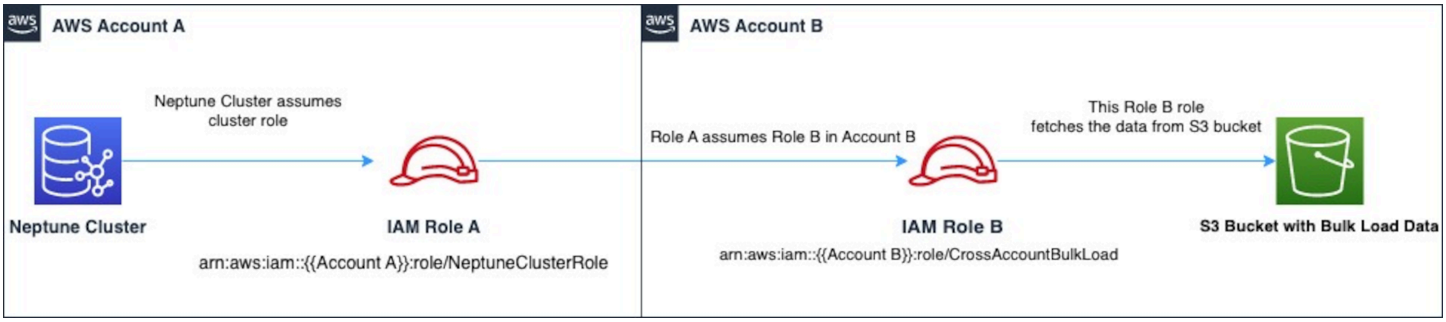
> **Note**
>
> The target Amazon S3 bucket must be in the same AWS Region as the cluster.

Cross-account access using chained roles

You can grant cross-account access by chaining a role or roles that belong to another account. When your cluster temporarily assumes a role belonging to another account, it can gain access to resources there.

For example, suppose **Account A** wants to access data in an Amazon S3 bucket that belongs to **Account B**:

- **Account A** creates an AWS service role for Neptune named `RoleA` and attaches it to a cluster.
- **Account B** creates a role named `RoleB` that's authorized to access the data in an **Account B** bucket.
- **Account A** attaches a permissions policy to `RoleA` that allows it to assume `RoleB`.
- **Account B** attaches a trust policy to `RoleB` that allows it to pass its permissions back to `RoleA`.
- To access the data in the **Account B** bucket, **Account A** runs a loader command using an `iamRoleArn` parameter that chains `RoleA` and `RoleB`. For the duration of the loader operation, `RoleA` then temporarily assumes `RoleB` to access the Amazon S3 bucket in **Account B**.



For example, `RoleA` would have a trust policy that establishes a trust relationship with Neptune:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
          "Service": "rds.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

`RoleA` would also have a permission policy that allows it to assume `RoleB`, which is owned by **Account B**:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1487639602000",
            "Effect": "Allow",
            "Action": [
                "sts:AssumeRole"
            ],
```

```
            "Resource": "arn:aws-eusc:iam::111122223333:role/RoleB"
        }
    ]
}
```

Conversely, `RoleB` would have a trust policy to establish a trust relationship with `RoleA`:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "sts:AssumeRole",
            "Principal": {
                "AWS": "arn:aws-eusc:iam::111122223333:role/RoleA"
            }
        }
    ]
}
```

`RoleB` would also need permission to access data in the Amazon S3 bucket located in **Account B**.

Creating an AWS Security Token Service (STS) VPC endpoint

The Neptune loader requires a VPC endpoint for AWS STS when you are chaining IAM roles to privately access AWS STS APIs through private IP addresses. You can connect directly from an Amazon VPC to AWS STS through a VPC Endpoint in a secure and scalable manner. When you use an interface VPC endpoint, it provides a better security posture because you don't need to open outbound traffic firewalls. It also provides the other benefits of using Amazon VPC endpoints.

When using a VPC Endpoint, traffic to AWS STS does not transmit over the internet and never leaves the Amazon network. Your VPC is securely connected to AWS STS without availability risks or bandwidth constraints on your network traffic. For more information, see Using AWS STS interface VPC endpoints.

To set up access for AWS Security Token Service (STS)

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://eusc-de-east-1.console.amazonaws-eusc.eu/vpc/.
2. In the navigation pane, choose **Endpoints**.
3. Choose **Create Endpoint**.
4. Choose the **Service Name**: `com.amazonaws.region.sts` for the Interface type endpoint.
5. Choose the **VPC** that contains your Neptune DB instance and EC2 instance.
6. Select the check box next to the subnet in which your EC2 instance is present. You can't select multiple subnets from the same Availability Zone.
7. For IP address type, choose from the following options:
   - **IPv4** – Assign IPv4 addresses to your endpoint network interfaces. This option is supported only if all selected subnets have IPv4 address ranges.
   - **IPv6** – Assign IPv6 addresses to your endpoint network interfaces. This option is supported only if all selected subnets are IPv6-only subnets.
   - **Dualstack** – Assign both IPv4 and IPv6 addresses to your endpoint network interfaces. This option is supported only if all selected subnets have both IPv4 and IPv6 address ranges.
8. For **Security groups**, select the security groups to associate with the endpoint network interfaces for the VPC endpoint. You would need to select all the security groups that is attached to your Neptune DB instance and EC2 instance.
9. For **Policy**, select **Full access** to allow all operations by all principals on all resources over the VPC endpoint. Otherwise, select **Custom** to attach a VPC endpoint policy that controls the permissions that principals have for performing actions on resources over the VPC endpoint. This option is available only if the service supports VPC endpoint policies. For more information, see Endpoint policies.
10. (*Optional*) To add a tag, choose **Add new tag** and enter the tag key and the tag value you want.
11. Choose **Create endpoint**.

For information about creating the endpoint, see VPC Endpoints in the Amazon VPC User Guide. Please note that Amazon STS VPC Endpoint is a required prerequisite for IAM role chaining.

Now that you have granted access to the AWS STS endpoint, you can prepare to load data. For information about supported formats, see Load Data Formats.

Chaining roles within a loader command

You can specify role chaining when you run a loader command by including a comma-separated list of role ARNs in the `iamRoleArn` parameter.

Although you'll mostly only need to have two roles in a chain, it is certainly possible to chain three or more together. For example, this loader command chains three roles:

AWS CLI

```
aws neptunedata start-loader-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --source "s3://(the target bucket name)/(the target date file name)" \
  --format "csv" \
  --iam-role-arn "arn:aws:iam::(Account A ID):role/(RoleA),arn:aws:iam::(Account B
 ID):role/(RoleB),arn:aws:iam::(Account C ID):role/(RoleC)" \
  --s3-bucket-region "us-east-1"
```

For more information, see start-loader-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_loader_job(
    source='s3://(the target bucket name)/(the target date file name)',
    format='csv',
    iamRoleArn='arn:aws:iam::(Account A ID):role/(RoleA),arn:aws:iam::(Account B
 ID):role/(RoleB),arn:aws:iam::(Account C ID):role/(RoleC)',
    s3BucketRegion='us-east-1'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/loader \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "source" : "s3://(the target bucket name)/(the target date file name)",
        "iamRoleArn" : "arn:aws:iam::(Account A
 ID):role/(RoleA),arn:aws:iam::(Account B ID):role/(RoleB),arn:aws:iam::(Account C
 ID):role/(RoleC)",
        "format" : "csv",
        "region" : "us-east-1"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -X POST https://your-neptune-endpoint:port/loader \
  -H 'Content-Type: application/json' \
  -d '{
        "source" : "s3://(the target bucket name)/(the target date file name)",
        "iamRoleArn" : "arn:aws:iam::(Account A
 ID):role/(RoleA),arn:aws:iam::(Account B ID):role/(RoleB),arn:aws:iam::(Account C
 ID):role/(RoleC)",
        "format" : "csv",
        "region" : "us-east-1"
      }'
```

Load Data Formats

The Amazon Neptune Load API supports loading data in a variety of formats.

Property-graph load formats

Data loaded in one of the following property-graph formats can then be queried using both Gremlin and openCypher:

- Gremlin load data format (`csv`): a comma-separated values (CSV) format.
- openCypher data load format (`opencypher`): a comma-separated values (CSV) format.

RDF load formats

To load Resource Description Framework (RDF) data that you query using SPARQL, you can use one of the following standard formats as specified by the World Wide Web Consortium (W3C):

- N-Triples (`ntriples`) from the specification at https://www.w3.org/TR/n-triples/.
- N-Quads (`nquads`) from the specification at https://www.w3.org/TR/n-quads/.
- RDF/XML (`rdfxml`) from the specification at https://www.w3.org/TR/rdf-syntax-grammar/.
- Turtle (`turtle`) from the specification at https://www.w3.org/TR/turtle/.

Load data must use UTF-8 encoding

> **Important**
>
> All load-data files must be encoded in UTF-8 form. If a file is not UTF-8 encoded, Neptune tries to load it as UTF-8 anyway.

For N-Quads and N-triples data that includes Unicode characters, `\u`*`xxxxx`* escape sequences are supported. However, Neptune does not support normalization. If a value is present that requires normalization, it will not match byte-to-byte during querying. For more information about normalization, see the Normalization page on Unicode.org.

If your data is not in a supported format, you must convert it before you load it.

A tool for converting GraphML to the Neptune CSV format is available in the GraphML2CSV project on GitHub.

Compression support for load-data files

Neptune supports compression of individual files in `gzip` or `bzip2` format.

The compressed file must have a `.gz` or `.bz2` extension, and must be a single text file encoded in UTF-8 format. You can load multiple files, but each one must be a separate `.gz`, `.bz2`, or uncompressed text file. Archive files with extensions such as `.tar`, `.tar.gz`, and `.tgz` are not supported.

The following sections describe the formats in more detail.

Topics

- Gremlin load data format
- Load format for openCypher data
- RDF load data formats

Gremlin load data format

To load Apache TinkerPop Gremlin data using the CSV format, you must specify the vertices and the edges in separate files.

The loader can load from multiple vertex files and multiple edge files in a single load job.

For each load command, the set of files to be loaded must be in the same folder in the Amazon S3 bucket, and you specify the folder name for the `source` parameter. The file names and file name extensions are not important.

The Amazon Neptune CSV format follows the RFC 4180 CSV specification. For more information, see Common Format and MIME Type for CSV Files on the Internet Engineering Task Force (IETF) website.

> **Note**
>
> All files must be encoded in UTF-8 format.

Each file has a comma-separated header row. The header row consists of both system column headers and property column headers.

System Column Headers

The required and allowed system column headers are different for vertex files and edge files.

Each system column can appear only once in a header.

All labels are case sensitive.

Vertex headers

- `~id` - **Required**

  An ID for the vertex.
- `~label`

  A label for the vertex. Multiple label values are allowed, separated by semicolons (`;`).

  If `~label` is not present, TinkerPop supplies a label with the value `vertex`, because every vertex must have at least one label.

Edge headers

- `~id` - **Required**

  An ID for the edge.
- `~from` - **Required**

  The vertex ID of the *from* vertex.
- `~to` - **Required**

  The vertex ID of the *to* vertex.
- `~label`

  A label for the edge. Edges can only have a single label.

  If `~label` is not present, TinkerPop supplies a label with the value `edge`, because every edge must have a label.

Property Column Headers

You can specify a column (`:`) for a property by using the following syntax. The type names are not case sensitive. Note, however, that if a colon appears within a property name, it must be escaped by preceding it with a backslash: `\:`.

```
propertyname:type
```

> **Note**
>
> Space, comma, carriage return and newline characters are not allowed in the column headers, so property names cannot include these characters.

You can specify a column for an array type by adding `[]` to the type:

```
propertyname:type[]
```

> **Note**
>
> Edge properties can only have a single value and will cause an error if an array type is specified or a second value is specified.

The following example shows the column header for a property named `age` of type `Int`.

```
age:Int
```

Every row in the file would be required to have an integer in that position or be left empty.

Arrays of strings are allowed, but strings in an array cannot include the semicolon (`;`) character unless it is escaped using a backslash (like this: `\;`).

Specifying the Cardinality of a Column

The column header can be used to specify *cardinality* for the property identified by the column. This allows the bulk loader to honor cardinality similarly to the way Gremlin queries do.

You specify the cardinality of a column like this:

```
propertyname:type(cardinality)
```

The *`cardinality`* value can be either `single` or `set`. The default is assumed to be `set`, meaning that the column can accept multiple values. In the case of edge files, cardinality is always single and specifying any other cardinality causes the loader to throw an exception.

If the cardinality is `single`, the loader throws an error if a previous value is already present when a value is loaded, or if multiple values are loaded. This behavior can be

overridden so that an existing value is replaced when a new value is loaded by using the `updateSingleCardinalityProperties` flag. See Loader Command.

It is possible to use a cardinality setting with an array type, although this is not generally necessary. Here are the possible combinations:

- `name:type` – the cardinality is `set`, and the content is single-valued.
- `name:type[]` – the cardinality is `set`, and the content is multi-valued.
- `name:type(single)` – the cardinality is `single`, and the content is single-valued.
- `name:type(set)` – the cardinality is `set`, which is the same as the default, and the content is single-valued.
- `name:type(set)[]` – the cardinality is `set`, and the content is multi-valued.
- `name:type(single)[]` – this is contradictory and causes an error to be thrown.

The following section lists all the available Gremlin data types.

Gremlin Data Types

This is a list of the allowed property types, with a description of each type.

Bool (or Boolean)

Indicates a Boolean field. Allowed values: `false`, `true`

> **Note**
>
> Any value other than `true` will be treated as false.

Whole Number Types

Values outside of the defined ranges result in an error.

Type	Range
Byte	-128 to 127
Short	-32768 to 32767

Int	-2^31 to 2^31-1
Long	-2^63 to 2^63-1

Decimal Number Types

Supports both decimal notation or scientific notation. Also allows symbols such as (+/-) Infinity or NaN. INF is not supported.

Type	Range
Float	32-bit IEEE 754 floating point
Double	64-bit IEEE 754 floating point

Float and double values that are too long are loaded and rounded to the nearest value for 24-bit (float) and 53-bit (double) precision. A midway value is rounded to 0 for the last remaining digit at the bit level.

String

Quotation marks are optional. Commas, newline, and carriage return characters are automatically escaped if they are included in a string surrounded by double quotation marks (`"`). *Example:* `"Hello, World"`

To include quotation marks in a quoted string, you can escape the quotation mark by using two in a row: *Example:* `"Hello ""World"""`

Arrays of strings are allowed, but strings in an array cannot include the semicolon (`;`) character unless it is escaped using a backslash (like this: `\;`).

If you want to surround strings in an array with quotation marks, you must surround the whole array with one set of quotation marks. *Example:* `"String one; String 2; String 3"`

Date

Java date in ISO-8601 format. Supports the following formats: `yyyy-MM-dd`, `yyyy-MM-ddTHH:mm`, `yyyy-MM-ddTHH:mm:ss`, `yyyy-MM-ddTHH:mm:ssZ`. The values are converted to epoch time and stored.

Datetime

Java date in ISO-8601 format. Supports the following formats: `yyyy-MM-dd`, `yyyy-MM-ddTHH:mm`, `yyyy-MM-ddTHH:mm:ss`, `yyyy-MM-ddTHH:mm:ssZ`. The values are converted to epoch time and stored.

Gremlin Row Format

Delimiters

Fields in a row are separated by a comma. Records are separated by a newline or a newline followed by a carriage return.

Blank Fields

Blank fields are allowed for non-required columns (such as user-defined properties). A blank field still requires a comma separator. Blank fields on required columns will result in a parsing error. Empty string values are interpreted as empty string value for the field; not as a blank field. The example in the next section has a blank field in each example vertex.

Vertex IDs

`~id` values must be unique for all vertices in every vertex file. Multiple vertex rows with identical `~id` values are applied to a single vertex in the graph. Empty string (`""`) is a valid id, and the vertex is created with an empty string as the id.

Edge IDs

Additionally, `~id` values must be unique for all edges in every edge file. Multiple edge rows with identical `~id` values are applied to the single edge in the graph. Empty string (`""`) is a valid id, and the edge is created with an empty string as the id.

Labels

Labels are case sensitive and cannot be empty. A value of `""` will result in an error.

String Values

Quotation marks are optional. Commas, newline, and carriage return characters are automatically escaped if they are included in a string surrounded by double quotation marks (`"`). Empty string values (`""`) are interpreted as an empty string value for the field; not as a blank field.

CSV Format Specification

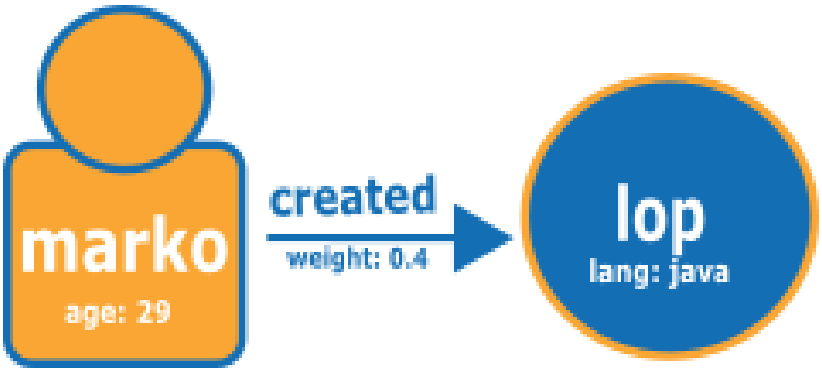
The Neptune CSV format follows the RFC 4180 CSV specification, including the following requirements.

- Both Unix and Windows style line endings are supported (\n or \r\n).
- Any field can be quoted (using double quotation marks).
- Fields containing a line-break, double-quote, or commas must be quoted. (If they are not, load aborts immediately.)
- A double quotation mark character (") in a field must be represented by two (double) quotation mark characters. For example, a string `Hello "World"` must be present as `"Hello ""World"""` in the data.
- Surrounding spaces between delimiters are ignored. If a row is present as `value1, value2`, they are stored as `"value1"` and `"value2"`.
- Any other escape characters are stored verbatim. For example, `"data1\tdata2"` is stored as `"data1\tdata2"`. No further escaping is needed as long as these characters are enclosed within quotation marks.
- Blank fields are allowed. A blank field is considered an empty value.
- Multiple values for a field are specified with a semicolon (`;`) between values.

For more information, see Common Format and MIME Type for CSV Files on the Internet Engineering Task Force (IETF) website.

Gremlin Example

The following diagram shows an example of two vertices and an edge taken from the TinkerPop Modern Graph.



The following is the graph in Neptune CSV load format.

Vertex file:

```
~id,name:String,age:Int,lang:String,interests:String[],~label
v1,"marko",29,,"sailing;graphs",person
v2,"lop",,"java",,software
```

Tabular view of the vertex file:

~id	name:String	age:Int	lang:String	interests :String[]	~label
v1	"marko"	29		["sailing", "graphs"]	person
v2	"lop"		"java"		software

Edge file:

```
~id,~from,~to,~label,weight:Double
e1,v1,v2,created,0.4
```

Tabular view of the edge file:

~id	~from	~to	~label	weight:Double
e1	v1	v2	created	0.4

Next Steps

Now that you know more about the loading formats, see Example: Loading Data into a Neptune DB Instance.

Load format for openCypher data

To load openCypher data using the openCypher CSV format, you must specify nodes and relationships in separate files. The loader can load from multiple of these node files and relationship files in a single load job.

For each load command, the set of files to be loaded must have the same path prefix in an Amazon Simple Storage Service bucket. You specify that prefix in the source parameter. The actual file names and extensions are not important.

In Amazon Neptune, the openCypher CSV format conforms to the RFC 4180 CSV specification. For more information, see Common Format and MIME Type for CSV Files (https://tools.ietf.org/html/rfc4180) on the Internet Engineering Task Force (IETF) website.

> **Note**
>
> These files MUST be encoded in UTF-8 format.

Each file has a comma-separated header row that contains both system column headers and property column headers.

System column headers in openCypher data loading files

A given system column can only appear once in each file. All system column header labels are case-sensitive.

The system column headers that are required and allowed are different for openCypher node load files and relationship load files:

System column headers in node files

- **`:ID`** – (Required) An ID for the node.

  An optional ID space can be added to the node `:ID` column header like this: `:ID(`*`ID Space`*`)`. An example is `:ID(movies)`.

  When loading relationships that connect the nodes in this file, use the same ID spaces in the relationship files' `:START_ID` and/or `:END_ID` columns.

  The node `:ID` column can optionally be stored as a property in the form, *`property name`*`:ID`. An example is `name:ID`.

  Node IDs should be unique across all node files in the current and previous loads. If an ID space is used, node IDs should be unique across all node files that use the same ID space in the current and previous loads.
- **`:LABEL`** – A label for the node.

When using multiple label values for a single node, each label should be separated by semicolons(`;`).

System column headers in relationship files

- **`:ID`** – An ID for the relationship. This is required when `userProvidedEdgeIds` is true (the default), but invalid when `userProvidedEdgeIds` is `false`.

  Relationship IDs should be unique across all relationship files in current and previous loads.
- **`:START_ID`** – (*Required*) The node ID of the node this relationship starts from.

  Optionally, an ID space can be associated with the start ID column in the form `:START_ID(`*`ID Space`*`)`. The ID space assigned to the start node ID should match the ID space assigned to the node in its node file.
- **`:END_ID`** – (*Required*) The node ID of the node this relationship ends at.

  Optionally, an ID space can be associated with the end ID column in the form `:END_ID(`*`ID Space`*`)`. The ID space assigned to the end node ID should match the ID space assigned to the node in its node file.
- **`:TYPE`** – A type for the relationship. Relationships can only have a single type.

> **Note**
>
> See Loading openCypher data for information about how duplicate node or relationship IDs are handled by the bulk load process.

Property column headers in openCypher data loading files

You can specify that a column holds the values for a particular property using a property column header in the following form:

```
propertyname:type
```

Space, comma, carriage return and newline characters are not allowed in the column headers, so property names cannot include these characters. Here is an example of a column header for a property named `age` of type `Int`:

```
age:Int
```

The column with `age:Int` as a column header would then have to contain either an integer or an empty value in every row.

Data types in Neptune openCypher data loading files

- **`Bool`** or **`Boolean`** – A Boolean field. Allowed values are `true` and `false`.

  Any value other than `true` is treated as `false`.
- **`Byte`** – A whole number in the range `-128` through `127`.
- **`Short`** – A whole number in the range `-32,768` through `32,767`.
- **`Int`** – A whole number in the range `-2^31` through `2^31 - 1`.
- **`Long`** – A whole number in the range `-2^63` through `2^63 - 1`.
- **`Float`** – A 32-bit IEEE 754 floating point number. Decimal notation and scientific notation are both supported. `Infinity`, `-Infinity`, and `NaN` are all recognized, but `INF` is not.

  Values with too many digits to fit are rounded to the nearest value (a midway value is rounded to 0 for the last remaining digit at the bit level).
- **`Double`** – A 64-bit IEEE 754 floating point number. Decimal notation and scientific notation are both supported. `Infinity`, `-Infinity`, and `NaN` are all recognized, but `INF` is not.

  Values with too many digits to fit are rounded to the nearest value (a midway value is rounded to 0 for the last remaining digit at the bit level).
- **`String`** – Quotation marks are optional. Comma, newline, and carriage return characters are automatically escaped if they are included in a string that is surrounded by double quotation marks (`"`) like `"Hello, World"`.

  You can include quotation marks in a quoted string by using two in a row, like `"Hello ""World"""`.
- **`DateTime`** – A Java date in one of the following ISO-8601 formats:
  - `yyyy-MM-dd`
  - `yyyy-MM-ddTHH:mm`
  - `yyyy-MM-ddTHH:mm:ss`
  - `yyyy-MM-ddTHH:mm:ssZ`

Auto-cast data types in Neptune openCypher data loading files

Auto-cast data types are provided to load data types not currently supported natively by Neptune. Data in such columns are stored as strings, verbatim with no verification against their intended formats. The following auto-cast data types are allowed:

- **`Char`** – A `Char` field. Stored as a string.
- **`Date`**, **`LocalDate`**, and **`LocalDateTime`**, – See Neo4j Temporal Instants for a description of the `date`, `localdate`, and `localdatetime` types. The values are loaded verbatim as strings, without validation.
- **`Duration`** – See the Neo4j Duration format. The values are loaded verbatim as strings, without validation.
- **Point** – A point field, for storing spatial data. See Spatial instants. The values are loaded verbatim as strings, without validation.

Example of the openCypher load format

The following diagram taken from the TinkerPop Modern Graph shows an example of two nodes and a relationship:

The following is the graph in the normal Neptune openCypher load format.

Node file:

```
:ID,name:String,age:Int,lang:String,:LABEL
v1,"marko",29,,person
v2,"lop",,"java",software
```

Relationship file:

```
:ID,:START_ID,:END_ID,:TYPE,weight:Double
e1,v1,v2,created,0.4
```

Alternatively, you could use ID spaces and ID as a property, as follows:

First node file:

```
name:ID(person),age:Int,lang:String,:LABEL
"marko",29,,person
```

Second node file:

```
name:ID(software),age:Int,lang:String,:LABEL
"lop",,"java",software
```

Relationship file:

```
:ID,:START_ID(person),:END_ID(software),:TYPE,weight:Double
e1,"marko","lop",created,0.4
```

RDF load data formats

To load Resource Description Framework (RDF) data, you can use one of the following standard formats as specified by the World Wide Web Consortium (W3C):

- N-Triples (`ntriples`) from the specification at https://www.w3.org/TR/n-triples/
- N-Quads (`nquads`) from the specification at https://www.w3.org/TR/n-quads/
- RDF/XML (`rdfxml`) from the specification at https://www.w3.org/TR/rdf-syntax-grammar/
- Turtle (`turtle`) from the specification at https://www.w3.org/TR/turtle/

> **Important**
>
> All files must be encoded in UTF-8 format.
> For N-Quads and N-triples data that includes Unicode characters, \u*xxxxx* escape sequences are supported. However, Neptune does not support normalization. If a value is present that requires normalization, it will not match byte-to-byte during querying. For more information about normalization, see the Normalization page on Unicode.org.

Next Steps

Now that you know more about the loading formats, see Example: Loading Data into a Neptune DB Instance.

Example: Loading Data into a Neptune DB Instance

This example shows how to load data into Amazon Neptune. Unless stated otherwise, you must follow these steps from an Amazon Elastic Compute Cloud (Amazon EC2) instance in the same Amazon Virtual Private Cloud (VPC) as your Neptune DB instance.

Prerequisites for the Data Loading Example

Before you begin, you must have the following:

- A Neptune DB instance.

  For information about launching a Neptune DB instance, see Creating an Amazon Neptune cluster.
- An Amazon Simple Storage Service (Amazon S3) bucket to put the data files in.

  You can use an existing bucket. If you don't have an S3 bucket, see Create a Bucket in the *Amazon S3 Getting Started Guide*.
- Graph data to load, in one of the formats supported by the Neptune loader:

  If you are using Gremlin to query your graph, Neptune can load data in a comma-separated-values (CSV) format, as described in Gremlin load data format.

  If you are using openCypher to query your graph, Neptune can also load data in an openCypher-specific CSV format, as described in Load format for openCypher data.

  If you are using SPARQL, Neptune can load data in a number of RDF formats, as described in RDF load data formats.
- An IAM role for the Neptune DB instance to assume that has an IAM policy that allows access to the data files in the S3 bucket. The policy must grant Read and List permissions.

  For information about creating a role that has access to Amazon S3 and then associating it with a Neptune cluster, see Prerequisites: IAM Role and Amazon S3 Access.

> **Note**
>
> The Neptune Load API needs read access to the data files only. The IAM policy doesn't need to allow write access or access to the entire bucket.

- An Amazon S3 VPC endpoint. For more information, see the Creating an Amazon S3 VPC Endpoint section.

Creating an Amazon S3 VPC Endpoint

The Neptune loader requires a VPC endpoint for Amazon S3.

To set up access for Amazon S3

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://eusc-de-east-1.console.amazonaws-eusc.eu/vpc/.
2. In the left navigation pane, choose **Endpoints**.
3. Choose **Create Endpoint**.
4. Choose the **Service Name** `com.amazonaws.`*`region`*`.s3`.

   > **Note**
   >
   > If the Region here is incorrect, make sure that the console Region is correct.

5. Choose the VPC that contains your Neptune DB instance.
6. Select the check box next to the route tables that are associated with the subnets related to your cluster. If you only have one route table, you must select that box.
7. Choose **Create Endpoint**.

For information about creating the endpoint, see VPC Endpoints in the *Amazon VPC User Guide*. For information about the limitations of VPC endpoints, VPC Endpoints for Amazon S3.

To load data into a Neptune DB instance

1. Copy the data files to an Amazon S3 bucket. The S3 bucket must be in the same AWS Region as the cluster that loads the data.

You can use the following AWS CLI command to copy the files to the bucket.

> **Note**
>
> This command does not need to be run from the Amazon EC2 instance.

```
aws s3 cp data-file-name s3://bucket-name/object-key-name
```

> **Note**
>
> In Amazon S3, an **object key name** is the entire path of a file, including the file name. *Example:* In the command `aws s3 cp datafile.txt s3://examplebucket/mydirectory/datafile.txt`, the object key name is **`mydirectory/datafile.txt`**.

Alternatively, you can use the AWS Management Console to upload files to the S3 bucket. Open the Amazon S3 console at https://eusc-de-east-1.console.amazonaws-eusc.eu/s3/, and choose a bucket. Choose **Upload** to upload files.

2. From a command line window, enter the following to run the Neptune loader, using the correct values for your endpoint, Amazon S3 path, format, and IAM role ARN.

   The `format` parameter can be any of the following values: `csv` for Gremlin, `opencypher` for openCypher, or `ntriples`, `nquads`, `turtle`, and `rdfxml` for RDF. For information about the other parameters, see Neptune Loader Command.

   For information about finding the hostname of your Neptune DB instance, see the Connecting to Amazon Neptune Endpoints section.

   The Region parameter must match the Region of the cluster and the S3 bucket.

   Amazon Neptune is available in the following AWS Regions:

   - US East (N. Virginia): `us-east-1`
   - US East (Ohio): `us-east-2`
   - US West (N. California): `us-west-1`

- US West (Oregon): `us-west-2`
- Canada (Central): `ca-central-1`
- Canada West (Calgary): `ca-west-1`
- South America (São Paulo): `sa-east-1`
- Europe (Stockholm): `eu-north-1`
- Europe (Spain): `eu-south-2`
- Europe (Ireland): `eu-west-1`
- Europe (London): `eu-west-2`
- Europe (Paris): `eu-west-3`
- Europe (Frankfurt): `eu-central-1`
- Middle East (Bahrain): `me-south-1`
- Middle East (UAE): `me-central-1`
- Israel (Tel Aviv): `il-central-1`
- Africa (Cape Town): `af-south-1`
- Asia Pacific (Hong Kong): `ap-east-1`
- Asia Pacific (Tokyo): `ap-northeast-1`
- Asia Pacific (Seoul): `ap-northeast-2`
- Asia Pacific (Osaka): `ap-northeast-3`
- Asia Pacific (Singapore): `ap-southeast-1`
- Asia Pacific (Sydney): `ap-southeast-2`
- Asia Pacific (Jakarta): `ap-southeast-3`
- Asia Pacific (Melbourne): `ap-southeast-4`
- Asia Pacific (Malaysia): `ap-southeast-5`
- Asia Pacific (Mumbai): `ap-south-1`
- Asia Pacific (Hyderabad): `ap-south-2`
- China (Beijing): `cn-north-1`
- China (Ningxia): `cn-northwest-1`
- AWS GovCloud (US-West): `us-gov-west-1`
- AWS GovCloud (US-East): `us-gov-east-1`

AWS CLI

```
aws neptunedata start-loader-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --source "s3://bucket-name/object-key-name" \
  --format "format" \
  --iam-role-arn "arn:aws:iam::account-id:role/role-name" \
  --s3-bucket-region "region" \
  --no-fail-on-error \
  --parallelism "MEDIUM" \
  --no-update-single-cardinality-properties \
  --queue-request \
  --dependencies "load_A_id" "load_B_id"
```

For more information, see start-loader-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_loader_job(
    source='s3://bucket-name/object-key-name',
    format='format',
    iamRoleArn='arn:aws:iam::account-id:role/role-name',
    s3BucketRegion='region',
    failOnError=False,
    parallelism='MEDIUM',
    updateSingleCardinalityProperties=False,
    queueRequest=True,
    dependencies=['load_A_id', 'load_B_id']
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/loader \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "source" : "s3://bucket-name/object-key-name",
        "format" : "format",
        "iamRoleArn" : "arn:aws:iam::account-id:role/role-name",
        "region" : "region",
        "failOnError" : "FALSE",
        "parallelism" : "MEDIUM",
        "updateSingleCardinalityProperties" : "FALSE",
        "queueRequest" : "TRUE",
        "dependencies" : ["load_A_id", "load_B_id"]
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -X POST https://your-neptune-endpoint:port/loader \
  -H 'Content-Type: application/json' \
  -d '{
        "source" : "s3://bucket-name/object-key-name",
        "format" : "format",
        "iamRoleArn" : "arn:aws:iam::account-id:role/role-name",
        "region" : "region",
        "failOnError" : "FALSE",
        "parallelism" : "MEDIUM",
        "updateSingleCardinalityProperties" : "FALSE",
        "queueRequest" : "TRUE",
        "dependencies" : ["load_A_id", "load_B_id"]
      }'
```

For information about creating and associating an IAM role with a Neptune cluster, see Prerequisites: IAM Role and Amazon S3 Access.

Note

See Neptune Loader Request Parameters) for detailed information about load request parameters. In brief:
The `source` parameter accepts an Amazon S3 URI that points to either a single file or a folder. If you specify a folder, Neptune loads every data file in the folder.
The folder can contain multiple vertex files and multiple edge files.
The URI can be in any of the following formats.

- `s3://`*`bucket_name`*`/`*`object-key-name`*
- `https://s3.amazonaws.com/`*`bucket_name`*`/`*`object-key-name`*
- `https://s3-us-east-1.amazonaws.com/`*`bucket_name`*`/`*`object-key-name`*

The `format` parameter can be one of the following:

- Gremlin CSV format (`csv`) for Gremlin property graphs
- openCypher CSV format (`opencypher`) for openCypher property graphs
- N -Triples (`ntriples`) format for RDF / SPARQL
- N-Quads (`nquads`) format for RDF / SPARQL
- RDF/XML (`rdfxml`) format for RDF / SPARQL
- Turtle (`turtle`) format for RDF / SPARQL

The optional `parallelism` parameter lets you restrict the number of threads used in the bulk load process. It can be set to `LOW`, `MEDIUM`, `HIGH`, or `OVERSUBSCRIBE`.
When `updateSingleCardinalityProperties` is set to `"FALSE"`, the loader returns an error if more than one value is provided in a source file being loaded for an edge or single-cardinality vertex property.
Setting `queueRequest` to `"TRUE"` causes the load request to be placed in a queue if there is already a load job running.

The `dependencies` parameter makes execution of the load request contingent on the successful completion of one or more load jobs that have already been placed in the queue.

3. The Neptune loader returns a job `id` that allows you to check the status or cancel the loading process; for example:

```
{
    "status" : "200 OK",
    "payload" : {
        "loadId" : "ef478d76-d9da-4d94-8ff1-08d9d4863aa5"
    }
}
```

4. Enter the following to get the status of the load with the `loadId` from **Step 3**:

AWS CLI

```
aws neptunedata get-loader-job-status \
  --endpoint-url https://your-neptune-endpoint:port \
  --load-id ef478d76-d9da-4d94-8ff1-08d9d4863aa5
```

For more information, see get-loader-job-status in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_loader_job_status(
    loadId='ef478d76-d9da-4d94-8ff1-08d9d4863aa5'
)

print(response)
```

awscurl

```
awscurl 'https://your-neptune-endpoint:port/loader/ef478d76-
d9da-4d94-8ff1-08d9d4863aa5' \
  --region us-east-1 \
  --service neptune-db
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -G 'https://your-neptune-endpoint:port/loader/ef478d76-
d9da-4d94-8ff1-08d9d4863aa5'
```

If the status of the load lists an error, you can request more detailed status and a list of the errors. For more information and examples, see Neptune Loader Get-Status API.

5. (Optional) Cancel the `Load` job.

   Enter the following to `Delete` the loader job with the job `id` from **Step 3**:

   AWS CLI

   ```
   aws neptunedata cancel-loader-job \
     --endpoint-url https://your-neptune-endpoint:port \
     --load-id ef478d76-d9da-4d94-8ff1-08d9d4863aa5
   ```

   For more information, see cancel-loader-job in the AWS CLI Command Reference.

   SDK

   ```
   import boto3
   from botocore.config import Config

   client = boto3.client(
   ```

```
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.cancel_loader_job(
    loadId='ef478d76-d9da-4d94-8ff1-08d9d4863aa5'
)

print(response)
```

awscurl

```
awscurl 'https://your-neptune-endpoint:port/loader/ef478d76-
d9da-4d94-8ff1-08d9d4863aa5' \
  --region us-east-1 \
  --service neptune-db \
  -X DELETE
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl -X DELETE 'https://your-neptune-endpoint:port/loader/ef478d76-
d9da-4d94-8ff1-08d9d4863aa5'
```

The `DELETE` command returns the HTTP code `200 OK` upon successful cancellation.

The data from files from the load job that has finished loading is not rolled back. The data remains in the Neptune DB instance.

Optimizing an Amazon Neptune bulk load

Use the following strategies to keep the load time to a minimum for a Neptune bulk load:

- **Clean your data:**
  - Be sure to convert your data into a supported data format before loading.
  - Remove any duplicates or known errors.
  - Reduce the number of unique predicates (such as properties of edges and vertices) as much as you can.
- **Optimize your files:**
  - If you load large files such as CSV files from an Amazon S3 bucket, the loader manages concurrency for you by parsing them into chunks that it can load in parallel. Using a very large number of tiny files can slow this process.
  - If you load multiple files from an Amazon S3 prefix, the loader automatically loads vertex files first, then edge files afterwards. However, if you know that you will only be loading edge files, `edgeOnlyLoad` can be set to `TRUE` to skip the first-pass where all files are scanned to determine their contents (vertices or edges) so that any vertex files found are loaded before any edge files. This can speed up the load time significantly, especially when many edge files are involved. In case some vertex files are also present in the same Amazon S3 prefix (`source` parameter), they will get loaded but without any ordering guarantees relative to other files. Also, if some `from` or `to` vertices are not present in the database, the edge insertion may report errors with the message `FROM_OR_TO_VERTEX_ARE_MISSING`. As a best practice, put nodes and edges in separate Amazon S3 prefix.
- **Check your loader settings:**
  - If you don't need to perform any other operations during the load, use the `OVERSUBSCRIBE` `parallelism` parameter. This parameter setting causes the bulk loader to use all available CPU resources when it runs. It generally takes 60%-70% of CPU capacity to keep the operation running as fast as I/O constraints permit.

    > **Note**
    >
    > When `parallelism` is set to `OVERSUBSCRIBE` or `HIGH` (the default setting), there is the risk when loading openCypher data that threads may encounter a race condition and deadlock, resulting in a `LOAD_DATA_DEADLOCK` error. In this case, set `parallelism` to a lower setting and retry the load.

  - If your load job will include multiple load requests, use the `queueRequest` parameter. Setting `queueRequest` to `TRUE` lets Neptune queue up your requests so you don't have to wait for one to finish before issuing another.

- If your load requests are being queued, you can set up levels of dependency using the `dependencies` parameter, so that the failure of one job causes dependent jobs to fail. This can prevent inconsistencies in the loaded data.
- If a load job is going to involve updating previously loaded values, be sure to set the `updateSingleCardinalityProperties` parameter to TRUE. If you don't, the loader will treat an attempt to update an existing single-cardinality value as an error. For Gremlin data, cardinality is also specified in property column headers (see Property Column Headers).

  > **Note**
  >
  > The `updateSingleCardinalityProperties` parameter is not available for Resource Description Framework (RDF) data.

- You can use the `failOnError` parameter to determine whether bulk load operations should fail or continue when an error is encountered. Also, you can use the `mode` parameter to be sure that a load job resumes loading from the point where a previous job failed rather than reloading data that had already been loaded.

- **Scale up** – Set the writer instance of your DB cluster to the maximum size before bulk loading. Note that if you do this, you must either scale up any read-replica instances in the DB cluster as well, or remove them until you have finished loading the data.

  When your bulk load is complete, be sure to scale the writer instance down again.

> **Important**
>
> If you experience a cycle of repeated read-replica restarts because of replication lag during a bulk load, your replicas are likely unable to keep up with the writer in your DB cluster. Either scale the readers to be larger than the writer, or temporarily remove them during the bulk load and then recreate them after it completes.

See Request Parameters for more details about setting loader request parameters.

Fast bulk load (offline loader)

The Amazon Neptune offline loader is a high-performance variant of the standard bulk loader. Use it for initial loads of large Gremlin CSV datasets. The loader reads files from Amazon Simple

Storage Service and builds graph index structures directly on the instance. When the load completes, Neptune swaps the new indexes into place atomically. Because it bypasses the per-row transactional write path, it loads large datasets significantly faster than the standard bulk loader.

When to use the offline loader

Use the offline loader when *all* of the following are true:

- You are loading Gremlin CSV data.
- The target graph is empty, or you are willing to reset the graph entirely before loading.
- The dataset is large enough that load throughput matters more than incremental write availability.
- Your source data is clean. The offline loader fails the entire load on any referential or cardinality violation.
- You can accept that Neptune blocks read and write queries for the duration of the load.

Use the standard bulk loader when any of the following apply:

- You are loading RDF, openCypher, or geospatial data.
- You need to append to an existing, non-empty graph.
- You need to resume a previously failed load (`mode=RESUME`).
- You need to queue multiple load jobs (`queueRequest=TRUE`).
- You need to overwrite single-cardinality property values (`updateSingleCardinalityProperties=TRUE`).
- Your source data might contain errors (for example, duplicate edge IDs or missing vertex references) and you want those rows skipped rather than treated as fatal.

Prerequisites and limitations

Engine version

The offline loader requires Neptune engine version 1.5.0.0 or later.

Instance requirements

The offline loader has the following instance requirements:

- Use provisioned instances only. The offline loader does not support serverless instances.
- Use a large instance type (for example, `db.r6i.12xlarge` or larger) to avoid out-of-memory failures. The offline loader uses all available vCPUs and memory on the instance.
- Use Neptune I/O-Optimized storage for better cost efficiency on large loads.

Limitations

The offline loader has the following limitations:

- **Gremlin CSV only** – The offline loader supports only Gremlin CSV format. It does not support openCypher, RDF, or geospatial formats.
- **No compressed files** – The offline loader does not support compressed source files. Provide uncompressed Gremlin CSV files.
- **Empty graph required** – The target graph must be empty. Use `fast reset` to clear the graph before loading.
- **One load at a time** – Only one offline load can run per cluster. Neptune rejects concurrent load requests.
- **No resume** – You cannot resume a failed load. Reset the graph and restart the load from the beginning.
- **No parallelism control** – The offline loader always uses all available vCPUs. It does not support the `parallelism` parameter. Neptune rejects the request if you include it.
- **Queries blocked** – Neptune blocks read and write queries while an offline load is in progress.
- **All-or-nothing semantics** – Referential integrity violations, duplicate edge IDs, and single-cardinality conflicts cause the entire load to fail. Neptune retains no partial data.

Starting an offline load

Use the same `/loader` endpoint as the standard bulk loader. Add `"loadType": "offline"` to the request body to select the offline loader.

The following example shows a request:

```
POST https://your-neptune-endpoint:port/loader
Content-Type: application/json
```

```
{
  "source": "s3://amzn-s3-demo-bucket/graph-data/",
  "format": "csv",
  "iamRoleArn": "arn:aws:iam::123456789012:role/NeptuneLoadFromS3",
  "region": "us-west-2",
  "loadType": "offline",
  "failOnError": "TRUE"
}
```

The offline loader supports the following parameters:

- `source` – The Amazon S3 URI of your data files.
- `format` – Must be `csv`.
- `iamRoleArn` – The IAM role ARN with read access to the Amazon S3 bucket.
- `region` – The AWS Region of the Amazon S3 bucket.
- `loadType` – Set to `offline` to use the offline loader.
- `failOnError` – Controls behavior on parse errors. When `TRUE`, any parse error aborts the load immediately.

The following standard loader parameters are *not* supported by the offline loader: `parallelism`, `mode`, `queueRequest`, `dependencies`, `updateSingleCardinalityProperties`, and `edgeOnlyLoad`.

The following example shows the response:

```
{
  "status": "200 OK",
  "payload": {
    "loadId": "ef478d76-d9da-4d94-8ff1-08d9d4863aa5"
  }
}
```

Monitoring an offline load

Use the same `GET /loader/loadId` endpoint as the standard bulk loader. While the load is in progress, the response includes an additional `progress` object that reports the current phase.

```
{
```

```
  "payload": {
    "overallStatus": {
      "status": "LOAD_IN_PROGRESS",
      "fullUri": "s3://amzn-s3-demo-bucket/graph-data/",
      "totalRecords": 1234567,
      "totalDuplicates": 0,
      "parsingErrors": 0,
      "insertErrors": 0,
      "progress": {
        "phase": "BUILDING_GRAPH",
        "phaseNumber": 3,
        "totalPhases": 5
      }
    }
  }
}
```

The `progress` object is present only while the load is in progress. Neptune removes it after the load reaches a terminal state.

Load phases

The offline loader processes data in five sequential phases.

Phase	Name	Description
1	`PARSING`	Neptune reads source files and collects unique strings.
2	`BUILDING_DICTIONARY`	Neptune builds the sorted string-to-identifier dictionary.
3	`BUILDING_GRAPH`	Neptune reads the data again and translates rows into graph statements using the dictionary.
4	`INDEXING`	Neptune sorts statements and builds the graph indexes. Referential and cardinality violations are detected during this phase.
5	`FINALIZING`	Neptune atomically swaps the new indexes into place and unblocks queries.

All other status fields (`status`, `fullUri`, `totalRecords`, `totalDuplicates`, `parsingErrors`, `insertErrors`, and the error log available with `errors=true`) have the same meaning as in the standard loader. See Neptune Loader Get-Status API.

How the offline loader differs from the standard loader

The following table compares key behaviors of the offline loader and the standard bulk loader.

Capability	Standard bulk loader	Offline loader
Supported data formats	Gremlin CSV, openCypher, RDF (NTriples, NQuads, RDF/XML, Turtle)	Gremlin CSV only
Target graph state	Empty or non-empty (appends)	Empty only. Reset the graph first.
Concurrent loads	Up to 64 queued jobs (`queueRequest=TRUE`)	One load at a time per cluster
Resume after failure	Yes (`mode=RESUME` or `mode=AUTO`)	Not supported. Reset the graph and restart the load.
Parallelism control	User-configurable via `parallelism`	Not configurable. Uses all available vCPUs.
Progress reporting	`status` only	`status` plus a `progress` object with five phases
Query availability during load	You can run read and write queries throughout the load	Neptune blocks queries during the load
Referential integrity	Neptune skips missing vertex references per row	The entire load fails with `FROM_OR_TO_VERTEX_ARE_MISSING`
Duplicate edge IDs	Neptune skips duplicates per row	The entire load fails with `ID_ASSIGNED_TO_MULTIPLE_EDGES`

Capability	Standard bulk loader	Offline loader
Single-cardinality conflicts	Neptune skips conflicts per row	The entire load fails with `SINGLE_CARDINALITY_VIOLATION`
Post-load statistics	Available immediately after load completes	Neptune triggers statistic s collection after the load completes. Statistics might take a few minutes to become available.

Error handling

The offline loader uses all-or-nothing semantics. If any data integrity violation is detected during the `INDEXING` phase, the entire load fails and no data is retained. The graph returns to its pre-load state (empty).

The following errors cause the entire load to fail, regardless of the `failOnError` setting.

Error code	Phase	Description
`FROM_OR_TO_VERTEX_ARE_MISSING`	`INDEXING`	An edge references a vertex ID that does not exist in the input data.
`ID_ASSIGNED_TO_MULTIPLE_EDGES`	`INDEXING`	Two or more edges share the same edge ID.
`SINGLE_CARDINALITY_VIOLATION`	`INDEXING`	A property declared with single cardinality has more than one value on the same vertex.
`PARSING_ERROR`	`PARSING`	• A string property value starts with an internal namespace prefix, which the offline loader does not currently support. • A data type conversion failure, such as a non-numeric value in an `Int` column.

Other parse errors (for example, a malformed CSV row or a value exceeding the 55 MB size limit) are handled according to the `failOnError` parameter:

- `failOnError=TRUE` – Any parse error aborts the load immediately.
- `failOnError=FALSE` – Neptune logs parse errors and skips the affected row. The load continues with remaining data.

Use `GET /loader/loadId?errors=true` to retrieve detailed error information after a failed load.

Best practices

- **Validate data before loading** – Validate your data before starting a load. The offline loader fails the entire load if it detects an integrity violation. Ensure all edge `~from` and `~to` values reference vertex IDs that exist in the input, edge IDs are unique, and single-cardinality properties have only one value per vertex.
- **Use a large instance** – The offline loader uses all available memory and vCPUs. A larger instance (for example, `db.r6i.12xlarge` or larger) loads data faster and avoids out-of-memory failures on large datasets.
- **Use fast reset before retrying** – If a load fails, use `fast reset` to clear the graph before retrying. The offline loader requires an empty graph.
- **Monitor phase progression** – Poll `GET /loader/loadId` to track which of the five phases the load is in. The `INDEXING` phase is where data integrity violations are detected, so a failure in this phase indicates a data quality issue.
- **Separate vertex and edge files** – Organize your Amazon S3 data with vertex files and edge files under separate prefixes. This makes it easier to validate referential integrity before loading.

> **Note**
>
> For boolean property columns, the offline loader treats any value other than `true` as `false`. For more information, see Gremlin load data formats.

Neptune Loader Reference

This section describes the `Loader` APIs for Amazon Neptune that are available from the HTTP endpoint of a Neptune DB instance.

> **Note**
>
> See Neptune Loader Error and Feed Messages for a list of the error and feed messages returned by the loader in case of errors.

Contents

- Neptune Loader Command
  - Neptune Loader Request Syntax
  - Neptune Loader Request Parameters
    - Special considerations for loading openCypher data
  - Neptune Loader Response Syntax
  - Neptune Loader Errors
  - Neptune Loader Examples
- Neptune Loader Get-Status API
  - Neptune Loader Get-Status requests
    - Loader Get-Status request syntax
    - Neptune Loader Get-Status request parameters
  - Neptune Loader Get-Status Responses
    - Neptune Loader Get-Status Response JSON layout
    - Neptune Loader Get-Status overallStatus and failedFeeds response objects
    - Neptune Loader Get-Status errors response object
    - Neptune Loader Get-Status errorLogs response object
  - Neptune Loader Get-Status Examples
    - Example request for load status
    - Example request for loadIds
    - Example request for detailed status
  - Neptune Loader Get-Status errorLogs examples

    - Example detailed status response when errors occurred
    - Example of a Data prefetch task interrupted error
- Neptune Loader Cancel Job
    - Cancel Job request syntax
    - Cancel Job Request Parameters
    - Cancel Job Response Syntax
    - Cancel Job Errors
    - Cancel Job Error Messages
    - Cancel Job Examples

Neptune Loader Command

Loads data from an Amazon S3 bucket into a Neptune DB instance.

To load data, you must send an HTTP POST request to the `https://`*`your-neptune-endpoint`*`:`*`port`*`/loader` endpoint. The parameters for the `loader` request can be sent in the POST body or as URL-encoded parameters.

> **Important**
>
> The MIME type must be `application/json`.

The Amazon S3 bucket must be in the same AWS Region as the cluster.

> **Note**
>
> You can load encrypted data from Amazon S3 if it was encrypted using the Amazon S3 `SSE-S3` mode. In that case, Neptune is able to impersonate your credentials and issue `s3:getObject` calls on your behalf.
> You can also load encrypted data from Amazon S3 that was encrypted using the `SSE-KMS` mode, as long as your IAM role includes the necessary permissions to access AWS KMS. Without proper AWS KMS permissions, the bulk load operation fails and returns a `LOAD_FAILED` response.

Neptune does not currently support loading Amazon S3 data encrypted using the `SSE-C` mode.

You don't have to wait for one load job to finish before you start another one. Neptune can queue up as many as 64 jobs requests at a time, provided that their `queueRequest` parameters are all set to `"TRUE"`. The queue order of the jobs will be first-in-first-out (FIFO). If you don't want a load job to be queued up, on the other hand, you can set its `queueRequest` parameter to `"FALSE"` (the default), so that the load job will fail if another one is already in progress.

You can use the `dependencies` parameter to queue up a job that must only be run after specified previous jobs in the queue have completed successfully. If you do that and any of those specified jobs fails, your job will not be run and its status will be set to `LOAD_FAILED_BECAUSE_DEPENDENCY_NOT_SATISFIED`.

Neptune Loader Request Syntax

```
{
  "source" : "string",
  "format" : "string",
  "iamRoleArn" : "string",
  "mode": "NEW|RESUME|AUTO",
  "region" : "us-east-1",
  "failOnError" : "string",
  "parallelism" : "string",
  "parserConfiguration" : {
    "baseUri" : "http://base-uri-string",
    "namedGraphUri" : "http://named-graph-string"
  },
  "updateSingleCardinalityProperties" : "string",
  "queueRequest" : "TRUE",
  "dependencies" : ["load_A_id", "load_B_id"]
}
```

edgeOnlyLoad Syntax

For an `edgeOnlyLoad`, the syntax would be:

```
{
"source" : "string",
"format" : "string",
```

```
"iamRoleArn" : "string",
"mode": "NEW|RESUME|AUTO",
"region" : "us-east-1",
"failOnError" : "string",
"parallelism" : "string",
"edgeOnlyLoad" : "string",
"parserConfiguration" : {
    "baseUri" : "http://base-uri-string",
    "namedGraphUri" : "http://named-graph-string"
},
"updateSingleCardinalityProperties" : "string",
"queueRequest" : "TRUE",
"dependencies" : ["load_A_id", "load_B_id"]
}
```

Neptune Loader Request Parameters

- **`source`** – An Amazon S3 URI.

  The SOURCE parameter accepts an Amazon S3 URI that identifies a single file, multiple files, a folder, or multiple folders. Neptune loads every data file in any folder that is specified.

  The URI can be in any of the following formats.

  - `s3://bucket_name/object-key-name`
  - `https://s3.amazonaws.com/bucket_name/object-key-name`
  - `https://s3.us-east-1.amazonaws.com/bucket_name/object-key-name`

  The `object-key-name` element of the URI is equivalent to the prefix parameter in an Amazon S3 ListObjects API call. It identifies all the objects in the specified Amazon S3 bucket whose names begin with that prefix. That can be a single file or folder, or multiple files and/or folders.

  The specified folder or folders can contain multiple vertex files and multiple edge files.

  For example, if you had the following folder structure and files in an Amazon S3 bucket named `bucket-name`:

  ```
  s3://bucket-name/a/bc
  s3://bucket-name/ab/c
  s3://bucket-name/ade
  s3://bucket-name/bcd
  ```

If the source parameter is specified as `s3://bucket-name/a`, the first three files will be loaded.

```
s3://bucket-name/a/bc
s3://bucket-name/ab/c
s3://bucket-name/ade
```

- **`format`** – The format of the data. For more information about data formats for the Neptune `Loader` command, see Using the Amazon Neptune bulk loader to ingest data.

  Allowed values

  - **`csv`** for the Gremlin CSV data format.
  - **`opencypher`** for the openCypher CSV data format.
  - **`ntriples`** for the N-Triples RDF data format.
  - **`nquads`** for the N-Quads RDF data format.
  - **`rdfxml`** for the RDF\XML RDF data format.
  - **`turtle`** for the Turtle RDF data format.

- **`iamRoleArn`** – The Amazon Resource Name (ARN) for an IAM role to be assumed by the Neptune DB instance for access to the S3 bucket. For information about creating a role that has access to Amazon S3 and then associating it with a Neptune cluster, see Prerequisites: IAM Role and Amazon S3 Access.

  Starting with engine release 1.2.1.0.R3, you can also chain multiple IAM roles if the Neptune DB instance and the Amazon S3 bucket are located in different AWS Accounts. In this case, `iamRoleArn` contains a comma-separated list of role ARNs, as described in Chaining IAM roles in Amazon Neptune. For example:

  AWS CLI

  ```
  aws neptunedata start-loader-job \
    --endpoint-url https://your-neptune-endpoint:port \
    --source "s3://(the target bucket name)/(the target date file name)" \
    --format "csv" \
    --iam-role-arn "arn:aws:iam::(Account A ID):role/(RoleA),arn:aws:iam::(Account B
   ID):role/(RoleB),arn:aws:iam::(Account C ID):role/(RoleC)" \
    --s3-bucket-region "us-east-1"
  ```

  For more information, see start-loader-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_loader_job(
    source='s3://(the target bucket name)/(the target date file name)',
    format='csv',
    iamRoleArn='arn:aws:iam::(Account A ID):role/(RoleA),arn:aws:iam::(Account B
 ID):role/(RoleB),arn:aws:iam::(Account C ID):role/(RoleC)',
    s3BucketRegion='us-east-1'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/loader \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "source" : "s3://(the target bucket name)/(the target date file name)",
        "iamRoleArn" : "arn:aws:iam::(Account A
 ID):role/(RoleA),arn:aws:iam::(Account B ID):role/(RoleB),arn:aws:iam::(Account C
 ID):role/(RoleC)",
        "format" : "csv",
        "region" : "us-east-1"
      }'
```

Note

This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -X POST https://your-neptune-endpoint:port/loader \
  -H 'Content-Type: application/json' \
  -d '{
        "source" : "s3:////(the target bucket name)/(the target date file name)",
        "iamRoleArn" : "arn:aws:iam::(Account A
 ID):role/(RoleA),arn:aws:iam::(Account B ID):role/(RoleB),arn:aws:iam::(Account C
 ID):role/(RoleC)",
        "format" : "csv",
        "region" : "us-east-1"
      }'
```

- **`region`** – The `region` parameter must match the AWS Region of the cluster and the S3 bucket.

  Amazon Neptune is available in the following Regions:

  - US East (N. Virginia): `us-east-1`
  - US East (Ohio): `us-east-2`
  - US West (N. California): `us-west-1`
  - US West (Oregon): `us-west-2`
  - Canada (Central): `ca-central-1`
  - Canada West (Calgary): `ca-west-1`
  - South America (São Paulo): `sa-east-1`
  - Europe (Stockholm): `eu-north-1`
  - Europe (Spain): `eu-south-2`
  - Europe (Ireland): `eu-west-1`
  - Europe (London): `eu-west-2`
  - Europe (Paris): `eu-west-3`
  - Europe (Frankfurt): `eu-central-1`
  - Middle East (Bahrain): `me-south-1`
  - Middle East (UAE): `me-central-1`
  - Israel (Tel Aviv): `il-central-1`
  - Africa (Cape Town): `af-south-1`

- Asia Pacific (Hong Kong): `ap-east-1`
- Asia Pacific (Tokyo): `ap-northeast-1`
- Asia Pacific (Seoul): `ap-northeast-2`
- Asia Pacific (Osaka): `ap-northeast-3`
- Asia Pacific (Singapore): `ap-southeast-1`
- Asia Pacific (Sydney): `ap-southeast-2`
- Asia Pacific (Jakarta): `ap-southeast-3`
- Asia Pacific (Melbourne): `ap-southeast-4`
- Asia Pacific (Malaysia): `ap-southeast-5`
- Asia Pacific (Mumbai): `ap-south-1`
- Asia Pacific (Hyderabad): `ap-south-2`
- China (Beijing): `cn-north-1`
- China (Ningxia): `cn-northwest-1`
- AWS GovCloud (US-West): `us-gov-west-1`
- AWS GovCloud (US-East): `us-gov-east-1`

- **`mode`** – The load job mode.

  Allowed values: `RESUME`, `NEW`, `AUTO`.

  Default value: `AUTO`

  - `RESUME` – In RESUME mode, the loader looks for a previous load from this source, and if it finds one, resumes that load job. If no previous load job is found, the loader stops.

    The loader avoids reloading files that were successfully loaded in a previous job. It only tries to process failed files. If you dropped previously loaded data from your Neptune cluster, that data is not reloaded in this mode. If a previous load job loaded all files from the same source successfully, nothing is reloaded, and the loader returns success.
  - `NEW` – In NEW mode, the creates a new load request regardless of any previous loads. You can use this mode to reload all the data from a source after dropping previously loaded data from your Neptune cluster, or to load new data available at the same source.
  - `AUTO` – In AUTO mode, the loader looks for a previous load job from the same source, and if it finds one, resumes that job, just as in `RESUME` mode.

If the loader doesn't find a previous load job from the same source, it loads all data from the source, just as in `NEW` mode.

- **`edgeOnlyLoad`** – A flag that controls file processing order during bulk loading.

  Allowed values: `"TRUE"`, `"FALSE"`.

  Default value: `"FALSE"`.

  When this parameter is set to "FALSE", the loader automatically loads vertex files first, then edge files afterwards. It does this by first scanning all files to determine their contents (vertices or edges). When this parameter is set to "TRUE", the loader skips the initial scanning phase and immediately loads all files in the order they appear. For more information see bulk load optimize.

- **`failOnError`** – A flag to toggle a complete stop on an error.

  Allowed values: `"TRUE"`, `"FALSE"`.

  Default value: `"TRUE"`.

  When this parameter is set to `"FALSE"`, the loader tries to load all the data in the location specified, skipping any entries with errors.

  When this parameter is set to `"TRUE"`, the loader stops as soon as it encounters an error. Data loaded up to that point persists.

- **`parallelism`** – This is an optional parameter that can be set to reduce the number of threads used by the bulk load process.

  Allowed values:

  - `LOW` – The number of threads used is the number of available vCPUs divided by 8.
  - `MEDIUM` – The number of threads used is the number of available vCPUs divided by 2.
  - `HIGH` – The number of threads used is the same as the number of available vCPUs.
  - `OVERSUBSCRIBE` – The number of threads used is the number of available vCPUs multiplied by 2. If this value is used, the bulk loader takes up all available resources.

    This does not mean, however, that the `OVERSUBSCRIBE` setting results in 100% CPU utilization. Because the load operation is I/O bound, the highest CPU utilization to expect is in the 60% to 70% range.

  Default value: `HIGH`

The `parallelism` setting can sometimes result in a deadlock between threads when loading openCypher data. When this happens, Neptune returns the `LOAD_DATA_DEADLOCK` error. You can generally fix the issue by setting `parallelism` to a lower setting and retrying the load command.

- **`parserConfiguration`** – An optional object with additional parser configuration values. Each of the child parameters is also optional:

Name	Example Value	Description
`namedGraphUri`	*`http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph`*	The default graph for all RDF formats when no graph is specified (for non-quads formats and NQUAD entries with no graph). The default is `http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph`
`baseUri`	*`http://aws.amazon.com/neptune/default`*	The base URI for RDF/XML and Turtle formats. The default is `http://aws.amazon.com/neptune/default` .
`allowEmptyStrings`	*`true`*	Gremlin users need to be able to pass empty string values("") as node and edge properties when loading CSV data. If `allowEmptyStrings` is set to `false` (the default), such empty strings are treated as nulls and are not loaded.

If `allowEmptyStrings` is set to `true`, the loader treats empty strings as valid property values and loads them accordingly.

For more information, see SPARQL Default Graph and Named Graphs.

- **`updateSingleCardinalityProperties`** – This is an optional parameter that controls how the bulk loader treats a new value for single-cardinality vertex or edge properties.

  Allowed values: `"TRUE"`, `"FALSE"`.

  Default value: `"FALSE"`.

  By default, or when `updateSingleCardinalityProperties` is explicitly set to `"FALSE"`, the loader treats a new value as an error, because it violates single cardinality.

  When `updateSingleCardinalityProperties` is set to `"TRUE"`, on the other hand, the bulk loader replaces the existing value with the new one. If multiple edge or single-cardinality vertex property values are provided in the source file(s) being loaded, the final value at the end of the bulk load could be any one of those new values. The loader only guarantees that the existing value has been replaced by one of the new ones.

- **`queueRequest`** – This is an optional flag parameter that indicates whether the load request can be queued up or not.

  You don't have to wait for one load job to complete before issuing the next one, because Neptune can queue up as many as 64 jobs at a time, provided that their `queueRequest` parameters are all set to `"TRUE"`. The queue order of the jobs will be first-in-first-out (FIFO).

  If the `queueRequest` parameter is omitted or set to `"FALSE"`, the load request will fail if another load job is already running.

  Allowed values: `"TRUE"`, `"FALSE"`.

  Default value: `"FALSE"`.

- **`dependencies`** – This is an optional parameter that can make a queued load request contingent on the successful completion of one or more previous jobs in the queue.

Neptune can queue up as many as 64 load requests at a time, if their `queueRequest` parameters are set to `"TRUE"`. The `dependencies` parameter lets you make execution of such a queued request dependent on the successful completion of one or more specified previous requests in the queue.

For example, if load `Job-A` and `Job-B` are independent of each other, but load `Job-C` needs `Job-A` and `Job-B` to be finished before it begins, proceed as follows:

1. Submit `load-job-A` and `load-job-B` one after another in any order, and save their load-ids.
2. Submit `load-job-C` with the load-ids of the two jobs in its `dependencies` field:

```
"dependencies" : ["job_A_load_id", "job_B_load_id"]
```

Because of the `dependencies` parameter, the bulk loader will not start `Job-C` until `Job-A` and `Job-B` have completed successfully. If either one of them fails, Job-C will not be executed, and its status will be set to `LOAD_FAILED_BECAUSE_DEPENDENCY_NOT_SATISFIED`.

You can set up multiple levels of dependency in this way, so that the failure of one job will cause all requests that are directly or indirectly dependent on it to be cancelled.

- **`userProvidedEdgeIds`** – This parameter is required only when loading openCypher data that contains relationship IDs. It must be included and set to `True` when openCypher relationship IDs are explicitly provided in the load data (recommended).

  When `userProvidedEdgeIds` is absent or set to `True`, an `:ID` column must be present in every relationship file in the load.

  When `userProvidedEdgeIds` is present and set to `False`, relationship files in the load **must not** contain an `:ID` column. Instead, the Neptune loader automatically generates an ID for each relationship.

  It's useful to provide relationship IDs explicitly so that the loader can resume loading after error in the CSV data have been fixed, without having to reload any relationships that have already been loaded. If relationship IDs have not been explicitly assigned, the loader cannot resume a failed load if any relationship file has had to be corrected, and must instead reload all the relationships.
- `accessKey` – **[deprecated]** An access key ID of an IAM role with access to the S3 bucket and data files.

The `iamRoleArn` parameter is recommended instead. For information about creating a role that has access to Amazon S3 and then associating it with a Neptune cluster, see Prerequisites: IAM Role and Amazon S3 Access.

For more information, see Access keys (access key ID and secret access key).

- `secretKey` – **[deprecated]** The `iamRoleArn` parameter is recommended instead. For information about creating a role that has access to Amazon S3 and then associating it with a Neptune cluster, see Prerequisites: IAM Role and Amazon S3 Access.

  For more information, see Access keys (access key ID and secret access key).

Special considerations for loading openCypher data

- When loading openCypher data in CSV format, the format parameter must be set to `opencypher`.
- The `updateSingleCardinalityProperties` parameter is supported for openCypher loads. All openCypher properties have single cardinality, and if an ID value appears more than once, it is treated as a duplicate or an insertion error (see below).
- The Neptune loader handles duplicates that it encounters in openCypher data as follows:
  - If the loader encounters multiple rows with the same node ID, they are merged using the following rule:
    - All the labels in the rows are added to the node.
    - For each property, only one of the property values is loaded. The selection of the one to load is non-deterministic.
  - If the loader encounters multiple rows with the same relationship ID, only one of them is loaded. The selection of the one to load is non-deterministic.
  - By default, the loader does not update property values of an existing node or relationship in the database if it encounters load data having the ID of the existing node or relationship. However, it does load node labels and properties that are not present in the existing node or relationship. To update existing property values, set `updateSingleCardinalityProperties` to `"TRUE"`.
- Although you don't have to assign IDs to relationships, it is usually a good idea (see the `userProvidedEdgeIds` parameter above). Without explicit relationship IDs, the loader must reload all relationships in case of an error in a relationship file, rather than resuming the load from where it failed.

Also, if the load data doesn't contain explicit relationship IDs, the loader has no way of detecting duplicate relationships.

Here is an example of an openCypher load command:

AWS CLI

```
aws neptunedata start-loader-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --source "s3://bucket-name/object-key-name" \
  --format "opencypher" \
  --user-provided-edge-ids \
  --iam-role-arn "arn:aws:iam::account-id:role/role-name" \
  --s3-bucket-region "region" \
  --no-fail-on-error \
  --parallelism "MEDIUM"
```

For more information, see start-loader-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_loader_job(
    source='s3://bucket-name/object-key-name',
    format='opencypher',
    userProvidedEdgeIds=True,
    iamRoleArn='arn:aws:iam::account-id:role/role-name',
    s3BucketRegion='region',
    failOnError=False,
    parallelism='MEDIUM'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/loader \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "source" : "s3://bucket-name/object-key-name",
        "format" : "opencypher",
        "userProvidedEdgeIds": "TRUE",
        "iamRoleArn" : "arn:aws:iam::account-id:role/role-name",
        "region" : "region",
        "failOnError" : "FALSE",
        "parallelism" : "MEDIUM"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -X POST https://your-neptune-endpoint:port/loader \
  -H 'Content-Type: application/json' \
  -d '{
        "source" : "s3://bucket-name/object-key-name",
        "format" : "opencypher",
        "userProvidedEdgeIds": "TRUE",
        "iamRoleArn" : "arn:aws:iam::account-id:role/role-name",
        "region" : "region",
        "failOnError" : "FALSE",
        "parallelism" : "MEDIUM"
      }'
```

The loader response is the same as normal. For example:

```
{
  "status" : "200 OK",
```

```
  "payload" : {
    "loadId" : "guid_as_string"
  }
}
```

Neptune Loader Response Syntax

```
{
    "status" : "200 OK",
    "payload" : {
        "loadId" : "guid_as_string"
    }
}
```

200 OK

Successfully started load job returns a `200` code.

Neptune Loader Errors

When an error occurs, a JSON object is returned in the `BODY` of the response. The `message` object contains a description of the error.

Error Categories

- `Error 400` – Syntax errors return an HTTP `400` bad request error. The message describes the error.
- `Error 500` – A valid request that cannot be processed returns an HTTP `500` internal server error. The message describes the error.

The following are possible error messages from the loader with a description of the error.

Loader Error Messages

- `Couldn't find the AWS credential for iam_role_arn` (HTTP 400)

  The credentials were not found. Verify the supplied credentials against the IAM console or AWS CLI output. Make sure that you have added the IAM role specified in `iamRoleArn` to the cluster.
- `S3 bucket not found for source` (HTTP 400)

  The S3 bucket does not exist. Check the name of the bucket.

- `The source source-uri does not exist/not reachable` (HTTP 400)

  No matching files were found in the S3 bucket.
- `Unable to connect to S3 endpoint. Provided source = source-uri and region = aws-region` (HTTP 500)

  Unable to connect to Amazon S3. Region must match the cluster Region. Ensure that you have a VPC endpoint. For information about creating a VPC endpoint, see Creating an Amazon S3 VPC Endpoint.
- `Bucket is not in provided Region (aws-region)` (HTTP 400)

  The bucket must be in the same AWS Region as your Neptune DB instance.
- `Unable to perform S3 list operation` (HTTP 400)

  The IAM user or role provided does not have `List` permissions on the bucket or the folder. Check the policy or the access control list (ACL) on the bucket.
- `Start new load operation not permitted on a read replica instance` (HTTP 405)

  Loading is a write operation. Retry load on the read/write cluster endpoint.
- `Failed to start load because of unknown error from S3` (HTTP 500)

  Amazon S3 returned an unknown error. Contact AWS Support.
- `Invalid S3 access key` (HTTP 400)

  Access key is invalid. Check the provided credentials.
- `Invalid S3 secret key` (HTTP 400)

  Secret key is invalid. Check the provided credentials.
- `Max concurrent load limit breached` (HTTP 400)

  If a load request is submitted without `"queueRequest" : "TRUE"`, and a load job is currently running, the request will fail with this error.
- `Failed to start new load for the source "source name". Max load task queue size limit breached. Limit is 64` (HTTP 400)

  Neptune supports queuing up as many as 64 loader jobs at a time. If an additional load request is submitted to the queue when it already contains 64 jobs, the request fails with this message.

Neptune Loader Examples

This example demonstrates how to use the Neptune loader to load data into a Neptune graph database using the Gremlin CSV format. The request is sent as an HTTP POST request to the Neptune loader endpoint, and the request body contains the necessary parameters to specify the data source, format, IAM role, and other configuration options. The response includes the load ID, which can be used to track the progress of the data loading process.

Example Request

The following is a request sent via HTTP POST using the `curl` command. It loads a file in the Neptune CSV format. For more information, see Gremlin load data format.

AWS CLI

```
aws neptunedata start-loader-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --source "s3://bucket-name/object-key-name" \
  --format "csv" \
  --iam-role-arn "ARN for the IAM role you are using" \
  --s3-bucket-region "region" \
  --no-fail-on-error \
  --parallelism "MEDIUM" \
  --no-update-single-cardinality-properties \
  --no-queue-request
```

For more information, see start-loader-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_loader_job(
    source='s3://bucket-name/object-key-name',
    format='csv',
    iamRoleArn='ARN for the IAM role you are using',
```

```
    s3BucketRegion='region',
    failOnError=False,
    parallelism='MEDIUM',
    updateSingleCardinalityProperties=False,
    queueRequest=False
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/loader \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "source" : "s3://bucket-name/object-key-name",
        "format" : "csv",
        "iamRoleArn" : "ARN for the IAM role you are using",
        "region" : "region",
        "failOnError" : "FALSE",
        "parallelism" : "MEDIUM",
        "updateSingleCardinalityProperties" : "FALSE",
        "queueRequest" : "FALSE"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -X POST https://your-neptune-endpoint:port/loader \
  -H 'Content-Type: application/json' \
  -d '{
        "source" : "s3://bucket-name/object-key-name",
        "format" : "csv",
        "iamRoleArn" : "ARN for the IAM role you are using",
        "region" : "region",
```

```
        "failOnError" : "FALSE",
        "parallelism" : "MEDIUM",
        "updateSingleCardinalityProperties" : "FALSE",
        "queueRequest" : "FALSE"
      }'
```

Example Response

```
{
    "status" : "200 OK",
    "payload" : {
        "loadId" : "ef478d76-d9da-4d94-8ff1-08d9d4863aa5"
    }
}
```

Neptune Loader Get-Status API

Gets the status of a `loader` job.

To get load status, you must send an HTTP `GET` request to the `https://`*`your-neptune-endpoint`*`:`*`port`*`/loader` endpoint. To get the status for a particular load request, you must include the `loadId` as a URL parameter, or append the `loadId` to the URL path.

Neptune only keeps track of the most recent 1,024 bulk load jobs, and only stores the last 10,000 error details per job.

See Neptune Loader Error and Feed Messages for a list of the error and feed messages returned by the loader in case of errors.

Contents

- Neptune Loader Get-Status requests
  - Loader Get-Status request syntax
  - Neptune Loader Get-Status request parameters
- Neptune Loader Get-Status Responses
  - Neptune Loader Get-Status Response JSON layout
  - Neptune Loader Get-Status overallStatus and failedFeeds response objects
  - Neptune Loader Get-Status errors response object
  - Neptune Loader Get-Status errorLogs response object

- Neptune Loader Get-Status Examples
  - Example request for load status
  - Example request for loadIds
  - Example request for detailed status
- Neptune Loader Get-Status errorLogs examples
  - Example detailed status response when errors occurred
  - Example of a Data prefetch task interrupted error

Neptune Loader Get-Status requests

Loader Get-Status request syntax

```
GET https://your-neptune-endpoint:port/loader?loadId=loadId
```

```
GET https://your-neptune-endpoint:port/loader/loadId
```

```
GET https://your-neptune-endpoint:port/loader
```

Neptune Loader Get-Status request parameters

- **`loadId`** – The ID of the load job. If you do not specify a `loadId`, a list of load IDs is returned.
- **`details`** – Include details beyond overall status.

  Allowed values: `TRUE`, `FALSE`.

  Default value: `FALSE`.
- **`errors`** – Include the list of errors.

  Allowed values: `TRUE`, `FALSE`.

  Default value: `FALSE`.

  The list of errors is paged. The `page` and `errorsPerPage` parameters allow you to page through all the errors.
- **`page`** – The error page number. Only valid with the `errors` parameter set to `TRUE`.

  Allowed values: Positive integers.

Default value: 1.

- **`errorsPerPage`** – The number of errors per each page. Only valid with the `errors` parameter set to `TRUE`.

  Allowed values: Positive integers.

  Default value: 10.
- **`limit`** – The number of load ids to list. Only valid when requesting a list of load IDs by sending a `GET` request with no `loadId` specified.

  Allowed values: Positive integers from 1 through 100.

  Default value: 100.
- **`includeQueuedLoads`** – An optional parameter that can be used to exclude the load IDs of queued load requests when a list of load IDs is requested.

  By default, the load IDs of all load jobs with status `LOAD_IN_QUEUE` are included in such a list. They appear before the load IDs of other jobs, sorted by the time they were added to the queue from most recent to earliest.

  Allowed values: `TRUE`, `FALSE`.

  Default value: `TRUE`.

Neptune Loader Get-Status Responses

The following example response from the Neptune Get-Status API describes the overall structure of the response, explains the various fields and their data types, as well as the error handling and error log details.

Neptune Loader Get-Status Response JSON layout

The general layout of a loader status response is as follows:

```
{
    "status" : "200 OK",
    "payload" : {
        "feedCount" : [
            {
                "LOAD_FAILED" : number
```

```
            }
        ],
        "overallStatus" : {
            "fullUri" : "s3://bucket/key",
            "runNumber" : number,
            "retryNumber" : number,
            "status" : "string",
            "totalTimeSpent" : number,
            "startTime" : number,
            "totalRecords" : number,
            "totalDuplicates" : number,
            "parsingErrors" : number,
            "datatypeMismatchErrors" : number,
            "insertErrors" : number,
        },
        "failedFeeds" : [
            {
                "fullUri" : "s3://bucket/key",
                "runNumber" : number,
                "retryNumber" : number,
                "status" : "string",
                "totalTimeSpent" : number,
                "startTime" : number,
                "totalRecords" : number,
                "totalDuplicates" : number,
                "parsingErrors" : number,
                "datatypeMismatchErrors" : number,
                "insertErrors" : number,
            }
        ],
        "errors" : {
            "startIndex" : number,
            "endIndex" : number,
            "loadId" : "string,
            "errorLogs" : [ ]
        }
    }
}
```

Neptune Loader Get-Status `overallStatus` and `failedFeeds` response objects

The possible responses returned for each failed feed, including the error descriptions, are the same as for the `overallStatus` object in a `Get-Status` response.

The following fields appear in the `overallStatus` object for all loads, and the `failedFeeds` object for each failed feed:

- **`fullUri`** – The URI of the file or files to be loaded.

  Type: string

  Format: `s3://bucket/key`.
- **`runNumber`** – The run number of this load or feed. This is incremented when the load is restarted.

  Type: unsigned long.
- **`retryNumber`** – The retry number of this load or feed. This is incremented when the loader automatically retries a feed or load.

  Type: unsigned long.
- **`status`** – The returned status of the load or feed. `LOAD_COMPLETED` indicates a successful load with no problems. For a list of other load-status messages, see Neptune Loader Error and Feed Messages.

  Type: string.
- **`totalTimeSpent`** – The time, in seconds, spent to parse and insert data for the load or feed. This does not include the time spent fetching the list of source files.

  Type: unsigned long.
- **`totalRecords`** – Total records loaded or attempted to load.

  Type: unsigned long.

  Note that when loading from a CSV file, the record count does not refer to the number of lines loaded, but rather to the number of individual records in those lines. For example, take a tiny CSV file like this:

  ```
  ~id,~label,name,team
  'P-1','Player','Stokes','England'
  ```

  Neptune would consider this file to contain 3 records, namely:

  ```
  P-1  label Player
  ```

```
P-1  name  Stokes
P-1  team  England
```

- **`totalDuplicates`** – The number of duplicate records encountered.

  Type: unsigned long.

  As in the case of the `totalRecords` count, this value contains the number of individual duplicate records in a CSV file, not the number of duplicate lines. Take this small CSV file, for example:

```
~id,~label,name,team
P-2,Player,Kohli,India
P-2,Player,Kohli,India
```

  The status returned after loading it would look like this, reporting 6 total records, of which 3 are duplicates:

```
{
  "status": "200 OK",
  "payload": {
    "feedCount": [
      {
        "LOAD_COMPLETED": 1
      }
    ],
    "overallStatus": {
      "fullUri": "(the URI of the CSV file)",
      "runNumber": 1,
      "retryNumber": 0,
      "status": "LOAD_COMPLETED",
      "totalTimeSpent": 3,
      "startTime": 1662131463,
      "totalRecords": 6,
      "totalDuplicates": 3,
      "parsingErrors": 0,
      "datatypeMismatchErrors": 0,
      "insertErrors": 0
    }
  }
}
```

For openCypher loads, a duplicate is counted when:

- The loader detects that a row in a node file has an ID without an ID space that is the same as another ID value without an ID space, either in another row or belonging to an existing node.
- The loader detects that a row in a node file has an ID with an ID space that is the same as another ID value with ID space, either in another row or belonging to an existing node.

See Special considerations for loading openCypher data.

- **`parsingErrors`** – The number of parsing errors encountered.

  Type: unsigned long.
- **`datatypeMismatchErrors`** – The number of records with a data type that did not match the given data.

  Type: unsigned long.
- **`insertErrors`** – The number of records that could not be inserted due to errors.

  Type: unsigned long.

Neptune Loader Get-Status `errors` response object

Errors fall into the following categories:

- **`Error 400`** – An invalid `loadId` returns an HTTP `400` bad request error. The message describes the error.
- **`Error 500`** – A valid request that cannot be processed returns an HTTP `500` internal server error. The message describes the error.

See Neptune Loader Error and Feed Messages for a list of the error and feed messages returned by the loader in case of errors.

When an error occurs, a JSON `errors` object is returned in the `BODY` of the response, with the following fields:

- **`startIndex`** – The index of the first included error.

  Type: unsigned long.
- **`endIndex`** – The index of the last included error.

Type: unsigned long.

- **`loadId`** – The ID of the load. You can use this ID to print the errors for the load by setting the `errors` parameter to `TRUE`.

  Type: string.

- **`errorLogs`** – A list of the errors.

  Type: list.

Neptune Loader Get-Status `errorLogs` response object

The `errorLogs` object under `errors` in the loader Get-Status response contains an object describing each error using the following fields:

- **`errorCode`** – Identifies the nature of error.

  It can take one of the following values:

  - `PARSING_ERROR`
  - `S3_ACCESS_DENIED_ERROR`
  - `FROM_OR_TO_VERTEX_ARE_MISSING`
  - `ID_ASSIGNED_TO_MULTIPLE_EDGES`
  - `SINGLE_CARDINALITY_VIOLATION`
  - `FILE_MODIFICATION_OR_DELETION_ERROR`
  - `OUT_OF_MEMORY_ERROR`
  - `INTERNAL_ERROR` (returned when the bulk loader cannot determine the type of the error).

- **`errorMessage`** – A message describing the error.

  This can be a generic message associated with the error code or a specific message containing details, for example about a missing from/to vertex or about a parsing error.

- **`fileName`** – The name of the feed.
- **`recordNum`** – In the case of a parsing error, this is the record number in the file of the record that could not be parsed. It is set to zero if the record number is not applicable to the error, or if it could not be determined.

For example, the bulk loader would generate a parsing error if it encountered a faulty row such as the following in an RDF nquads file:

```
<http://base#subject> |http://base#predicate> <http://base#true> .
```

As you can see, the second `http` in the row above should be preceded by `<` rather than `|`. The resulting error object under `errorLogs` in a status response would look like this:

```
{
    "errorCode" : "PARSING_ERROR",
    "errorMessage" : "Expected '<', found: |",
    "fileName" : "s3://bucket/key",
    "recordNum" : 12345
},
```

Neptune Loader Get-Status Examples

The following examples showcase the usage of the Neptune loader's GET-Status API, which allows you to retrieve information about the status of your data loads into the Amazon Neptune graph database. These examples cover three main scenarios: retrieving the status of a specific load, listing the available load IDs, and requesting detailed status information for a specific load.

Example request for load status

The following is a request sent via HTTP `GET` using the `curl` command.

AWS CLI

```
aws neptunedata get-loader-job-status \
  --endpoint-url https://your-neptune-endpoint:port \
  --load-id loadId (a UUID)
```

For more information, see get-loader-job-status in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
```

```
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_loader_job_status(
    loadId='loadId (a UUID)'
)

print(response)
```

awscurl

```
awscurl 'https://your-neptune-endpoint:port/loader/loadId (a UUID)' \
  --region us-east-1 \
  --service neptune-db
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -X GET 'https://your-neptune-endpoint:port/loader/loadId (a UUID)'
```

Example Response

```
{
    "status" : "200 OK",
    "payload" : {
        "feedCount" : [
            {
                "LOAD_FAILED" : 1
            }
        ],
        "overallStatus" : {
            "datatypeMismatchErrors" : 0,
            "fullUri" : "s3://bucket/key",
            "insertErrors" : 0,
            "parsingErrors" : 5,
```

```
            "retryNumber" : 0,
            "runNumber" : 1,
            "status" : "LOAD_FAILED",
            "totalDuplicates" : 0,
            "totalRecords" : 5,
            "totalTimeSpent" : 3.0
        }
    }
}
```

Example request for loadIds

The following is a request sent via HTTP `GET` using the `curl` command.

AWS CLI

```
aws neptunedata list-loader-jobs \
  --endpoint-url https://your-neptune-endpoint:port \
  --limit 3
```

For more information, see list-loader-jobs in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.list_loader_jobs(
    limit=3
)

print(response)
```

awscurl

```
awscurl 'https://your-neptune-endpoint:port/loader?limit=3' \
  --region us-east-1 \
```

```
  --service neptune-db
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -X GET 'https://your-neptune-endpoint:port/loader?limit=3'
```

Example Response

```
{
    "status" : "200 OK",
    "payload" : {
         "loadIds" : [
            "a2c0ce44-a44b-4517-8cd4-1dc144a8e5b5",
            "09683a01-6f37-4774-bb1b-5620d87f1931",
            "58085eb8-ceb4-4029-a3dc-3840969826b9"
        ]
    }
}
```

Example request for detailed status

The following is a request sent via HTTP `GET` using the `curl` command.

AWS CLI

```
aws neptunedata get-loader-job-status \
  --endpoint-url https://your-neptune-endpoint:port \
  --load-id loadId (a UUID) \
  --details
```

For more information, see get-loader-job-status in the AWS CLI Command Reference.

SDK

```
import boto3
```

```
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_loader_job_status(
    loadId='loadId (a UUID)',
    details=True
)

print(response)
```

awscurl

```
awscurl 'https://your-neptune-endpoint:port/loader/loadId (a UUID)?details=true' \
  --region us-east-1 \
  --service neptune-db
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl -X GET 'https://your-neptune-endpoint:port/loader/loadId (a UUID)?details=true'
```

Example Response

```
{
    "status" : "200 OK",
    "payload" : {
        "failedFeeds" : [
            {
                "datatypeMismatchErrors" : 0,
                "fullUri" : "s3://bucket/key",
```

```
                "insertErrors" : 0,
                "parsingErrors" : 5,
                "retryNumber" : 0,
                "runNumber" : 1,
                "status" : "LOAD_FAILED",
                "totalDuplicates" : 0,
                "totalRecords" : 5,
                "totalTimeSpent" : 3.0
            }
        ],
        "feedCount" : [
            {
                "LOAD_FAILED" : 1
            }
        ],
        "overallStatus" : {
            "datatypeMismatchErrors" : 0,
            "fullUri" : "s3://bucket/key",
            "insertErrors" : 0,
            "parsingErrors" : 5,
            "retryNumber" : 0,
            "runNumber" : 1,
            "status" : "LOAD_FAILED",
            "totalDuplicates" : 0,
            "totalRecords" : 5,
            "totalTimeSpent" : 3.0
        }
    }
}
```

Neptune Loader Get-Status `errorLogs` examples

The following examples showcase the detailed status response from the Neptune loader when errors have occurred during the data loading process. The examples illustrate the structure of the response, including information about failed feeds, overall status, and detailed error logs.

Example detailed status response when errors occurred

This a request sent via HTTP GET using `curl`:

AWS CLI

```
aws neptunedata get-loader-job-status \
```

```
  --endpoint-url https://your-neptune-endpoint:port \
  --load-id 0a237328-afd5-4574-a0bc-c29ce5f54802 \
  --details \
  --errors \
  --errors-per-page 3 \
  --page 1
```

For more information, see get-loader-job-status in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_loader_job_status(
    loadId='0a237328-afd5-4574-a0bc-c29ce5f54802',
    details=True,
    errors=True,
    errorsPerPage=3,
    page=1
)

print(response)
```

awscurl

```
awscurl 'https://your-neptune-endpoint:port/loader/0a237328-afd5-4574-a0bc-
c29ce5f54802?details=true&errors=true&page=1&errorsPerPage=3' \
  --region us-east-1 \
  --service neptune-db
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl -X GET 'https://your-neptune-endpoint:port/loader/0a237328-afd5-4574-a0bc-c29ce5f54802?details=true&errors=true&page=1&errorsPerPage=3'
```

Example of a detailed response when errors have occurred

This is an example of the response that you might get from the query above, with an `errorLogs` object listing the load errors encountered:

```
{
    "status" : "200 OK",
    "payload" : {
        "failedFeeds" : [
            {
                "datatypeMismatchErrors" : 0,
                "fullUri" : "s3://bucket/key",
                "insertErrors" : 0,
                "parsingErrors" : 5,
                "retryNumber" : 0,
                "runNumber" : 1,
                "status" : "LOAD_FAILED",
                "totalDuplicates" : 0,
                "totalRecords" : 5,
                "totalTimeSpent" : 3.0
            }
        ],
        "feedCount" : [
            {
                "LOAD_FAILED" : 1
            }
        ],
        "overallStatus" : {
            "datatypeMismatchErrors" : 0,
            "fullUri" : "s3://bucket/key",
            "insertErrors" : 0,
            "parsingErrors" : 5,
            "retryNumber" : 0,
            "runNumber" : 1,
            "status" : "LOAD_FAILED",
            "totalDuplicates" : 0,
            "totalRecords" : 5,
```

```
            "totalTimeSpent" : 3.0
        },
        "errors" : {
            "endIndex" : 3,
            "errorLogs" : [
                {
                    "errorCode" : "PARSING_ERROR",
                    "errorMessage" : "Expected '<', found: |",
                    "fileName" : "s3://bucket/key",
                    "recordNum" : 1
                },
                {
                    "errorCode" : "PARSING_ERROR",
                    "errorMessage" : "Expected '<', found: |",
                    "fileName" : "s3://bucket/key",
                    "recordNum" : 2
                },
                {
                    "errorCode" : "PARSING_ERROR",
                    "errorMessage" : "Expected '<', found: |",
                    "fileName" : "s3://bucket/key",
                    "recordNum" : 3
                }
            ],
            "loadId" : "0a237328-afd5-4574-a0bc-c29ce5f54802",
            "startIndex" : 1
        }
    }
}
```

Example of a `Data prefetch task interrupted` error

Occasionally when you get a `LOAD_FAILED` status and then request more detailed information, the error returned may be a `PARSING_ERROR` with a `Data prefetch task interrupted` message, like this:

```
"errorLogs" : [
    {
        "errorCode" : "PARSING_ERROR",
        "errorMessage" : "Data prefetch task interrupted: Data prefetch task for 11467
 failed",
        "fileName" : "s3://amzn-s3-demo-bucket/some-source-file",
        "recordNum" : 0
```

```
    }
]
```

This error occurs when there was a temporary interruption in the data load process that was typically not caused by your request or your data. It can usually be resolved simply by running the bulk upload request again. If you are using default settings, namely `"mode":"AUTO"`, and `"failOnError":"TRUE"`, the loader skips the files that it already successfully loaded and resumes loading files it had not yet loaded when the interruption occurred.

Neptune Loader Cancel Job

Cancels a load job.

To cancel a job, you must send an HTTP `DELETE` request to the `https://`*`your-neptune-endpoint`*`:`*`port`*`/loader` endpoint. The `loadId` can be appended to the `/loader` URL path, or included as a variable in the URL.

Cancel Job request syntax

```
DELETE https://your-neptune-endpoint:port/loader?loadId=loadId
```

```
DELETE https://your-neptune-endpoint:port/loader/loadId
```

Cancel Job Request Parameters

loadId

The ID of the load job.

Cancel Job Response Syntax

```
no response body
```

200 OK

Successfully deleted load job returns a `200` code.

Cancel Job Errors

When an error occurs, a JSON object is returned in the `BODY` of the response. The `message` object contains a description of the error.

Error Categories

- **`Error 400`** – An invalid `loadId` returns an HTTP `400` bad request error. The message describes the error.
- **`Error 500`** – A valid request that cannot be processed returns an HTTP `500` internal server error. The message describes the error.

Cancel Job Error Messages

The following are possible error messages from the cancel API with a description of the error.

- `The load with id = load_id does not exist or not active` (HTTP 404) – The load was not found. Check the value of `id` parameter.
- `Load cancellation is not permitted on a read replica instance.` (HTTP 405) – Loading is a write operation. Retry load on the read/write cluster endpoint.

Cancel Job Examples

Example Request

The following is a request sent via HTTP `DELETE` using the `curl` command.

AWS CLI

```
aws neptunedata cancel-loader-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --load-id 0a237328-afd5-4574-a0bc-c29ce5f54802
```

For more information, see cancel-loader-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)
```

```
response = client.cancel_loader_job(
    loadId='0a237328-afd5-4574-a0bc-c29ce5f54802'
)

print(response)
```

awscurl

```
awscurl 'https://your-neptune-endpoint:port/loader/0a237328-afd5-4574-a0bc-
c29ce5f54802' \
  --region us-east-1 \
  --service neptune-db \
  -X DELETE
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -X DELETE 'https://your-neptune-endpoint:port/loader/0a237328-afd5-4574-a0bc-
c29ce5f54802'
```

Using AWS Database Migration Service to load data into Amazon Neptune from a different data store

AWS Database Migration Service (AWS DMS) can load data into Neptune from supported source databases quickly and securely. The source database remains fully operational during the migration, minimizing downtime for applications that rely on it.

You can find detailed information about AWS DMS in the AWS Database Migration Service User Guide and the AWS Database Migration Service API Reference. In particular, you can find out how to set up a Neptune cluster as a target for migration in Using Amazon Neptune as a Target for AWS Database Migration Service.

Here are some prerequisites for importing data into Neptune using AWS DMS:

- You will need to create a AWS DMS table mapping object to define how data should be extracted from the source database (see Specifying table selection and transformations by table mapping using JSON in the AWS DMS Userguide for details). This table mapping configuration object specifies which tables should be read and in what order, and how their columns are named. It can also filter the rows being copied and provide simple value transformations such as converting to lower case or rounding.
- You will need to create a Neptune `GraphMappingConfig` to specify how the data extracted from the source database should be loaded into Neptune. For RDF data (queried using SPARQL), the `GraphMappingConfig` is written in the W3's standard R2RML mapping language. For property graph data (queried using Gremlin), the `GraphMappingConfig` is a JSON object, described in GraphMappingConfig Layout for Property-Graph/Gremlin Data.
- You must use AWS DMS to create a replication instance in the same VPC as your Neptune DB cluster, to mediate the transfer of data.
- You will also need an Amazon S3 bucket to be used as intermediate storage for staging the migration data.

Creating a Neptune GraphMappingConfig

The `GraphMappingConfig` that you create specifies how data extracted from a source data store should be loaded into a Neptune DB cluster. Its format differs depending on whether it is intended for loading RDF data or for loading property-graph data.

For RDF data, you can use the W3 R2RML language for mapping relational data to RDF.

If you are loading property-graph data to be queried using Gremlin, you create a JSON object for `GraphMappingConfig`.

GraphMappingConfig Layout for RDF/SPARQL Data

If you are loading RDF data to be queried using SPARQL, you write the `GraphMappingConfig` in R2RML. R2RML is a standard W3 language for mapping relational data to RDF. Here is one example:

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix ex: <http://example.com/ns#> .

<#TriplesMap1>
    rr:logicalTable [ rr:tableName "nodes" ];
    rr:subjectMap [
        rr:template "http://data.example.com/employee/{id}";
```

```
        rr:class ex:Employee;
    ];
    rr:predicateObjectMap [
        rr:predicate ex:name;
        rr:objectMap [ rr:column "label" ];
    ] .
```

Here is another example:

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix ex: <http://example.com/#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<#TriplesMap2>
    rr:logicalTable [ rr:tableName "Student" ];
    rr:subjectMap   [ rr:template "http://example.com/{ID}{Name}";
                      rr:class foaf:Person ];
    rr:predicateObjectMap [
        rr:predicate ex:id ;
        rr:objectMap  [ rr:column "ID";
                        rr:datatype xsd:integer ]
    ];
    rr:predicateObjectMap [
        rr:predicate foaf:name ;
        rr:objectMap  [ rr:column "Name" ]
    ] .
```

The W3 Recommendation at R2RML: RDB to RDF Mapping Language provides details of the language.

GraphMappingConfig Layout for Property-Graph/Gremlin Data

A comparable `GraphMappingConfig` for property-graph data is a JSON object that provides a mapping rule for each graph entity to be genereated from the source data. The following template shows what each rule in this object looks like:

```
{
  "rules": [
    {
      "rule_id": "(an identifier for this rule)",
      "rule_name": "(a name for this rule)",
```

```
      "table_name": "(the name of the table or view being loaded)",
      "vertex_definitions": [
        {
          "vertex_id_template": "{col1}",
          "vertex_label": "(the vertex to create)",
          "vertex_definition_id": "(an identifier for this vertex)",
          "vertex_properties": [
            {
              "property_name": "(name of the property)",
              "property_value_template": "{col2} or text",
              "property_value_type": "(data type of the property)"
            }
          ]
        }
      ]
    },
    {
      "rule_id": "(an identifier for this rule)",
      "rule_name": "(a name for this rule)",
      "table_name": "(the name of the table or view being loaded)",
      "edge_definitions": [
        {
          "from_vertex": {
            "vertex_id_template": "{col1}",
            "vertex_definition_id": "(an identifier for the vertex referenced above)"
          },
          "to_vertex": {
            "vertex_id_template": "{col3}",
            "vertex_definition_id": "(an identifier for the vertex referenced above)"
          },
          "edge_id_template": {
            "label": "(the edge label to add)",
            "template": "{col1}_{col3}"
          },
          "edge_properties":[
            {
              "property_name": "(the property to add)",
              "property_value_template": "{col4} or text",
              "property_value_type": "(data type like String, int, double)"
            }
          ]
        }
      ]
    }
```

```
  ]
}
```

Note that the presence of a vertex label implies that the vertex is being created here, whereas its absence implies that the vertex is created by a different source, and this definition is only adding vertex properties.

Here is a sample rule for an employee record:

```
{
  "rules": [
    {
      "rule_id": "1",
      "rule_name": "vertex_mapping_rule_from_nodes",
      "table_name": "nodes",
      "vertex_definitions": [
        {
          "vertex_id_template": "{emp_id}",
          "vertex_label": "employee",
          "vertex_definition_id": "1",
          "vertex_properties": [
            {
              "property_name": "name",
              "property_value_template": "{emp_name}",
              "property_value_type": "String"
            }
          ]
        }
      ]
    },
    {
      "rule_id": "2",
      "rule_name": "edge_mapping_rule_from_emp",
      "table_name": "nodes",
      "edge_definitions": [
        {
          "from_vertex": {
            "vertex_id_template": "{emp_id}",
            "vertex_definition_id": "1"
          },
          "to_vertex": {
            "vertex_id_template": "{mgr_id}",
            "vertex_definition_id": "1"
```

```
          },
          "edge_id_template": {
            "label": "reportsTo",
            "template": "{emp_id}_{mgr_id}"
          },
          "edge_properties":[
            {
              "property_name": "team",
              "property_value_template": "{team}",
              "property_value_type": "String"
            }
          ]
        }
      ]
    }
  ]
}
```

Creating an AWS DMS replication task with Neptune as the target

Once you have created your table mapping and graph mapping configurations, use the following process to load data from the source store into Neptune. Consult the AWS DMS documentation for more details about the APIs in question.

Create an AWS DMS replication instance

Create an AWS DMS replication instance in the VPC where your Neptune DB cluster is running (see Working with an AWS DMS Replication Instance and CreateReplicationInstance in the AWS DMS User Guide). You can use an AWS CLI command like the following to do that:

```
aws dms create-replication-instance \
    --replication-instance-identifier (the replication instance identifier) \
    --replication-instance-class (the size and capacity of the instance, like
 'dms.t2.medium') \
    --allocated-storage (the number of gigabytes to allocate for the instance
 initially) \
    --engine-version (the DMS engine version that the instance should use) \
    --vpc-security-group-ids (the security group to be used with the instance)
```

Create an AWS DMS endpoint for the source database

The next step is to create an AWS DMS endpoint for your source data store. You can use the AWS DMS CreateEndpoint API in the AWS CLI like this:

```
aws dms create-endpoint \
    --endpoint-identifier (source endpoint identifier) \
    --endpoint-type source \
    --engine-name (name of source database engine) \
    --username (user name for database login) \
    --password (password for login) \
    --server-name (name of the server) \
    --port (port number) \
    --database-name (database name)
```

Set up an Amazon S3 bucket for Neptune to use for staging data

If you do not have an Amazon S3 bucket that you can use for staging data, create one as explained in Creating a Bucket in the Amazon S3 Getting-Started Guide, or How Do I Create an S3 Bucket? in the Console User Guide.

You will need to create an IAM policy granting `GetObject`, `PutObject`, `DeleteObject` and `ListObject` permissions to the bucket if you do not already have one:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "ListObjectsInBucket",
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws-eusc:s3:::amzn-s3-demo-bucket"
      ]
    },
    {
      "Sid": "AllObjectActions",
```

```
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObject"
      ],
      "Resource": [
        "arn:aws-eusc:s3:::amzn-s3-demo-bucket/*"
      ]
    }
  ]
}
```

If your Neptune DB cluster has IAM authentication enabled, you will also need to include the following policy:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": "neptune-db:*",
      "Resource": "arn:aws-eusc:neptune-db:eusc-de-east-1:111122223333:cluster-
resource-id/*"
    }
  ]
}
```

Create an IAM role as a trust document to attach the policy to:

JSON

```
{
```

```
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "dms.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    },
    {
      "Sid": "neptune",
      "Effect": "Allow",
      "Principal": {
        "Service": "rds.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

After attaching the policy to the role, attach the role to your Neptune DB cluster. This will allow AWS DMS to use the bucket for staging the data being loaded.

Create an Amazon S3 endpoint in the Neptune VPC

Now create a VPC Gateway endpoint for your intermediary Amazon S3 bucket, in the VPC where your Neptune cluster is located. You can use the AWS Management Console or the AWS CLI to do this, as described in Creating a gateway endpoint.

Create an AWS DMS target endpoint for Neptune

Create an AWS DMS endpoint for your target Neptune DB cluster. You can use the AWS DMS CreateEndpoint API with the `NeptuneSettings` parameter like this::

```
aws dms create-endpoint \
    --endpoint-identifier (target endpoint identifier) \
    --endpoint-type target \
    --engine-name neptune \
    --server-name (name of the server) \
    --port (port number) \
    --neptune-settings '{ \
```

```
      "ServiceAccessRoleArn": "(ARN of the service access role)", \
      "S3BucketName": "(name of S3 bucket to use for staging files when migrating)", \
      "S3BucketFolder": "(name of the folder to use in that S3 bucket)", \
      "ErrorRetryDuration": (number of milliseconds to wait between bulk-load retries),
\
      "MaxRetryCount": (the maximum number of times to retry a failing bulk-load job),
\
      "MaxFileSize": (maximum file size, in bytes, of the staging files written to S3),
\
      "IamAuthEnabled": (set to true if IAM authentication is enabled on the Neptune
cluster) }'
```

The JSON object passed to the AWS DMS `CreateEndpoint` API in its `NeptuneSettings` parameter has the following fields:

- **`ServiceAccessRoleArn`** – *(required)* The ARN of an IAM role that permits fine-grained access to the S3 bucket used to stage migration of the data into Neptune. This Role should also have permissions to access your Neptune DB cluster if IAM authorization is enabled on it.
- **`S3BucketName`** – *(required)* For Full Load migration, the replication instance converts all RDS data into CSV, quad files and uploads them to this staging bucket in S3 and then bulk-loads them into Neptune.
- **`S3BucketFolder`** – *(required)* The folder to use in the S3 staging bucket.
- **`ErrorRetryDuration`** – *(optional)* The number of milliseconds to wait after a Neptune request fails before making a retry request. The default is 250.
- **`MaxRetryCount`** – *(optional)* The maximum number of retry requests AWS DMS should make after a retryable failure. The default is 5.
- **`MaxFileSize`** – *(optional)* The maximum size in bytes of each staging file saved to S3 during the migration. The default is 1,048,576 KB (1 GB).
- **`IsIAMAuthEnabled`** – *(optional)* Set to `true` if IAM authentication is enabled on the Neptune DB cluster, or `false` if not. The default is `false`.

Test connections to the new endpoints

You can test the connection to each of these new endpoints using the AWS DMS TestConnection API like this:

```
aws dms test-connection \
```

```
    --replication-instance-arn (the ARN of the replication instance) \
    --endpoint-arn (the ARN of the endpoint you are testing)
```

Create an AWS DMS replication task

Once you have completed the previous steps successfully, create a replication task for migrating data from your source data store to Neptune, using the AWS DMS CreateReplicationTask API like this:

```
aws dms create-replication-task \
    --replication-task-identifier (name for the replication task) \
    --source-endpoint-arn (ARN of the source endpoint) \
    --target-endpoint-arn (ARN of the target endpoint) \
    --replication-instance-arn (ARN of the replication instance) \
    --migration-type full-load \
    --table-mappings (table-mapping JSON object or URI like 'file:///tmp/table-
mappings,json') \
    --task-data (a GraphMappingConfig object or URI like 'file:///tmp/graph-mapping-
config.json')
```

The `TaskData` parameter provides the GraphMappingConfig that specifies how the data being copied should be stored in Neptune.

Start the AWS DMS replication task

Now you can start the replication task:

```
aws dms start-replication-task
    --replication-task-arn (ARN of the replication task started in the previous step)
    --start-replication-task-type start-replication
```

Loading data into Amazon Neptune using queries

Neptune supports writing data directly through query language operations. You can use standard write operations like `CREATE` and `MERGE` in openCypher, `INSERT` in SPARQL, or `mergeV()` and `mergeE()` in Gremlin to add or modify data in your graph. These operations are suitable for incremental updates and transactional writes.

For loading data from Amazon S3, use the Using the Amazon Neptune bulk loader to ingest data for large datasets requiring optimized performance. For smaller datasets in one or a few Amazon

S3 files, you can use query-based loading functions to read and process data directly within your queries.

The following query-based loading functions are available:

openCypher: neptune.read()

The `neptune.read()` function reads CSV or Parquet files from Amazon S3 within a `CALL` subquery, allowing you to process and load data at query time.

```
CALL neptune.read({
  source: "s3://bucket/data.csv",
  format: "csv"
})
YIELD row
CREATE (n:Person {id: row.id, name: row.name})
```

For complete documentation, see neptune.read().

SPARQL: LOAD and UNLOAD

SPARQL `LOAD` operations import RDF data from a URI into a named graph. `UNLOAD` exports data from a graph to Amazon S3.

```
LOAD <s3://bucket/data.ttl> INTO GRAPH <http://example.org/graph>
```

For complete documentation, see Using SPARQL UPDATE LOAD to import data into Neptune.

Gremlin: io() step

You can also use Gremlin's `g.io(URL).read()` step to read in data files in GraphML (an XML format), GraphSON (a JSON format), and other formats.

```
g.io("https://example.com/data/my-graph.graphml").read().iterate()
```

Use the `g.io()` step to read from an HTTPS URL. To load a file that you store as an Amazon S3 object, first generate a presigned URL. Then pass that HTTPS URL to `g.io()`. You can generate a presigned URL with the AWS CLI `aws s3 presign` command:

```
aws s3 presign s3://amzn-s3-demo-bucket/data/vertices.graphml
```

For more information about presigned URLs, see Download and upload objects with presigned URLs in the *Amazon S3 User Guide*.

See TinkerPop documentation for details.

Storage

Neptune data is stored in a cluster volume, which is a single, virtual volume that uses Non-Volatile Memory Express (NVMe) SSD-based drives. The cluster volume consists of a collection of logical blocks known as segments. Each of these segments is allocated 10 gigabytes (GB) of storage. The data in each segment is replicated across three Availability Zones (AZs) in a single AWS Region where the DB cluster resides.

When a Neptune DB cluster is created, it is allocated a single segment of 10 GB. As the volume of data increases and exceeds the currently allocated storage, Neptune automatically expands the cluster volume by adding new segments. A Neptune cluster volume can grow to a maximum size of 128 tebibytes (TiB) in all supported Regions except the China Regions and the AWS GovCloud (US) Regions, where it is limited to 64 TiB.

The DB cluster volume contains all your user data, indices and dictionaries (described in the Neptune Graph Data Model section), as well as internal metadata such as internal transaction logs. All this graph data, including indices and internal logs, cannot exceed the maximum size of the cluster volume.

I/O–Optimized storage option

Neptune offers two pricing models for storage:

- **Standard storage** – Standard storage provides cost-effective database storage for applications with moderate to low I/O usage.
- **I/O–Optimized storage** – With I/O–Optimized storage, you pay only for the storage you are using, at a higher cost than for standard storage, and you pay nothing for the I/O that you use.

  With I/O–Optimized storage, you get predictable costs, low I/O latency, and consistent I/O throughput for I/O–intensive graph workloads.

  For more information, see I/O–Optimized storage.

Neptune storage allocation

Even though a Neptune cluster volume can grow to 128 TiB (or 64 TiB in a few regions), you are charged for space allocated. The total space allocated is determined by the storage *high*

water mark, which is the maximum amount allocated to the cluster volume at any time during its existence.

This means that even if you remove user data from a cluster volume, such as by running a drop query like `g.V().drop()`, the total allocated space remains the same. Neptune does automatically optimize the unused allocated space for reuse in the future.

In addition to user data, dictionary data and internal transaction logs also consume storage. Dictionary data persists even after you delete the graph data it supports, though Neptune can re-use those entries if you re-introduce the data. To clean up unused dictionary entries, you can enable Dictionary garbage collection. Internal logs use a separate storage space with its own high water mark. Expired logs are re-used only for other logs, not for graph data. The space allocated for logs is included in the total reported by the `VolumeBytesUsed` CloudWatch metric.

Check Storage best practices for ways to keep allocated storage to a minimum and to re-use space.

Neptune storage billing

Storage costs are billed based on the storage *high water mark*, as described in the preceding section. Amazon Neptune replicates your data across multiple Availability Zones, but you pay for only one copy of the data.

You can determine what the current storage high water mark of your DB cluster is by monitoring the `VolumeBytesUsed` CloudWatch metric (see Monitoring Neptune Using Amazon CloudWatch).

Other factors that can affect your Neptune storage costs include database snapshots and backup, which are billed separately as backup storage and are based on the Neptune storage costs (see CloudWatch metrics that are useful for managing Neptune backup storage).

If you create a clone of your database, however, the clone points to the same cluster volume that your DB cluster itself uses, so there is no additional storage charge for the original data. Subsequent changes to the clone use the copy-on-write protocol, and do result in additional storage costs.

For more Neptune pricing information, see Amazon Neptune Pricing on the AWS website.

Neptune storage usage breakdown

With the `GetStorageInfo` API, you can get a detailed breakdown of storage usage for your Neptune database, including undo logs, dictionary entries, change-stream data, DFE statistics,

property graph schema, other internal metadata tables, and all of the quad indexes – grouped by property graph (PG) and Resource Description Framework (RDF). The response also includes a `totalStorageBytesUsed` value that sums the reported on-disk components. You can use this information to understand how your storage is distributed, to identify storage bloat, and to establish a baseline before and after an online index rebuild.

Use `GetStorageInfo` in the following situations:

- To understand how your storage is distributed across indexes and dictionaries.
- To identify storage bloat after bulk deletes or data migrations.
- To establish a baseline before and after an index rebuild.
- To monitor storage trends over time.

Prerequisites

Engine version

The `GetStorageInfo` API requires Neptune engine release 1.5.0.0 or later.

IAM permissions

If your cluster uses IAM authentication, the IAM policy attached to the requesting principal must allow the `neptune-db:GetStorageInfo` action:

```
{
  "Effect": "Allow",
  "Action": ["neptune-db:GetStorageInfo"],
  "Resource": "arn:aws:neptune-db:<region>:<account-id>:<cluster-resource-id>/*"
}
```

For more information about data-plane IAM actions, see IAM actions for data access in Amazon Neptune.

Request

To retrieve storage information, send an HTTP `GET` request to the `/storage/info` endpoint. The request takes no body and no query parameters.

AWS CLI

```
aws neptunedata get-storage-info \
  --endpoint-url https://your-neptune-endpoint:port
```

For more information, see get-storage-info in the AWS CLI Command Reference.

SDK

```
import boto3
import json
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_storage_info()

print(json.dumps(response, indent=2, default=str))
```

awscurl

```
awscurl https://your-neptune-endpoint:port/storage/info \
  --region us-east-1 \
  --service neptune-db
```

> **Prerequisites**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl -G https://your-neptune-endpoint:port/storage/info
```

Response

The `GetStorageInfo` API returns a JSON response that reports the on-disk storage used by each Neptune internal component: the dictionaries, undo logs, change streams, DFE statistics, property graph schema, other internal metadata tables, and each quad index. Neptune reports quad index sizes per data model under `PG` (property graph) and `RDF`, and includes an index only when it contains data. Neptune omits the following top-level sections from the response when they are empty: `dictionaries.blobs`, `streams`, `dfeStatistics`, `schema`, and `internals`.

The following example shows the response for a property graph cluster. Because this cluster contains only property graph data, the `RDF` object does not appear in the response.

```
{
  "totalStorageBytesUsed": 955367424,
  "dictionaries": {
    "total": { "sizeInBytes": 259080192, "count": 653990 },
    "blobs": { "sizeInBytes":  54657024, "count":  22323 }
  },
  "undoLogs":      { "sizeInBytes": 150994944, "listSize": 2290 },
  "streams":       { "sizeInBytes":  77594624, "count": 1238185 },
  "dfeStatistics": { "sizeInBytes":    229376, "count":     813 },
  "schema":        { "sizeInBytes":     65536, "count":     154 },
  "internals":     { "sizeInBytes":    753664, "count":     202 },
  "quads": {
    "PG": {
      "SPOG":  { "sizeInBytes": 177995776, "count": 1222436 },
      "POGS":  { "sizeInBytes":  96534528, "count": 1222436 },
      "GPSO":  { "sizeInBytes":  79036416, "count": 1222436 },
      "OSGP":  { "sizeInBytes": 104071168, "count": 1222436 },
      "PZSOG": { "sizeInBytes":   9011200, "count":   61149 }
    }
  }
}
```

The response contains the following fields:

Section	Field	Description
(top-level)	`totalStorageBytesUsed`	Total on-disk storage used by Neptune. This is the sum of the `sizeInBytes` values of `dictionaries.total`, `undoLogs`, `streams`, `dfeStatis`

Section	Field	Description
		`tics` , `schema`, `internals` , and every quad index. `dictionaries.blobs` is already included in `dictionaries.total` , so it is not added again. The total also includes components that are omitted from the response because they are empty, so it can be slightly larger than the sum of the values shown.
`dictionaries.total`	`count` / `sizeInBytes`	Approximate number of distinct dictionary entries and their total storage size. Neptune persists unique dictionary entries indefinitely, so this count might not reflect the number of entries currently referenced by your data.
`dictionaries.blobs`	`count` / `sizeInBytes`	Approximate number of dictionary entries stored as blobs (values too large to inline) and their storage size. These entries are a subset of `dictionaries.total` . Omitted when no blob entries exist.
`undoLogs`	`sizeInBytes`	Storage allocated for the Neptune transactional system to track modifications. This value reflects the current on-disk size of the undo tablespace, which grows over time and does not shrink automatically.
`undoLogs`	`listSize`	Current number of uncommitted undo records not yet purged. This value mirrors the `UndoLogListSize` CloudWatch metric. Because Neptune purges undo entries asynchronously in the background, this value can fluctuate rapidly under active write workloads.
`streams`	`count` / `sizeInBytes`	Approximate number of records in the Neptune change-log stream and their storage size. For more information, see Capturing graph changes in real time using Neptune streams. Omitted when Neptune streams are not enabled or the stream is empty.

Section	Field	Description
`dfeStatistics`	`count` / `sizeInBytes`	Approximate number of rows in the DFE statistics tables and their storage size. For more information, see Managing statistics for the Neptune DFE to use. Omitted when no statistics have been computed.
`schema`	`count` / `sizeInBytes`	Approximate number of rows in the property graph schema tables (label, property-key, and edge metadata) and their storage size. Omitted when the cluster contains no property graph data.
`internals`	`count` / `sizeInBytes`	Approximate number of rows in other Neptune internal metadata tables and their storage size. Omitted when empty.
`quads.PG.*`	`count` / `sizeInBytes`	Entry count and storage size for each property graph index (Gremlin and openCypher): SPOG, POGS, GPSO, OSGP, and PZSOG.
`quads.RDF.*`	`count` / `sizeInBytes`	Entry count and storage size for each RDF index (SPARQL): SPOG, POGS, GPSO, OSGP, and PZSOG.

Note the following about the response:

- Neptune omits any top-level section or per-index entry whose `count` is `0`. An absent entry means that the component holds no data. Both the PG and RDF objects can appear if your cluster has data in both data models.
- The OSGP index is optional. For more information about Neptune storage, see Neptune storage.
- The PZSOG index is absent unless it is populated on your cluster. Starting with engine release 1.4.7.0, the PZSOG index is enabled by default for all customers.

Interpreting the values

The `sizeInBytes` values reflect the on-disk allocation for each component at the time of the response. The `count` values are approximations. Neptune derives them from InnoDB row-count statistics, which Neptune refreshes periodically. These statistics can drift

from true values, particularly after large ingestion or deletion workloads. For the SPOG, POGS, GPSO, and OSGP quad indexes, `count` is a uniform approximation across those four indexes; the PZSOG index reports its own count. For details on how Neptune refreshes these statistics, see Managing statistics for the Neptune DFE to use. Use `GetStorageInfo` to understand the relative distribution of storage across your indexes and dictionaries, and use `VolumeBytesUsed` for billing and capacity planning.

Counting graph elements

If you need counts of the vertices, edges, or other elements in your graph, use the graph summary API instead of `GetStorageInfo`. The graph summary API reports node, edge, and property counts for property graphs, or quad, subject, and predicate counts for RDF graphs. You can manually trigger a statistics update right before calling it to get the freshest values possible. For more information, see Getting a quick summary report about your graph.

GetStorageInfo vs VolumeBytesUsed

The sizes reported by `GetStorageInfo` are not expected to add up to the CloudWatch `VolumeBytesUsed` metric. `GetStorageInfo` reports the logical storage consumed by specific Neptune components, whereas `VolumeBytesUsed` reflects the total physical storage billed at the volume level. The volume also contains internal overhead that is not attributed to any component in the `GetStorageInfo` response, including:

- Block-level allocation
- System metadata
- Transaction and redo logs
- Temporary space for ongoing operations
- Space that has been allocated but not yet reclaimed

Use `GetStorageInfo` to understand the relative distribution of storage across your indexes and dictionaries, and use `VolumeBytesUsed` for billing and capacity planning.

Errors

The following table lists the errors that `GetStorageInfo` can return.

HTTP status	Error code	Description
403	`AccessDeniedException`	The request does not have the required IAM permission.
404	`ResourceNotFoundException`	The specified database was not found.
429	`ThrottlingException`	The request was denied due to request throttling.
500	`InternalServerException`	An internal service error occurred.

Neptune storage best practices

Because certain types of data consume permanent storage in Neptune, use these best practices to avoid large spikes in storage growth:

- When designing your graph data model, avoid as much as possible using property keys and user-facing values that are temporary in nature.
- If you plan on making changes to your data model, do not load data onto an existing DB cluster using the new model until you have cleared the data in that DB cluster using the fast reset API. The best thing is often to load data that uses a new model onto a new DB cluster.
- Transactions that operate on large amounts of data generate correspondingly large internal logs, which can permanently increase the high water mark of the internal log space. For example, a single transaction that deletes all the data in your DB cluster can generate a large internal log. This log requires allocating a great deal of internal storage, which permanently reduces the space available for graph data.

  To avoid this, split large transactions into smaller ones and allow time between them so that the associated internal logs have a chance to expire and release their internal storage for re-use by subsequent logs.

- For monitoring the growth of your Neptune cluster volume, you can set a CloudWatch alarm on the `VolumeBytesUsed` CloudWatch metric. This can be particularly helpful if your data is reaching the maximum size of the cluster volume. For more information, see Using Amazon CloudWatch alarms.
- Use Neptune storage usage breakdown to see how your storage is distributed across dictionaries, quad indexes, and internal metadata tables before you decide how to reclaim space. Large index sizes relative to entry counts indicate fragmentation. High dictionary counts after large deletes indicate bloat.

If your DB cluster has significant unused allocated space, you can reclaim it in one of two ways:

- **Online index rebuild.** Run an online index rebuild to defragment the Neptune indexes and reclaim space in place, without taking your database offline. Use Neptune storage usage breakdown before and after to quantify the space reclaimed.
- **Export and reload.** Export all the data in your graph and then reload it into a new DB cluster. See the Neptune data export service and utility for an easy way to export data from a DB cluster, and the Neptune bulk loader for an easy way to import data back into Neptune.

> **Note**
>
> Creating and restoring a snapshot does not reduce the amount of storage allocated for your DB cluster, because a snapshot retains the original image of the cluster's underlying storage. To reclaim unused storage, run an online index rebuild, or export your graph data and reload it into a new DB cluster.

Neptune storage reliability and high availability

Amazon Neptune is designed to be reliable, durable, and fault tolerant.

Because copies of your Neptune data are maintained across three Availability Zones (AZs), storage of the data is highly durable, with very low likelihood of data loss. Amazon Neptune replicates the data automatically across the Availability Zones, regardless of whether there are DB instances in them. The amount of replication is independent of the number of DB instances in your cluster.

This means that you can add a read-replica quickly, because Neptune doesn't make a new copy of the graph data. Instead, the read-replica connects to the cluster volume that already contains your data. Similarly, removing a read-replica doesn't remove any of the underlying data.

You can delete the cluster volume and its data only after deleting all of its DB instances.

Neptune also automatically detects failures in the segments that make up the cluster volume. When a copy of the data in a segment is corrupted, Neptune immediately repairs that segment, using other copies of the data within the same segment to ensure that the repaired data is current. As a result, Neptune avoids data loss and reduces the need to perform point-in-time restore to recover from a disk failure.

Neptune storage cleanup

You can reclaim Neptune storage in two ways. Both keep your database fully operational.

- **Dictionary garbage collection.** Reclaims dictionary entries that are no longer referenced.
- **Online index rebuild.** Defragments the quad indexes and reclaims unused space in place.

Neptune dictionary garbage collection

Neptune supports dictionary garbage collection (GC) which can be enabled via the `neptune_lab_mode` parameter for property graph data. The feature requires a writer instance reboot to activate. This feature is available from engine release 1.4.3.0.

> **Compatibility with engine releases before 1.4.8.0**
>
> Prior to engine release 1.4.8.0, you could enable dictionary GC only on clusters without `neptune_streams` enabled. Neptune automatically disabled the feature if `neptune_streams` was enabled. Neptune also disabled the feature if any unexpired `neptune_streams` data existed. Starting with engine release 1.4.8.0, this restriction no longer applies, and you can enable dictionary GC on clusters that have `neptune_streams` enabled.

When enabled, the unused dictionary entries are cleaned up by a background job. It does not reduce `VolumeBytesUsed`, instead it frees up space in the index for new inserts. The rate of

growth in `VolumeBytesUsed` is likely to be less when dictionary GC is enabled relative to when it is not.

Dictionary garbage collection runs in the background and scans all graph and dictionary data to find terms that are not in use. A new run is triggered on start up once approximately 6% of the data has changed. GC competes with query threads for server resources such as CPU, buffer cache, undo log generation, and write I/O operations, potentially reducing query throughput. Since GC scans data that is not actively touched by queries, it can impact the buffer cache on the writer node. The cluster could see additional write I/O operations and have more undo logs to purge as GC performs new deletes, which may also result in higher values for the `UndoLogListSize` metric.

GC can be run in two modes, `soft_delete` and `enabled`. When run in the `soft_delete` mode, unused dictionary entries are marked deleted (soft_delete) but are not explicitly deleted. This mode could also be used to understand performance characteristics after the background operation is turned on. When the `enabled` mode is used, entries are explicitly deleted ('hard' delete). It is recommended to run GC in `soft_delete` mode for a period of time before switching to `enabled` mode.

Dictionary GC supports a maximum concurrency of 16 (on machines with 16 or more cores). It runs by default with a single thread, but it can be run with higher concurrency when enabled for the first time. Dictionary GC thread(s) run at equal priority with the query threads, and they contend with resources on the writer equally.

Dictionary GC can be enabled via the `neptune_lab_mode` parameter by setting the `DictionaryGCMode` key. It accepts three possible values: `disabled` (default), `soft_delete`, or `enabled`. For example, the following code sample would set the `DictionaryGCMode` to `soft_delete`:

```
neptune_lab_mode = 'DictionaryGCMode=soft_delete'
```

The concurrency parameter, `DictionaryGCConcurrency`, is optional and can take a value between 1 and 16. If set to a higher value than the minimum of 16 and number of cores, the concurrency is capped at that value.

```
neptune_lab_mode = 'DictionaryGCMode=soft_delete,DictionaryGCConcurrency=2'
```

The dictionary GC job is enabled in the background after the server starts, once there is some data available. The engine status displays the current status of dictionary GC. The example output

shown below shows that dictionary GC is in `soft_delete` mode and running with a concurrency of 2. If the background job is running, it could be actively scanning for unused dictionary entries and deleting them, or waiting for new set of deletes to trigger a new round of GC.

```
{"status":"healthy",...,"labMode":
{"ObjectIndex":"disabled","DictionaryGC":"{Mode=enabled,Concurrency=2}"},...}
```

Dictionary GC is paused when any of these conditions are met:

- Active bulk load.
- Freeable memory is less than 15Gb.
- `UndoLogListSize` is higher than 1,000,000.

Neptune online index rebuild

With online index rebuild, you can defragment Neptune indexes and reclaim unused storage space while your database stays fully operational. This lets you optimize storage costs and improve query performance without taking your database offline. Over time, insert, update, and delete operations create gaps of unused space within data pages, which fragments the indexes. Fragmentation keeps the `VolumeBytesUsed` metric high even after you delete data. It also increases I/O costs, because the same logical data spreads across more physical storage pages. An online index rebuild compacts the indexes to reclaim this space and reduce fragmentation.

During the online rebuild, your database remains fully available. You can continue running concurrent read and write queries in Gremlin, openCypher, and SPARQL.

You can rebuild all indexes (quads and dictionaries), only the quad indexes, or only the dictionary indexes. The online index rebuild APIs are available starting with Neptune engine release 1.5.0.0.

Prerequisites

Engine version

The online index rebuild APIs require Neptune engine release 1.5.0.0 or later.

Writer instance

You must initiate a rebuild on the writer instance, and the writer must be a provisioned instance. Online index rebuild is not supported on a Neptune Serverless writer instance. Read replicas are unaffected by the rebuild and can be either serverless or provisioned.

IAM permissions

If your cluster uses IAM authentication, the IAM policy attached to the requesting principal must allow the index rebuild actions:

```
{
  "Effect": "Allow",
  "Action": [
    "neptune-db:StartIndexRebuild",
    "neptune-db:GetIndexRebuildStatus",
    "neptune-db:ListIndexRebuilds",
    "neptune-db:CancelIndexRebuild"
  ],
  "Resource": "arn:aws:neptune-db:<region>:<account-id>:<cluster-resource-id>/*"
}
```

For more information about data-plane IAM actions, see IAM actions for data access in Amazon Neptune.

When to use it

- After significant data deletions (from queries or dictionary garbage collection), when `VolumeBytesUsed` stays high despite less data.
- When your database is approaching the storage limit and you need to reclaim space.
- As periodic maintenance (for example, quarterly or yearly) to keep indexes optimized.

What to expect during a rebuild:

- **Your database stays online.** Read queries and write queries continue to work on the writer instance. Read replicas are unaffected.
- **Transient query errors are possible.** During the rebuild, some queries can fail with a retriable `CancelledByMaintenanceException`. Your application must catch this exception, treat it like a `ConcurrentModificationException`, and retry with exponential backoff.
- **There can be a small performance impact** on queries. The rebuild competes for resources such as the buffer cache and I/O.
- **Neptune rejects bulk load API calls during a rebuild.** If you call the bulk load API (`POST /loader`) while an index rebuild is in progress, the request fails with a

`CancelledByMaintenanceException`. Cancel the rebuild with `CancelIndexRebuild` and wait for it to reach `CANCELLED`, or wait for the rebuild to complete, and then retry the load.

- **Neptune rejects fast reset API calls during a rebuild.** A call to the fast reset API (`ExecuteFastReset`) fails with a `CancelledByMaintenanceException` while an index rebuild is in progress. Cancel the rebuild with `CancelIndexRebuild` and wait for it to reach `CANCELLED`, or wait for the rebuild to complete, and then retry.

API reference

Online index rebuild includes four APIs: `StartIndexRebuild`, `GetIndexRebuildStatus`, `ListIndexRebuilds`, and `CancelIndexRebuild`. All of them must be sent to the writer instance. For the complete request and response members of each operation, see the Neptune data plane API reference.

StartIndexRebuild

Initiates a rebuild operation. Send an HTTP `POST` request to the `/storage/indexes/rebuilds` endpoint.

```
POST https://your-neptune-endpoint:port/storage/indexes/rebuilds
POST https://your-neptune-endpoint:port/storage/indexes/rebuilds?scope=QUADS
POST https://your-neptune-endpoint:port/storage/indexes/rebuilds?scope=DICTIONARIES
```

The request accepts the following optional query parameters:

- **`scope`**—The scope of the rebuild.

  Type: String. *Valid values*: `ALL` (quads and dictionaries), `QUADS` (quad indexes only), or `DICTIONARIES` (dictionary indexes only). *Default*: `ALL`.

- **`concurrency`**—The number of parallel workers used for the rebuild.

  Type: Integer. *Valid values*: An integer between 1 and the maximum number of vCPUs available on the writer instance type. If the value exceeds the maximum available vCPUs, Neptune caps it at the maximum rather than returning an error. *Default*: 1.

  If your workload is heavy, use a lower concurrency to reduce contention with your other workloads.

- **`writeBufferSize`**— Controls the size of the buffer that tracks concurrent write operations during the rebuild. If your database has high write traffic during the rebuild, a larger buffer helps prevent the operation from failing due to buffer overflow.

  Type: String. *Valid values*: `LOW` (light to moderate write activity), `MEDIUM` (sustained write traffic), or `HIGH` (heavy write workloads). *Default*: `LOW`.

The following examples show several ways to start a rebuild:

AWS CLI

```
# Rebuild everything (default)
aws neptunedata start-index-rebuild \
  --endpoint-url https://your-neptune-endpoint:port

# Rebuild quad indexes only
aws neptunedata start-index-rebuild \
  --endpoint-url https://your-neptune-endpoint:port \
  --scope QUADS

# Rebuild with 4 workers and a high write buffer
aws neptunedata start-index-rebuild \
  --endpoint-url https://your-neptune-endpoint:port \
  --scope ALL \
  --concurrency 4 \
  --write-buffer-size HIGH
```

For more information, see start-index-rebuild in the AWS CLI Command Reference.

SDK

```
import boto3
import json
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

# Rebuild everything (default)
```

```
response = client.start_index_rebuild()

# Rebuild quad indexes only
response = client.start_index_rebuild(Scope='QUADS')

# Rebuild with 4 workers and a high write buffer
response = client.start_index_rebuild(
    Scope='ALL',
    Concurrency=4,
    WriteBufferSize='HIGH'
)

print(json.dumps(response, indent=2, default=str))
```

awscurl

```
# Rebuild everything (default)
awscurl https://your-neptune-endpoint:port/storage/indexes/rebuilds \
  --region us-east-1 \
  --service neptune-db \
  -X POST

# Rebuild quad indexes only
awscurl "https://your-neptune-endpoint:port/storage/indexes/rebuilds?scope=QUADS" \
  --region us-east-1 \
  --service neptune-db \
  -X POST

# Rebuild with 4 workers and a high write buffer
awscurl "https://your-neptune-endpoint:port/storage/indexes/rebuilds?
concurrency=4&writeBufferSize=HIGH" \
  --region us-east-1 \
  --service neptune-db \
  -X POST
```

Prerequisites

This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
# Rebuild everything (default)
curl -X POST https://your-neptune-endpoint:port/storage/indexes/rebuilds

# Rebuild quad indexes only
curl -X POST "https://your-neptune-endpoint:port/storage/indexes/rebuilds?
scope=QUADS"

# Rebuild with 4 workers and a high write buffer
curl -X POST "https://your-neptune-endpoint:port/storage/indexes/rebuilds?
concurrency=4&writeBufferSize=HIGH"
```

A successful request returns HTTP 200 with the initial rebuild state:

```
{
  "status": "PREPARING",
  "rebuildId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "scope": "ALL",
  "concurrency": 1,
  "writeBufferSize": "LOW",
  "startTime": "2026-04-22T22:58:45Z"
}
```

Save the `rebuildId` from the response—you use it to query status and to cancel the rebuild. Only one rebuild can run at a time. If you start a rebuild while one is already in progress, you receive a `409 ConflictException`.

Errors

The following table lists the errors that `StartIndexRebuild` can return.

HTTP status	Error code	Description
400	`InvalidParameterException`	One or more parameters are invalid.
409	`ConflictException`	A rebuild is already in progress for this database.

HTTP status	Error code	Description
429	`ThrottlingException`	The request was denied due to request throttling.
500	`InternalServerException`	An internal service error occurred.

GetIndexRebuildStatus

Returns the status and progress of a rebuild operation. Send an HTTP `GET` request with the `rebuildId` appended to the path.

AWS CLI

```
aws neptunedata get-index-rebuild-status \
  --endpoint-url https://your-neptune-endpoint:port \
  --rebuild-id rebuildId
```

For more information, see get-index-rebuild-status in the AWS CLI Command Reference.

SDK

```
import boto3
import json
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_index_rebuild_status(
    RebuildId='rebuildId'
)

print(json.dumps(response, indent=2, default=str))
```

awscurl

```
awscurl https://your-neptune-endpoint:port/storage/indexes/rebuilds/rebuildId \
  --region us-east-1 \
  --service neptune-db
```

> **Prerequisites**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl -G https://your-neptune-endpoint:port/storage/indexes/rebuilds/rebuildId
```

The response reports the current state of the rebuild:

```
{
  "status": "REBUILDING",
  "rebuildId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "scope": "ALL",
  "concurrency": 1,
  "writeBufferSize": "LOW",
  "startTime": "2026-04-22T02:00:05Z",
  "estimatedEndTime": "2026-04-23T02:00:05Z",
  "progressPercentage": 45
}
```

The response contains the following fields:

Field	Description
`status`	The current status: `PREPARING` , `REBUILDING` , `COMPLETED` , `FAILED`, `CANCELLING` , or `CANCELLED` .
`rebuildId`	The unique identifier for the rebuild operation (UUID).
`scope`	The scope of the rebuild (`ALL`, `QUADS`, or `DICTIONARIES`).

Field	Description
`concurrency`	The concurrency of the rebuild operation.
`writeBufferSize`	The write buffer size of the rebuild operation (`LOW`, `MEDIUM`, or `HIGH`).
`startTime`	When the rebuild started (ISO-8601).
`estimatedEndTime`	The estimated completion time (ISO-8601). Present during `REBUILDING` and `CANCELLING`.
`endTime`	When the rebuild reached a terminal state (ISO-8601). Present for `COMPLETED`, `FAILED`, or `CANCELLED`.
`progressPercentage`	The percentage of rebuild completion (0–100).
`failureReason`	The reason for failure. Present only when `status` is `FAILED`.

Errors

The following table lists the errors that `GetIndexRebuildStatus` can return.

HTTP status	Error code	Description
404	`ResourceNotFoundException`	The rebuild ID was not found.
429	`ThrottlingException`	The request was denied due to request throttling.
500	`InternalServerException`	An internal service error occurred.

ListIndexRebuilds

Returns the list of rebuild operation IDs. Send an HTTP `GET` request to the `/storage/indexes/rebuilds` endpoint.

AWS CLI

```
aws neptunedata list-index-rebuilds \
  --endpoint-url https://your-neptune-endpoint:port
```

For more information, see list-index-rebuilds in the AWS CLI Command Reference.

SDK

```
import boto3
import json
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.list_index_rebuilds()

print(json.dumps(response, indent=2, default=str))
```

awscurl

```
awscurl https://your-neptune-endpoint:port/storage/indexes/rebuilds \
  --region us-east-1 \
  --service neptune-db
```

> **Prerequisites**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl -G https://your-neptune-endpoint:port/storage/indexes/rebuilds
```

The response contains a `rebuildIds` array. Use each ID with `GetIndexRebuildStatus` to retrieve detailed status.

```
{
  "rebuildIds": [
    "a1b2c3d4-e5f6-7890-abcd-ef1234567890"
  ]
}
```

> **Rebuild history**
>
> Only the most recent rebuild is retained. For details, see Limitations.

Errors

The following table lists the errors that `ListIndexRebuilds` can return.

HTTP status	Error code	Description
429	`ThrottlingException`	The request was denied due to request throttling.
500	`InternalServerException`	An internal service error occurred.

CancelIndexRebuild

Cancels an in-progress rebuild. Send an HTTP POST request to the `cancel` sub-resource of the rebuild.

AWS CLI

```
aws neptunedata cancel-index-rebuild \
  --endpoint-url https://your-neptune-endpoint:port \
  --rebuild-id rebuildId
```

For more information, see cancel-index-rebuild in the AWS CLI Command Reference.

SDK

```
import boto3
import json
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.cancel_index_rebuild(
    RebuildId='rebuildId'
)

print(json.dumps(response, indent=2, default=str))
```

awscurl

```
awscurl https://your-neptune-endpoint:port/storage/indexes/rebuilds/rebuildId/cancel
 \
  --region us-east-1 \
  --service neptune-db \
  -X POST
```

> **Prerequisites**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl -X POST https://your-neptune-endpoint:port/storage/indexes/rebuilds/rebuildId/
cancel
```

The `CancelIndexRebuild` API makes a best-effort attempt to cancel the rebuild. When you call it, the rebuild transitions to `CANCELLING` status. During this phase, work that is already in progress must finish before the rebuild reaches `CANCELLED` status. This can take from minutes to hours

depending on what is currently in flight. If the rebuild is still in PREPARING (no work has started), it transitions directly to CANCELLED. Your database remains fully operational throughout.

```
{
  "status": "CANCELLING",
  "rebuildId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "scope": "ALL",
  "concurrency": 1,
  "writeBufferSize": "LOW",
  "startTime": "2026-04-22T02:00:05Z",
  "estimatedEndTime": "2026-04-22T22:00:00Z",
  "progressPercentage": 37
}
```

Errors

The following table lists the errors that CancelIndexRebuild can return.

HTTP status	Error code	Description
400	InvalidParameterException	The rebuild is not in a cancellable state (for example, it is already COMPLETED , FAILED, or CANCELLED).
404	ResourceNotFoundException	The rebuild ID was not found.
429	ThrottlingException	The request was denied due to request throttling.
500	InternalServerException	An internal service error occurred.

Status lifecycle

The following description shows the possible state transitions from initiation through completion or cancellation:

A rebuild starts in PREPARING. From PREPARING, it either transitions directly to CANCELLED (if you cancel before any work starts) or advances to REBUILDING. From REBUILDING, the rebuild can

complete successfully (`COMPLETED`), fail (`FAILED`), or enter `CANCELLING`. A `CANCELLING` rebuild eventually reaches `CANCELLED`.

- `PREPARING`—The rebuild is initializing.
- `REBUILDING`—Index tables are actively being rebuilt. Your database remains fully operational.
- `COMPLETED`—The rebuild finished successfully.
- `FAILED`—The rebuild encountered an error. Your database is fully operational. Check `failureReason` for details.
- `CANCELLING`—A cancel was requested and in-flight work is draining. This can take minutes to hours depending on what is currently in flight.
- `CANCELLED`—The rebuild was cancelled. Your database is fully operational.

Storage safety mechanisms

The online index rebuild includes built-in safety mechanisms to prevent storage exhaustion during the rebuild process:

- **Storage budget with safety margin.** The rebuild applies an 80% safety margin against available volume space. A rebuild also requires additional scratch space while it runs. At maximum concurrency, this is up to roughly 25% of the data being rebuilt; the amount decreases proportionally at lower concurrency. The budget reserves headroom for this scratch space plus the concurrent writes and transaction-log growth that occur during the rebuild.
- **Adaptive scheduling.** The rebuild continuously checks how much free volume space is available and adjusts its parallelism accordingly. When free space is tight, the rebuild slows down to reduce peak storage consumption. When more space is available, the rebuild runs more work in parallel to finish faster.
- **Fail-fast on storage exhaustion.** If the rebuild cannot proceed after all in-flight work completes and the budget refreshes, it fails immediately. Neptune sets the `failureReason` message to `Insufficient storage to perform index rebuild`. The rebuild does not skip work silently.
- **100 TB default limit.** By default, an index rebuild request fails immediately on a cluster with more than 100 TB of `VolumeBytesUsed`. This limit is a safety precaution. A rebuild on a large cluster can require significant additional scratch space (up to roughly 25% of the data being rebuilt at maximum concurrency). This can push a cluster near or over its storage limit.

If your cluster exceeds 100 TB and you need to run an index rebuild, contact AWS Support to open a case with the Neptune team. We can review your cluster's size, growth rate, and workload, and work with you to plan the rebuild safely. Depending on the situation, we might recommend running the rebuild at reduced concurrency to lower scratch-space needs, scoping the rebuild to only the indexes that need it (`scope=QUADS` or `scope=DICTIONARIES`), or scheduling it during a lower-traffic window.

Typical workflow

Use the following workflow to assess and optimize your Neptune storage.

1. **Assess storage usage.** Call GetStorageInfo to understand your current storage distribution and look for large index sizes relative to entry counts, which indicate fragmentation. Note the current `VolumeBytesUsed` value as a baseline.

   AWS CLI

   ```
   aws neptunedata get-storage-info \
     --endpoint-url https://your-neptune-endpoint:port
   ```

   SDK

   ```
   response = client.get_storage_info()
   ```

   awscurl

   ```
   awscurl https://your-neptune-endpoint:port/storage/info \
     --region us-east-1 \
     --service neptune-db
   ```

   curl

   ```
   curl -G https://your-neptune-endpoint:port/storage/info
   ```

2. **Start an index rebuild.** If you identify fragmentation or bloat, initiate a rebuild and save the `rebuildId` from the response.

   AWS CLI

   ```
   aws neptunedata start-index-rebuild \
     --endpoint-url https://your-neptune-endpoint:port \
   ```

```
  --scope QUADS
```

SDK

```
response = client.start_index_rebuild(Scope='QUADS')
```

awscurl

```
awscurl "https://your-neptune-endpoint:port/storage/indexes/rebuilds?scope=QUADS"
 \
  --region us-east-1 \
  --service neptune-db \
  -X POST
```

curl

```
curl -X POST "https://your-neptune-endpoint:port/storage/indexes/rebuilds?
scope=QUADS"
```

3. **Monitor progress.** Poll the rebuild status periodically. The `progressPercentage` field tracks completion and `estimatedEndTime` estimates when the rebuild will finish.

   AWS CLI

```
aws neptunedata get-index-rebuild-status \
  --endpoint-url https://your-neptune-endpoint:port \
  --rebuild-id rebuildId
```

   SDK

```
response = client.get_index_rebuild_status(
    RebuildId='rebuildId'
)
```

   awscurl

```
awscurl https://your-neptune-endpoint:port/storage/indexes/rebuilds/rebuildId \
  --region us-east-1 \
  --service neptune-db
```

curl

```
curl -G https://your-neptune-endpoint:port/storage/indexes/rebuilds/rebuildId
```

4. **Verify results.** After the rebuild completes, call `GetStorageInfo` again to confirm that storage sizes decreased. You can also track the `VolumeBytesUsed` and `VolumeBytesLeftTotal` CloudWatch metrics over time.

   AWS CLI

   ```
   aws neptunedata get-storage-info \
     --endpoint-url https://your-neptune-endpoint:port
   ```

   SDK

   ```
   response = client.get_storage_info()
   ```

   awscurl

   ```
   awscurl https://your-neptune-endpoint:port/storage/info \
     --region us-east-1 \
     --service neptune-db
   ```

   curl

   ```
   curl -G https://your-neptune-endpoint:port/storage/info
   ```

Best practices

- **Implement retry logic.** During the rebuild, some queries can fail with a transient `CancelledByMaintenanceException`, which you should treat like a `ConcurrentModificationException`. Retry failed queries with exponential backoff.
- **Monitor progress.** Poll `GetIndexRebuildStatus` to track progress and estimated completion time.
- **Run GetStorageInfo before and after.** This lets you quantify the space reclaimed and track storage efficiency over time. You can also track the `VolumeBytesUsed` and `VolumeBytesLeftTotal` CloudWatch metrics during the rebuild.

- **Target the scope to the component that holds the reclaimable space.** Use GetStorageInfo to see whether your quad indexes or your dictionaries account for the most reclaimable space, and rebuild only that component with `scope=QUADS` or `scope=DICTIONARIES` for a faster, lower-impact rebuild. Use `scope=ALL` only when both need optimization.
- **Tune concurrency to balance speed against workload impact.** A higher `concurrency` finishes faster. However, it competes far more aggressively for buffer cache and I/O. This competition can noticeably slow your concurrent queries. Start with a low concurrency (the default is 1) when the rebuild runs alongside production traffic, and increase it only when you have spare instance capacity or are running in a maintenance window.
- **Consider the write buffer size.** If you have a heavy write workload during the rebuild, use `writeBufferSize=HIGH` so that all of the write workload can be tracked and applied to the rebuilt indexes.
- **Consider scaling up the writer instance for the duration of the rebuild.** Rebuilds run faster on larger writer instances because parallel workers scale with available vCPUs and a larger buffer pool reduces I/O when the graph exceeds the pool size on a smaller instance. If wall-clock time matters, temporarily use a larger writer during the rebuild and scale back afterward.
- **Plan for cancellation.** Cancellation is best-effort. When you call `CancelIndexRebuild`, work that is already in progress must finish before the rebuild reaches `CANCELLED` status. The time this takes can range from minutes to hours, depending on what is currently in flight.

Limitations

- **One rebuild at a time.** Only one index rebuild can run at a time for each cluster.
- **Transient query failures.** During brief internal commit phases of the rebuild, some queries can fail with a retriable `CancelledByMaintenanceException`. Implement retry logic with exponential backoff.
- **Rebuild history.** Currently, only the most recent rebuild is retained in memory. Persistent rebuild history is planned for a future release. If the engine restarts or crashes, the most recent rebuild record is lost.
- **100 TB limit.** If your cluster exceeds a `VolumeBytesUsed` of 100 TB, contact AWS Support to coordinate the index rebuild with the Neptune team.
- **Estimated end time is approximate.** The `estimatedEndTime` returned by `GetIndexRebuildStatus` is a best-effort estimate. Actual completion time can vary depending on data sizes, concurrent workload, and storage conditions. A writer instance with a larger buffer pool can complete the rebuild significantly faster.

- **Extremely heavy workload can fail the rebuild.** A very heavy concurrent workload (through read or write queries) can cause the rebuild to fail, either from heavy contention for hardware resources such as the buffer cache and I/O, or because the write buffer fills up. To resolve this, reduce your workload, or reduce the scope and concurrency of the rebuild and raise its `writeBufferSize` (up to the `HIGH` maximum). If the rebuild still fails at `writeBufferSize=HIGH`, reduce the concurrent write workload, or run the rebuild during a lower-traffic window.

Frequently asked questions

Does the index rebuild take my database offline?

No. During an online index rebuild, you can run concurrent read and write queries. Your database remains fully operational throughout the rebuild, regardless of the `scope` you specify.

Does the index rebuild support the OSGP and PZSOG indexes?

Yes. If you have the OSGP index and/or the PZSOG index (for geospatial data) enabled, the rebuild rebuilds them too when `scope` is `ALL` or `QUADS`.

How long does an index rebuild take?

The duration depends on the total size of your indexes and the volume of write operations during the rebuild, because those writes must be reapplied to the new indexes as well. You can monitor progress with `GetIndexRebuildStatus`, which provides `progressPercentage` and `estimatedEndTime`. Under heavy concurrent write activity, `estimatedEndTime` might not be accurate.

Will I lose data during the rebuild?

No. The online rebuild is designed to preserve all data. Concurrent writes during the rebuild are committed normally.

What happens if the rebuild fails?

Your database remains fully operational. `GetIndexRebuildStatus` shows `status=FAILED` with a `failureReason`. Common failure causes include:

- **Insufficient storage.** The volume cannot provide enough scratch space to run the rebuild within its safety margin. See Storage safety mechanisms.

- **Write buffer overflow.** A sustained very heavy write workload during the rebuild can exhaust the internal buffer that tracks changes. Increase `writeBufferSize` (up to `HIGH`) or reduce the concurrent write workload and retry.

After investigating the cause, you can start a new rebuild.

A writer restart or failover does not produce a `FAILED` status. In those cases, Neptune loses the in-progress rebuild and its record, so `GetIndexRebuildStatus` returns `ResourceNotFoundException` for that `rebuildId`. For details, see What happens if my engine restarts during a rebuild? and What happens if a failover occurs during the rebuild?

Can I cancel a rebuild?

Yes. Use the `CancelIndexRebuild` API. Cancellation is best-effort— in-flight work must drain before the rebuild reaches `CANCELLED`. Your database remains operational throughout.

What happens if my engine restarts during a rebuild?

The rebuild stops, and the engine restart discards its progress and `rebuildId`. Your database recovers to normal operation, and you can start a new rebuild.

What happens if a failover occurs during the rebuild?

If a failover switches the writer to a read replica during an active rebuild, the in-progress rebuild is lost. The new writer is healthy and fully operational, and you can start a new rebuild after the failover completes.

Can I run other Neptune APIs during the rebuild?

Yes for most APIs. `GetStorageInfo`, the query APIs (Gremlin, openCypher, SPARQL), and export continue to work. The bulk load API (`POST /loader`) and the fast reset API (`ExecuteFastReset`) are rejected during a rebuild.

How often should I run an index rebuild?

Consider running a rebuild when you observe significant storage bloat (through `GetStorageInfo`) or degraded query performance after heavy delete operations. Quarterly reviews are a reasonable starting point, but the optimal frequency depends on your workload patterns.

Does the rebuild affect my read replicas?

No. Read replicas are not affected by the rebuild and continue to serve read queries normally.

My cluster only receives read queries—should I clone it, run the rebuild on the clone, and switch to the clone?

Cloning is a valid alternative only if you can pause all writes to the original cluster for the duration of the rebuild, or the cluster has no writes at all. Neptune does not replicate to the clone any writes made to the original cluster after the clone point — those writes are lost when you switch to the clone as your new primary. For workloads with ongoing writes, run the rebuild online on the primary instead.

If you can pause writes, follow this approach:

1. Use the Neptune fast-clone feature to create a point-in-time copy of the cluster.
2. Run the rebuild on the clone at your chosen concurrency without affecting the live cluster.
3. Update your application to point at the clone's endpoint once the rebuild completes.

For more information about cluster cloning, see Database Cloning in Neptune.

Why does the rebuild take longer, or why do my queries slow down, when other workloads run at the same time?

The rebuild relies heavily on the buffer cache and I/O, so it competes for those resources with your other workloads. Monitor the `BufferCacheHitRatio` CloudWatch metric, which is the percentage of requests served by the buffer cache. If the cache hit ratio drops under 99.9%, consider upgrading the instance type to cache more data in memory, or reduce the concurrency of the rebuild to lessen competition with other workloads.

Neptune inlined server-generated edge ID

Neptune supports inline Server-Generated Edge IDs. It can be enabled via the Neptune configuration parameter `neptune_enable_inline_server_generated_edge_id` when `neptune_streams` is not enabled. This feature is available for Gremlin queries starting with engine release 1.4.3.0, and will be available for OpenCypher queries in a future release.

Edge ID is a unique identifier for an edge. An edge ID can be provided when inserting an edge. If no ID is provided, the server generates and assigns a UUID based ID to the edge by default. Like the user-defined ID, the UUID-based server-generated ID is stored in the dictionary.

When the `neptune_enable_inline_server_generated_edge_id` feature is enabled, the server generates a unique inlined ID when no ID is provided in the query. The inlined edge IDs are

not stored in the dictionary, improving the storage efficiency. The server-generated inlined IDs begin with the reserved prefix `neptune_reserved`.

> **Warning**
>
> Neptune reserves the `'neptune_reserved'` prefix for server generated inlined IDs. An error will be shown for queries attempting to insert data with a user-defined ID that begins with the reserved prefix.

The inlined server-generated edge ID feature can be enabled by setting the cluster-level parameter `neptune_enable_inline_server_generated_edge_id` to 1. A reboot of the instance is required. The following example enables the server-generated edge ID feature:

```
"ParameterName=neptune_enable_inline_server_generated_edge_id,ParameterValue=1,ApplyMethod=pend
reboot"
```

To verify if the feature is enabled, you can check the features in the engine status. This feature is automatically disabled if `neptune_streams` is enabled. The following example output shows the engine status for the enabled feature:

```
"features":{"InlineServerGeneratedEdgeId":"enabled"}
```

The following Gremlin example adds an edge without a user-defined ID when the inline server-generated edge ID feature is enabled:

```
curl - X POST--url https: //<neptune-cluster-endpoint>:8182/gremlin/ --data
 '{"gremlin":"g.withSideEffect(\"Neptune#disablePushdownOptimization\",
 true).addV().property(id, \"a\").addV().property(id, \"b\").addE(\"el\").to(V(\"a
\"))"}'
{
    "requestId": "b6b84605-53ad-4c04-baf1-7f0f31a3aeaf",
    "status": {
        "message": "",
        "code": 200,
        "attributes": {
            "@type": "g:Map",
            "@value": []
        }
    },
```

```
    "result": {
        "data": {
            "@type": "g:List",
            "@value": [{
                "@type": "g:Edge",
                "@value": {
                    "id": "neptune_reserved_231850767",
                    "label": "el",
                    "inVLabel": "vertex",
                    "outVLabel": "vertex",
                    "inV": "a",
                    "outV": "b"
                }
            }]
        },
        "meta": {
            "@type": "g:Map",
            "@value": []
        }
    }
}
```

Querying a Neptune Graph

Neptune supports the following graph query languages to access a graph:

- Gremlin, defined by Apache TinkerPop for creating and querying property graphs.

  A query in Gremlin is a traversal made up of discrete steps, each of which follows an edge to a node.

  See Accessing a Neptune graph with Gremlin to learn about using Gremlin in Neptune, and Gremlin standards compliance in Amazon Neptune to find specific details about the Neptune implementation of Gremlin.
- openCypher is a declarative query language for property graphs that was originally developed by Neo4j, then open-sourced in 2015, and contributed to the openCypher project under an Apache 2 open-source license. Its syntax is documented in the openCypher spec.
- SPARQL is a declarative language based on graph pattern-matching, for querying RDF data. It is supported by the World Wide Web Consortium.

  See Accessing the Neptune graph with SPARQL to learn about using SPARQL in Neptune, and SPARQL standards compliance in Amazon Neptune to find specific details about the Neptune implementation of SPARQL.

> **Note**
>
> Both Gremlin and openCypher can be used to query any property-graph data stored in Neptune, regardless of how it was loaded.

Topics

- Query queuing in Amazon Neptune
- Query plan cache in Amazon Neptune
- Inject a Custom ID Into a Neptune Gremlin or SPARQL Query
- Accessing a Neptune graph with Gremlin
- Accessing the Neptune Graph with openCypher
- Accessing the Neptune graph with SPARQL

Query queuing in Amazon Neptune

When developing and tuning graph applications, it can be helpful to know the implications of how queries are being queued by the database. In Amazon Neptune, query queuing occurs as follows:

- The maximum number of queries that can be queued up per instance, regardless of the instance size, is 8,192. Any queries over that number are rejected and fail with a `ThrottlingException`.
- The maximum number of queries that can be executing at one time is determined by the number of worker threads assigned, which is generally set to twice the number of virtual CPU cores (vCPUs) that are available.
- Query latency includes the time a query spends in the queue as well as network round-tripping and the time it actually takes to execute.

Determining how many queries are in your queue at a given moment

The `MainRequestQueuePendingRequests` CloudWatch metric records the the number of requests waiting in the input queue at a five-minutes granularity (see Neptune CloudWatch Metrics).

For Gremlin, you can obtain a current count of queries in the queue using the `acceptedQueryCount` value returned by the Gremlin query status API. Note, however, that the `acceptedQueryCount` value returned by the SPARQL query status API includes all queries accepted since the server was started, including completed queries.

How query queuing can affect timeouts

As noted above, query latency includes the time a query spends in the queue as well as the time it takes to execute.

Because a query's timeout period is generally measured starting from when it enters the queue, a slow-moving queue can make many queries time out as soon as they are dequeued. This is obviously undesirable, so it is good to avoid queuing up a large number of queries unless they can be executed rapidly.

Query plan cache in Amazon Neptune

When a query is submitted to Neptune, the query string is parsed, optimized, and transformed into a query plan, which then gets executed by the engine. Applications are often backed by

common query patterns that are instantiated with different values. Query plan cache can reduce the overall latency by caching the query plans and thereby avoiding parsing and optimization for such repeated patterns.

Query Plan Cache can be used for **OpenCypher** queries — both non-parameterized or parameterized queries. It is enabled for READ, and for HTTP and Bolt. It is **not** supported for OC mutation queries. It is **not** supported for Gremlin or SPARQL queries.

How to force enable or disable query plan cache

Query plan cache is enabled by default for low-latency parameterized queries. A plan for a parameterized query is cached only when latency is lower than the threshold of **100ms**. This behavior can be overridden on a per-query (parameterized or not) basis by the query-level Query Hint `QUERY:PLANCACHE`. It needs to be used with the `USING` clause. The query hint accepts `enabled` or `disabled` as a value.

AWS CLI

Forcing plan to be cached or reused:

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "Using QUERY:PLANCACHE \"enabled\" MATCH(n) RETURN n LIMIT 1"
```

With parameters:

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "Using QUERY:PLANCACHE \"enabled\" RETURN \$arg" \
  --parameters '{"arg": 123}'
```

Forcing plan to be neither cached nor reused:

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "Using QUERY:PLANCACHE \"disabled\" MATCH(n) RETURN n LIMIT 1"
```

For more information, see execute-open-cypher-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

# Forcing plan to be cached or reused
response = client.execute_open_cypher_query(
    openCypherQuery='Using QUERY:PLANCACHE "enabled" MATCH(n) RETURN n LIMIT 1'
)

print(response['results'])
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

Forcing plan to be cached or reused:

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=Using QUERY:PLANCACHE \"enabled\" MATCH(n) RETURN n LIMIT 1"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

Forcing plan to be cached or reused:

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=Using QUERY:PLANCACHE \"enabled\" MATCH(n) RETURN n LIMIT 1"
```

With parameters:

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=Using QUERY:PLANCACHE \"enabled\" RETURN \$arg" \
  -d "parameters={\"arg\": 123}"
```

Forcing plan to be neither cached nor reused:

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=Using QUERY:PLANCACHE \"disabled\" MATCH(n) RETURN n LIMIT 1"
```

How to determine if a plan is cached or not

For HTTP READ, if the query was submitted and the plan was cached, `explain` would show details relevant to query plan cache.

AWS CLI

```
aws neptunedata execute-open-cypher-explain-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "Using QUERY:PLANCACHE \"enabled\" MATCH(n) RETURN n LIMIT 1"
 \
  --explain-mode details
```

For more information, see execute-open-cypher-explain-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_explain_query(
    openCypherQuery='Using QUERY:PLANCACHE "enabled" MATCH(n) RETURN n LIMIT 1',
```

```
    explainMode='details'
)

print(response['results'].read().decode('utf-8'))
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=Using QUERY:PLANCACHE \"enabled\" MATCH(n) RETURN n LIMIT 1" \
  -d "explain=details"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=Using QUERY:PLANCACHE \"enabled\" MATCH(n) RETURN n LIMIT 1" \
  -d "explain=details"
```

If the plan was cached, the `explain` output shows:

```
Query: <QUERY STRING>
Plan cached by request: <REQUEST ID OF FIRST TIME EXECUTION>
Plan cached at: <TIMESTAMP OF FIRST TIME EXECUTION>
Parameters: <PARAMETERS, IF QUERY IS PARAMETERIZED QUERY>
Plan cache hits: <NUMBER OF CACHE HITS FOR CACHED PLAN>
First query evaluation time: <LATENCY OF FIRST TIME EXECUTION>

The query has been executed based on a cached query plan. Detailed explain with
 operator runtime statistics can be obtained by running the query with plan cache
 disabled (using HTTP parameter planCache=disabled).
```

When using Bolt, the explain feature is not supported.

Eviction

A query plan is evicted by the cache time to live (TTL) or when a maximum number of cached query plans have been reached. When the query plan is hit, the TTL is refreshed. The defaults are:

- 1000 - The maximum number of plans that can be cached per instance.
- TTL - 300,000 milliseconds or 5 minutes. The cache hit restarts the TTL, and resets it back to 5 min.

Conditions causing the plan not to be cached

Query plan cache would not be used under the following conditions:

1. When a query is submitted using the query hint `QUERY:PLANCACHE "disabled"`. You can re-run the query and remove `QUERY:PLANCACHE "disabled"` to enable the query plan cache.
2. If the query that was submitted is not a parameterized query and does not contain the hint `QUERY:PLANCACHE "enabled"`.
3. If the query evaluation time is larger than the latency threshold, the query is not cached and is considered a long-running query that would not benefit from the query plan cache.
4. If the query contains a pattern that doesn't return any results.
   - i.e. `MATCH (n:nonexistentLabel) return n` when there are zero nodes with the specified label.
   - i.e. `MATCH (n {name: $param}) return n` with `parameters={"param": "abcde"}` when there are zero nodes containing `name=abcde`.
5. If the query parameter is a composite type, such as a `list` or a `map`.

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=Using QUERY:PLANCACHE \"enabled\" RETURN \$arg" \
  -d "parameters={\"arg\": [1, 2, 3]}"

curl https://your-neptune-endpoint:port/openCypher \
  -d "query=Using QUERY:PLANCACHE \"enabled\" RETURN \$arg" \
  -d "parameters={\"arg\": {\"a\": 1}}"
```

6. If the query parameter is a string that has not been part of a data load or data insertion operation. For example, if `CREATE (n {name: "X"})` is ran to insert `"X"`, then `RETURN "X"`

is cached, while `RETURN "Y"` would not be cached, as `"Y"` has not been inserted and does not exist in the database.

Inject a Custom ID Into a Neptune Gremlin or SPARQL Query

By default, Neptune assigns a unique `queryId` value to every query. You can use this ID to get information about a running query (see Gremlin query status API or SPARQL query status API), or cancel it (see Gremlin query cancellation or SPARQL query cancellation).

Neptune also lets you specify your own `queryId` value for a Gremlin or SPARQL query, either in the HTTP header, or for a SPARQL query by using the `queryId` query hint. Assigning your own `queryID` makes it easy to keep track of a query so as to get status or cancel it.

Injecting a Custom `queryId` Value Using the HTTP Header

For both Gremlin and SPARQL, the HTTP header can be used to inject your own `queryId` value into a query.

Gremlin Example

```
curl -XPOST https://your-neptune-endpoint:port \
    -d "{\"gremlin\": \
        \"g.V().limit(1).count()\" , \
        \"queryId\":\"4d5c4fae-aa30-41cf-9e1f-91e6b7dd6f47\"  }"
```

SPARQL Example

```
curl https://your-neptune-endpoint:port/sparql \
    -d "query=SELECT * WHERE { ?s ?p ?o } " \
       --data-urlencode \
       "queryId=4d5c4fae-aa30-41cf-9e1f-91e6b7dd6f47"
```

Injecting a Custom `queryId` Value Using a SPARQL Query Hint

Here is an example of how you would use the SPARQL `queryId` query hint to inject a custom `queryId` value into a SPARQL query:

```
curl https://your-neptune-endpoint:port/sparql \
    -d "query=PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#> \
```

```
      SELECT * WHERE { hint:Query hint:queryId \"4d5c4fae-
aa30-41cf-9e1f-91e6b7dd6f47\" \
      {?s ?p ?o}}"
```

Using the queryId Value to Check Query Status

Gremlin Example

```
curl https://your-neptune-endpoint:port/gremlin/status \
    -d "queryId=4d5c4fae-aa30-41cf-9e1f-91e6b7dd6f47"
```

SPARQL Example

```
curl https://your-neptune-endpoint:port/sparql/status \
    -d "queryId=4d5c4fae-aa30-41cf-9e1f-91e6b7dd6f47"
```

Accessing a Neptune graph with Gremlin

Amazon Neptune is compatible with Apache TinkerPop and Gremlin. This means that you can connect to a Neptune DB instance and use the Gremlin traversal language to query the graph (see The Graph in the Apache TinkerPop documentation). For differences in the Neptune implementation of Gremlin, see Gremlin standards compliance.

A *traversal* in Gremlin is a series of chained steps. It starts at a vertex (or edge). It walks the graph by following the outgoing edges of each vertex and then the outgoing edges of those vertices. Each step is an operation in the traversal. For more information, see The Traversal in the TinkerPop documentation.

Different Neptune engine versions support different Gremlin versions. Check the engine release page of the Neptune version you are running to determine which Gremlin release it supports or consult the following table which lists the earliest and latest versions of TinkerPop supported by different Neptune engine versions:

Neptune Engine Version	Minimum TinkerPop Version	Maximum TinkerPop Version
1.5.0.0 and newer	3.8.0	3.8.1
1.3.2.0 <= 1.4.x	3.7.1	3.7.7

Neptune Engine Version	Minimum TinkerPop Version	Maximum TinkerPop Version
1.3.1.0	3.6.2	3.6.5
1.3.0.0	3.6.2	3.6.4
1.2.1.0 <= 1.2.1.2	3.6.2	3.6.2
1.1.1.0 <= 1.2.0.2	3.5.5	3.5.6
1.1.0.0 and older	(deprecated)	(deprecated)

TinkerPop clients are usually backwards compatible within a series (`3.6.x`, for example, or `3.7.x`) and while they can often work across those boundaries, the table above recommends the version combinations to use for the best possible experience and compatibility. Unless otherwise advised, it is generally best to adhere to these guidelines and upgrade client applications to match the version of TinkerPop you are using.

When upgrading TinkerPop versions it is always important to refer to TinkerPop's upgrade documentation which will help you identify new features you can take advantage of, but also issues you may need to be aware of as you approach your upgrade. You should typically expect existing queries and features to work after upgrade unless something in particular is called out as an issue to consider. Finally, it is important to note that should a version you upgrade to have a new feature, you may not be able to use it if it is from a version later than what Neptune supports.

There are Gremlin language variants and support for Gremlin access in various programming languages. For more information, see On Gremlin Language Variants in the TinkerPop documentation.

This documentation describes how to access Neptune with the following variants and programming languages:

- Set up the Gremlin console to connect to a Neptune DB instance
- Using the HTTPS REST endpoint to connect to a Neptune DB instance
- Java-based Gremlin clients to use with Amazon Neptune
- Using Python to connect to a Neptune DB instance
- Using .NET to connect to a Neptune DB instance
- Using Node.js to connect to a Neptune DB instance

- Using Go to connect to a Neptune DB instance

As discussed in Encrypting connections to your Amazon Neptune database with SSL/HTTPS, you must use Transport Layer Security/Secure Sockets Layer (TLS/SSL) when connecting to Neptune in all AWS Regions.

Before you begin, you must have the following:

- A Neptune DB instance. For information about creating a Neptune DB instance, see Creating an Amazon Neptune cluster.
- An Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB instance.

For more information about loading data into Neptune, including prerequisites, loading formats, and load parameters, see Loading data into Amazon Neptune.

Topics

- Set up the Gremlin console to connect to a Neptune DB instance
- Using the HTTPS REST endpoint to connect to a Neptune DB instance
- Java-based Gremlin clients to use with Amazon Neptune
- Using Python to connect to a Neptune DB instance
- Using .NET to connect to a Neptune DB instance
- Using Node.js to connect to a Neptune DB instance
- Using Go to connect to a Neptune DB instance
- Using the AWS SDK to run Gremlin queries
- Gremlin query hints
- Gremlin query status API
- Gremlin query cancellation
- Support for Gremlin script-based sessions
- Gremlin transactions in Neptune
- Streaming query results with Gremlin
- Using the Gremlin API with Amazon Neptune
- Caching query results in Amazon Neptune Gremlin
- Making efficient upserts with Gremlin mergeV() and mergeE() steps

- Making efficient Gremlin upserts with fold()/coalesce()/unfold()
- Analyzing Neptune query execution using Gremlin explain
- Using Gremlin with the Amazon Neptune DFE query engine

Set up the Gremlin console to connect to a Neptune DB instance

The Gremlin Console allows you to experiment with TinkerPop graphs and queries in a REPL (read-eval-print loop) environment.

Installing the Gremlin console and connecting to it in the usual way

You can use the Gremlin Console to connect to a remote graph database. The following section walks you through installing and configuring the Gremlin Console to connect remotely to a Neptune DB instance. You must follow these instructions from an Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB instance.

For help connecting to Neptune with SSL/TLS (which is required), see SSL/TLS configuration.

> **Note**
>
> If you have IAM authentication enabled on your Neptune DB cluster, follow the instructions in Connecting to Amazon Neptune databases using IAM authentication with Gremlin console to install the Gremlin console rather than the instructions here.

To install the Gremlin Console and connect to Neptune

1. The Gremlin Console binaries for TinkerPop 3.8.x require Java 11 or later. These instructions assume Java 11. To install Java 11 on your EC2 instance, do the following:

   - If you're using Amazon Linux 2 (AL2):

     ```
     sudo amazon-linux-extras install java-openjdk11
     ```

   - If you're using Amazon Linux 2023 (AL2023):

     ```
     sudo yum install java-11-amazon-corretto-devel
     ```

   - For other distributions, use whichever of the following is appropriate:

```
sudo yum install java-11-openjdk-devel
```

or:

```
sudo apt-get install openjdk-11-jdk
```

2. Enter the following to set Java 11 as the default runtime on your EC2 instance.

```
sudo /usr/sbin/alternatives --config java
```

When prompted, enter the number for Java 11.

3. Download the appropriate version of the Gremlin Console from the Apache web site. You can check the Accessing a Neptune graph with Gremlin to determine which Gremlin version your version of Neptune supports. For example, if you need version 3.8.0, you can download the Gremlin console from the Apache Tinkerpop website onto your EC2 instance like this:

```
wget https://archive.apache.org/dist/tinkerpop/3.8.0/apache-tinkerpop-gremlin-
console-3.8.0-bin.zip
```

4. Unzip the Gremlin Console zip file.

```
unzip apache-tinkerpop-gremlin-console-3.8.0-bin.zip
```

5. Change directories into the unzipped directory.

```
cd apache-tinkerpop-gremlin-console-3.8.0
```

6. In the `conf` subdirectory of the extracted directory, create a file named `neptune-remote.yaml` with the following text. Replace *`your-neptune-endpoint`* with the hostname or IP address of your Neptune DB instance. The square brackets (`[ ]`) are required.

> **Note**
>
> For information about finding the hostname of your Neptune DB instance, see the Connecting to Amazon Neptune Endpoints section.

```
hosts: [your-neptune-endpoint]
port: 8182
connectionPool: { enableSsl: true }
serializer: { className:
 org.apache.tinkerpop.gremlin.util.ser.GraphBinaryMessageSerializerV1,
              config: { serializeResultToString: true }}
```

> **Note**
>
> Serializers were moved from the `gremlin-driver` module to the new `gremlin-util` module in version 3.7.0. The package changed from org.apache.tinkerpop.gremlin.driver.ser to org.apache.tinkerpop.gremlin.util.ser.

7. In a terminal, navigate to the Gremlin Console directory (`apache-tinkerpop-gremlin-console-3.8.0`), and then enter the following command to run the Gremlin Console.

```
bin/gremlin.sh
```

You should see the following output:

```
         \,,,/
         (o o)
-----oOOo-(3)-oOOo-----
plugin activated: tinkerpop.server
plugin activated: tinkerpop.utilities
plugin activated: tinkerpop.tinkergraph
gremlin>
```

You are now at the `gremlin>` prompt. You will enter the remaining steps at this prompt.

8. At the `gremlin>` prompt, enter the following to connect to the Neptune DB instance.

```
:remote connect tinkerpop.server conf/neptune-remote.yaml
```

9. At the `gremlin>` prompt, enter the following to switch to remote mode. This sends all Gremlin queries to the remote connection.

```
:remote console
```

10. Enter the following to send a query to the Gremlin Graph.

```
g.V().limit(1)
```

11. When you are finished, enter the following to exit the Gremlin Console.

```
:exit
```

> **Note**
>
> Use a semicolon (`;`) or a newline character (\n) to separate each statement.
> Each traversal preceding the final traversal must end in `next()` to be executed. Only the data from the final traversal is returned.

For more information on the Neptune implementation of Gremlin, see the section called "Gremlin standards compliance".

An alternate way to connect to the Gremlin console

Drawbacks of the normal connection approach

The most common way to connect to the Gremlin console is the one explained above, using commands like this at the `gremlin>` prompt:

```
gremlin> :remote connect tinkerpop.server conf/(file name).yaml
gremlin> :remote console
```

This works well, and lets you send queries to Neptune. However, it takes the Groovy script engine out of the loop, so Neptune treats all queries as pure Gremlin. This means that the following query forms fail:

```
gremlin> 1 + 1
gremlin> x = g.V().count()
```

The closest you can get to using a variable when connected this way is to use the `result` variable maintained by the console and send the query using `:>`, like this:

```
gremlin> :remote console
```

```
==>All scripts will now be evaluated locally - type ':remote console' to return
 to remote mode for Gremlin Server - [krl-1-cluster.cluster-ro-cm9t6tfwbtsr.us-
east-1.neptune.amazonaws.com/172.31.19.217:8182]
gremlin> :> g.V().count()
==>4249

gremlin> println(result)
[result{object=4249 class=java.lang.Long}]

gremlin> println(result['object'])
[4249]
```

A different way to connect

You can also connect to the Gremlin console in a different way, which you may find nicer, like this:

```
gremlin> g = traversal().withRemote('conf/neptune.properties')
```

Here `neptune.properties` takes this form:

```
gremlin.remote.remoteConnectionClass=org.apache.tinkerpop.gremlin.driver.remote.DriverRemoteCon
gremlin.remote.driver.clusterFile=conf/my-cluster.yaml
gremlin.remote.driver.sourceName=g
```

The `my-cluster.yaml` file should look like this:

```
hosts: [my-cluster-abcdefghijk.us-east-1.neptune.amazonaws.com]
port: 8182
serializer: { className:
 org.apache.tinkerpop.gremlin.util.ser.GraphBinaryMessageSerializerV1,
              config: { serializeResultToString: false } }
connectionPool: { enableSsl: true }
```

> **Note**
>
> Serializers were moved from the `gremlin-driver` module to the new `gremlin-util` module in version 3.7.0. The package changed from org.apache.tinkerpop.gremlin.driver.ser to org.apache.tinkerpop.gremlin.util.ser.

Configuring the Gremlin console connection like that lets you make the following kinds of queries successfully:

```
gremlin> 1+1
==>2

gremlin> x=g.V().count().next()
==>4249

gremlin> println("The answer was ${x}")
The answer was 4249
```

You can avoid displaying the result, like this:

```
gremlin> x=g.V().count().next();[]
gremlin> println(x)
4249
```

All the usual ways of querying (without the terminal step) continue to work. For example:

```
gremlin> g.V().count()
==>4249
```

You can even use the `g.io().read()` step to load a file with this kind of connection.

IAM authentication

Neptune supports IAM authentication to control access to your DB cluster. If you have IAM authentication enabled, you need to use Signature Version 4 signing to authenticate your requests. For detailed instructions and code examples for connecting from the Gremlin console, see Connecting to Amazon Neptune databases using IAM authentication with Gremlin console.

Using the HTTPS REST endpoint to connect to a Neptune DB instance

Amazon Neptune provides an HTTPS endpoint for Gremlin queries. The REST interface is compatible with whatever Gremlin version your DB cluster is using (see the engine release page of the Neptune engine version you are running to determine which Gremlin release it supports).

> **Note**
>
> As discussed in Encrypting connections to your Amazon Neptune database with SSL/HTTPS, Neptune now requires that you connect using HTTPS instead of HTTP. In addition, Neptune does not currently support HTTP/2 for REST API requests. Clients must use HTTP/1.1 when connecting to endpoints.

The following instructions walk you through connecting to the Gremlin endpoint using the `curl` command and HTTPS. You must follow these instructions from an Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB instance.

The HTTPS endpoint for Gremlin queries to a Neptune DB instance is `https://`*`your-neptune-endpoint`*`:`*`port`*`/gremlin`.

> **Note**
>
> For information about finding the hostname of your Neptune DB instance, see Connecting to Amazon Neptune Endpoints.

To connect to Neptune using the HTTP REST endpoint

The following examples show how to submit a Gremlin query to the REST endpoint. You can use the AWS SDK, the AWS CLI, or **curl**.

AWS CLI

```
aws neptunedata execute-gremlin-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --gremlin-query "g.V().limit(1)"
```

For more information, see execute-gremlin-query in the AWS CLI Command Reference.

SDK

```
import boto3
import json
from botocore.config import Config
```

```
client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_gremlin_query(
    gremlinQuery='g.V().limit(1)',
    serializer='application/vnd.gremlin-v3.0+json;types=false'
)

print(json.dumps(response['result'], indent=2))
```

For AWS SDK examples in other languages like Java, .NET, and more, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/gremlin \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d '{"gremlin":"g.V().limit(1)"}'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

The following example uses **curl** to submit a Gremlin query through HTTP **POST**. The query is submitted in JSON format in the body of the post as the `gremlin` property.

```
curl -X POST -d '{"gremlin":"g.V().limit(1)"}' https://your-neptune-endpoint:port/
gremlin
```

Although HTTP **POST** requests are recommended for sending Gremlin queries, it is also possible to use HTTP **GET** requests:

```
curl -G "https://your-neptune-endpoint:port?gremlin=g.V().count()"
```

These examples return the first vertex in the graph by using the `g.V().limit(1)` traversal. You can query for something else by replacing it with another Gremlin traversal.

> **Important**
>
> By default, the REST endpoint returns all results in a single JSON result set. If this result set is too large, an `OutOfMemoryError` exception can occur on the Neptune DB instance. You can avoid this by enabling chunked responses (results returned in a series of separate responses). See Use optional HTTP trailing headers to enable multi-part Gremlin responses.

> **Note**
>
> Neptune does not support the `bindings` property.

Use optional HTTP trailing headers to enable multi-part Gremlin responses

By default, the HTTP response to Gremlin queries is returned in a single JSON result set. In the case of a very large result set, this can cause an `OutOfMemoryError` exception on the DB instance.

However, you can enable *chunked* responses (responses that are returned in multiple separate parts). You do this by including a transfer-encoding (TE) trailers header (`te: trailers`) in your request. See the MDN page about TE request headers) for more information about TE headers.

When a response is returned in multiple parts, it can be hard to diagnose a problem that occurs after the first part is received, since the first part arrives with an HTTP status code of `200` (OK). A subsequent failure usually results in a message body containing a corrupt response, at the end of which Neptune appends an error message.

To make detection and diagnosis of this kind of failure easier, Neptune also includes two new header fields within the trailing headers of every response chunk:

- `X-Neptune-Status` – contains the response code followed by a short name. For instance, in case of success the trailing header would be: `X-Neptune-Status: 200 OK`. In the case of

failure, the response code would be one of the Neptune engine error code, such as `X-Neptune-Status: 500 TimeLimitExceededException`.

- `X-Neptune-Detail –` is empty for successful requests. In the case of errors, it contains the JSON error message. Because only ASCII characters are allowed in HTTP header values, the JSON string is URL encoded.

> **Note**
>
> Neptune does not currently support `gzip` compression of chunked responses. If the client requests both chunked encoding and compression at the same time, Neptune skips the compression.

Java-based Gremlin clients to use with Amazon Neptune

You can use either of two open-source Java-based Gremlin clients with Amazon Neptune: the Apache TinkerPop Java Gremlin client, or the Gremlin client for Amazon Neptune.

Apache TinkerPop Java Gremlin client

The Apache TinkerPop Java gremlin-driver is the standard, official Gremlin client that works with any TinkerPop-enabled graph database. Use this client when you need maximum compatibility with the broader TinkerPop development space, when you're working with multiple graph database systems, or when you don't require the advanced cluster management and load balancing features specific to Neptune. This client is also suitable for simple applications that connect to a single Neptune instance or when you prefer to handle load balancing at the infrastructure level rather than within the client.

> **Important**
>
> Choosing the correct Apache TinkerPop Gremlin driver version is critical for compatibility with your Neptune engine version. Using an incompatible version can result in connection failures or unexpected behavior. For detailed version compatibility information, see Accessing a Neptune graph with Gremlin.

> **Note**
>
> The table that helps you determine the correct Apache TinkerPop version to use with Neptune has been moved to the section called "Gremlin". This table was previously located on this page for many years and is now more centralized for reference for all programming languages that TinkerPop supports.

Gremlin Java client for Amazon Neptune

The Gremlin client for Amazon Neptune is an open-source Java-based Gremlin client that acts as a drop-in replacement for the standard TinkerPop Java client.

The Neptune Gremlin client is optimized for Neptune clusters. It lets you manage traffic distribution across multiple instances in a cluster, and adapts to changes in cluster topology when you add or remove a replica. You can even configure the client to distribute requests across a subset of instances in your cluster, based on role, instance type, availability zone (AZ), or tags associated with instances.

The latest version of the Neptune Gremlin Java client is available on Maven Central.

For more information about the Neptune Gremlin Java client, see this blog post. For code samples and demos, check out the client's GitHub project.

When choosing the version of the Neptune Gremlin client, you need to consider the underlying TinkerPop version in relation to your Neptune engine version. Refer to the compatibility table at the section called "Gremlin" to determine the correct TinkerPop version for your Neptune engine, then use the following table to select the appropriate Neptune Gremlin client version:

Neptune Gremlin client version compatibility

Neptune Gremlin client version	TinkerPop version
3.x	3.7.x (AWS SDK for Java 2.x/1.x)
2.1.x	3.7.x (AWS SDK for Java 1.x)
2.0.x	3.6.x

Neptune Gremlin client version	TinkerPop version
1.12	3.5.x

Using a Java client to connect to a Neptune DB instance

The following section walks you through the running of a complete Java sample that connects to a Neptune DB instance and performs a Gremlin traversal using the Apache TinkerPop Gremlin client.

These instructions must be followed from an Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB instance.

To connect to Neptune using Java

1. Install Apache Maven on your EC2 instance. If using Amazon Linux 2023 (preferred), use:

   ```
   sudo dnf update -y
   sudo dnf install maven -y
   ```

   If using Amazon Linux 2, download the latest binary from https://maven.apache.org/download.cgi:

   ```
   sudo yum remove maven -y
   wget https://dlcdn.apache.org/maven/maven-3/ <version>/binaries/apache-maven-
   <version>-bin.tar.gz
   sudo tar -xzf apache-maven-<version>-bin.tar.gz -C /opt/
   sudo ln -sf /opt/apache-maven-<version> /opt/maven
   echo 'export MAVEN_HOME=/opt/maven' >> ~/.bashrc
   echo 'export PATH=$MAVEN_HOME/bin:$PATH' >> ~/.bashrc
   source ~/.bashrc
   ```

2. **Install Java.** The Gremlin libraries for TinkerPop 3.8.x require Java 11 or later. To install Java 11, do the following:

   - If you're using Amazon Linux 2 (AL2):

     ```
     sudo amazon-linux-extras install java-openjdk11
     ```

   - If you're using Amazon Linux 2023 (AL2023):

```
sudo yum install java-11-amazon-corretto-devel
```

- For other distributions, use whichever of the following is appropriate:

```
sudo yum install java-11-openjdk-devel
```

or:

```
sudo apt-get install openjdk-11-jdk
```

3. **Set Java 11 as the default runtime on your EC2 instance:** Enter the following to set Java 11 as the default runtime on your EC2 instance:

```
sudo /usr/sbin/alternatives --config java
```

When prompted, enter the number for Java 11.

4. **Create a new directory named `gremlinjava`:**

```
mkdir gremlinjava
cd gremlinjava
```

5. In the `gremlinjava` directory, create a `pom.xml` file, and then open it in a text editor:

```
nano pom.xml
```

6. Copy the following into the `pom.xml` file and save it:

```
<project xmlns="https://maven.apache.org/POM/4.0.0"
         xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://
maven.apache.org/maven-v4_0_0.xsd">
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.amazonaws</groupId>
  <artifactId>GremlinExample</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>GremlinExample</name>
```

```
  <url>https://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>org.apache.tinkerpop</groupId>
      <artifactId>gremlin-driver</artifactId>
      <version>3.8.0</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-jdk14</artifactId>
      <version>1.7.25</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <source>11</source>
          <target>11</target>
        </configuration>
      </plugin>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>exec-maven-plugin</artifactId>
          <version>1.3</version>
          <configuration>
            <executable>java</executable>
            <arguments>
              <argument>-classpath</argument>
              <classpath/>
              <argument>com.amazonaws.App</argument>
            </arguments>
            <mainClass>com.amazonaws.App</mainClass>
            <complianceLevel>1.11</complianceLevel>
            <killAfter>-1</killAfter>
          </configuration>
        </plugin>
    </plugins>
  </build>
</project>
```

> **Note**
>
> If you are modifying an existing Maven project, the required dependency is highlighted in the preceding code.

7. Create subdirectories for the example source code (`src/main/java/com/amazonaws/`) by typing the following at the command line:

```
mkdir -p src/main/java/com/amazonaws/
```

8. In the `src/main/java/com/amazonaws/` directory, create a file named `App.java`, and then open it in a text editor.

```
nano src/main/java/com/amazonaws/App.java
```

9. Copy the following into the `App.java` file. Replace *`your-neptune-endpoint`* with the address of your Neptune DB instance. Do *not* include the `https://` prefix in the `addContactPoint` method.

> **Note**
>
> For information about finding the hostname of your Neptune DB instance, see Connecting to Amazon Neptune Endpoints.

```
package com.amazonaws;
import org.apache.tinkerpop.gremlin.driver.Cluster;
import
 org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversalSource;
import org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversal;
import static
 org.apache.tinkerpop.gremlin.process.traversal.AnonymousTraversalSource.traversal;
import org.apache.tinkerpop.gremlin.driver.remote.DriverRemoteConnection;
import org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.__;
import org.apache.tinkerpop.gremlin.structure.T;

public class App
{
  public static void main( String[] args )
```

```
  {
    Cluster.Builder builder = Cluster.build();
    builder.addContactPoint("your-neptune-endpoint");
    builder.port(8182);
    builder.enableSsl(true);

    Cluster cluster = builder.create();

    GraphTraversalSource g =
 traversal().withRemote(DriverRemoteConnection.using(cluster));

    // Add a vertex.
    // Note that a Gremlin terminal step, e.g. iterate(), is required to make a
 request to the remote server.
    // The full list of Gremlin terminal steps is at https://tinkerpop.apache.org/
docs/current/reference/#terminal-steps
    g.addV("Person").property("Name", "Justin").iterate();

    // Add a vertex with a user-supplied ID.
    g.addV("Custom Label").property(T.id, "CustomId1").property("name", "Custom id
 vertex 1").iterate();
    g.addV("Custom Label").property(T.id, "CustomId2").property("name", "Custom id
 vertex 2").iterate();

    g.addE("Edge Label").from(__.V("CustomId1")).to(__.V("CustomId2")).iterate();

    // This gets the vertices, only.
    GraphTraversal t = g.V().limit(3).elementMap();

    t.forEachRemaining(
      e ->  System.out.println(t.toList())
    );

    cluster.close();
  }
}
```

For help connecting to Neptune with SSL/TLS (which is required), see SSL/TLS configuration.

10. Compile and run the sample using the following Maven command:

```
mvn compile exec:exec
```

The preceding example returns a map of the key and values of each property for the first two vertexes in the graph by using the `g.V().limit(3).elementMap()` traversal. To query for something else, replace it with another Gremlin traversal with one of the appropriate ending methods.

> **Note**
>
> The final part of the Gremlin query, `.toList()`, is required to submit the traversal to the server for evaluation. If you don't include that method or another equivalent method, the query is not submitted to the Neptune DB instance.
> You also must append an appropriate ending when you add a vertex or edge, such as when you use the `addV( )` step.

The following are examples of methods that submit the query to the Neptune DB instance:

- `toList()`
- `toSet()`
- `next()`
- `iterate()`

Use `iterate()` when you don't need the results of your queries (e.g. mutations) as it saves serialization costs.

These terminal steps behave differently depending on whether you submit queries as strings (script mode) or as bytecode traversal objects (bytecode mode). The differences are especially important for transaction handling. For details, see Test Gremlin code in the context where you will deploy it.

SSL/TLS configuration for Gremlin Java client

Neptune requires SSL/TLS to be enabled by default. Typically, if the Java driver is configured with `enableSsl(true)`, it can connect to Neptune without having to set up a `trustStore()` or `keyStore()` with a local copy of a certificate.

However, if the instance with which you are connecting doesn't have an internet connection through which to verify a public certificate, or if the certificate you're using isn't public, you can take the following steps to configure a local certificate copy:

Setting up a local certificate copy to enable SSL/TLS

1. Download and install keytool from Oracle. This will make setting up the local key store much easier.
2. Download the `SFSRootCAG2.pem`CA certificate (the Gremlin Java SDK requires a certificateto verify the remote certificate):

```
wget https://www.amazontrust.com/repository/SFSRootCAG2.pem
```

3. Create a key store in either JKS or PKCS12 format. This example uses JKS. Answer the questions that follow at the prompt. The password that you create here will be needed later:

```
keytool -genkey -alias (host name) -keyalg RSA -keystore server.jks
```

4. Import the `SFSRootCAG2.pem` file that you downloaded into the newly created key store:

```
keytool -import -keystore server.jks -file .pem
```

5. Configure the `Cluster` object programmatically:

```
Cluster cluster = Cluster.build("(your neptune endpoint)")
                         .port(8182)
                         .enableSSL(true)
                         .keyStore('server.jks')
                         .keyStorePassword("(the password from step 2)")
                         .create();
```

You can do the same thing in a configuration file if you want, as you might do with the Gremlin console:

```
hosts: [(your neptune endpoint)]
port: 8182
connectionPool: { enableSsl: true, keyStore: server.jks, keyStorePassword: (the
 password from step 2) }
serializer: { className:
 org.apache.tinkerpop.gremlin.util.ser.GraphBinaryMessageSerializerV1, config:
 { serializeResultToString: true }}
```

IAM authentication

Neptune supports IAM authentication to control access to your DB cluster. If you have IAM authentication enabled, you need to use Signature Version 4 signing to authenticate your requests. For detailed instructions and code examples for connecting from a Java client, see Connecting to Amazon Neptune databases using IAM with Gremlin Java.

Java example of connecting to a Neptune DB instance with re-connect logic

The following Java example demonstrates how to connect to the Gremlin client with reconnect logic to recover from an unexpected disconnect.

For detailed guidance on developing a practical retry strategy, including exponential backoff with jitter, see the section called "Exceptions and Retries".

It has the following dependencies:

```
<dependency>
    <groupId>org.apache.tinkerpop</groupId>
    <artifactId>gremlin-driver</artifactId>
    <version>${gremlin.version}</version>
</dependency>

<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>amazon-neptune-sigv4-signer</artifactId>
    <version>${sig4.signer.version}</version>
</dependency>

<dependency>
    <groupId>com.evanlennick</groupId>
    <artifactId>retry4j</artifactId>
    <version>0.15.0</version>
</dependency>
```

Here is the sample code:

> **Important**
>
> The `CallExecutor` from Retry4j may not be thread-safe. Consider having each thread use its own `CallExecutor` instance, or use a different retrying library.

> **Note**
>
> The following example has been updated to include the use of requestInterceptor(). This was added in TinkerPop 3.6.6. Prior to TinkerPop version 3.6.6, the code example used handshakeInterceptor(), which was deprecated with that release.

```
public static void main(String args[]) {
  boolean useIam = true;

  // Create Gremlin cluster and traversal source
  Cluster.Builder builder = Cluster.build()
         .addContactPoint(System.getenv("neptuneEndpoint"))
         .port(Integer.parseInt(System.getenv("neptunePort")))
         .enableSsl(true)
         .minConnectionPoolSize(1)
         .maxConnectionPoolSize(1)
         .serializer(Serializers.GRAPHBINARY_V1D0)
         .reconnectInterval(2000);

  if (useIam) {
      builder.requestInterceptor( r -> {
         try {
            NeptuneNettyHttpSigV4Signer sigV4Signer =
                        new NeptuneNettyHttpSigV4Signer("(your region)", new
 DefaultAWSCredentialsProviderChain());
            sigV4Signer.signRequest(r);
         } catch (NeptuneSigV4SignerException e) {
            throw new RuntimeException("Exception occurred while signing the request",
 e);
         }
         return r;
      });
   }

  Cluster cluster = builder.create();

  GraphTraversalSource g = AnonymousTraversalSource
      .traversal()
      .withRemote(DriverRemoteConnection.using(cluster));

  // Configure retries
```

```
  // NOTE: This example uses a fixed backoff for simplicity.
  // In a production application consider use of exponential backoff with jitter.
  RetryConfig retryConfig = new RetryConfigBuilder()
      .retryOnCustomExceptionLogic(getRetryLogic())
      .withDelayBetweenTries(1000, ChronoUnit.MILLIS)
      .withMaxNumberOfTries(5)
      .withFixedBackoff()
      .build();

  @SuppressWarnings("unchecked")
  CallExecutor<Object> retryExecutor = new CallExecutorBuilder<Object>()
      .config(retryConfig)
      .build();

  // Do lots of queries
  for (int i = 0; i < 100; i++){
    String id = String.valueOf(i);

    @SuppressWarnings("unchecked")
    Callable<Object> query = () -> g.mergeV(Map.of(T.id, id))
        .option(Merge.onCreate, Map.of(T.label, "Person"))
        .id().next();

    // Retry query
    // If there are connection failures, the Java Gremlin client will automatically
    // attempt to reconnect in the background, so all we have to do is wait and retry.
    Status<Object> status = retryExecutor.execute(query);

    System.out.println(status.getResult().toString());
  }

  cluster.close();
}

private static Function<Exception, Boolean> getRetryLogic() {

  return e -> {

    Class<? extends Exception> exceptionClass = e.getClass();

    StringWriter stringWriter = new StringWriter();
    e.printStackTrace(new PrintWriter(stringWriter));
    String message = stringWriter.toString();
```

```
    if (RemoteConnectionException.class.isAssignableFrom(exceptionClass)){
      System.out.println("Retrying because RemoteConnectionException");
      return true;
    }

    // Check for connection issues
    if (message.contains("Timed out while waiting for an available host") ||
        message.contains("Timed-out") && message.contains("waiting for connection on
 Host") ||
        message.contains("Connection to server is no longer active") ||
        message.contains("Connection reset by peer") ||
        message.contains("SSLEngine closed already") ||
        message.contains("Pool is shutdown") ||
        message.contains("ExtendedClosedChannelException") ||
        message.contains("Broken pipe") ||
        message.contains(System.getenv("neptuneEndpoint")))
    {
      System.out.println("Retrying because connection issue");
      return true;
    };

    // Concurrent writes can sometimes trigger a ConcurrentModificationException.
    // In these circumstances you may want to backoff and retry.
    if (message.contains("ConcurrentModificationException")) {
      System.out.println("Retrying because ConcurrentModificationException");
      return true;
    }

    // If the primary fails over to a new instance, existing connections to the old
 primary will
    // throw a ReadOnlyViolationException. You may want to back and retry.
    if (message.contains("ReadOnlyViolationException")) {
      System.out.println("Retrying because ReadOnlyViolationException");
      return true;
    }

    System.out.println("Not a retriable error");
    return false;
  };
}
```

Using Python to connect to a Neptune DB instance

> **Important**
>
> Choosing the correct Apache TinkerPop Gremlin driver version is critical for compatibility with your Neptune engine version. Using an incompatible version can result in connection failures or unexpected behavior. For detailed version compatibility information, see Accessing a Neptune graph with Gremlin.

The following section walks you through the running of a Python sample that connects to an Amazon Neptune DB instance and performs a Gremlin traversal.

You must follow these instructions from an Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB instance.

Before you begin, do the following:

- Download and install Python 3.10 or later from the Python.org website. TinkerPop 3.8.x requires Python 3.10 or later.
- Verify that you have **pip** installed. If you don't have **pip** or you're not sure, see Do I need to install pip? in the **pip** documentation.
- If your Python installation does not already have it, download `futures` as follows: `pip install futures`

To connect to Neptune using Python

1. Enter the following to install the `gremlinpython` package:

   ```
   pip install --user gremlinpython
   ```

2. Create a file named `gremlinexample.py`, and then open it in a text editor.
3. Copy the following into the `gremlinexample.py` file. Replace *`your-neptune-endpoint`* with the address of your Neptune DB cluster and *`your-neptune-port`* with the port of your Neptune DB cluster (default:8182).

For information about finding the address of your Neptune DB instance, see the Connecting to Amazon Neptune Endpoints section.

The example below demonstrates how to connect with Gremlin Python.

```
import boto3
import os
from botocore.auth import SigV4Auth
from botocore.awsrequest import AWSRequest
from gremlin_python.driver.driver_remote_connection import DriverRemoteConnection
from gremlin_python.process.anonymous_traversal import traversal

database_url = "wss://your-neptune-endpoint:your-neptune-port/gremlin"

remoteConn = DriverRemoteConnection(database_url, "g")

g = traversal().withRemote(remoteConn)

print(g.inject(1).toList())
remoteConn.close()
```

4. Enter the following command to run the sample:

```
python gremlinexample.py
```

The Gremlin query at the end of this example returns the vertices (`g.V().limit(2)`) in a list. This list is then printed with the standard Python `print` function.

> **Note**
>
> The final part of the Gremlin query, `toList()`, is required to submit the traversal to the server for evaluation. If you don't include that method or another equivalent method, the query is not submitted to the Neptune DB instance.

The following are examples of methods that submit the query to the Neptune DB instance:

- `toList()`
- `toSet()`

- `next()`
- `iterate()`

These terminal steps behave differently in script mode and bytecode mode. For the canonical list of terminal steps and details about how they affect transactions, see Test Gremlin code in the context where you will deploy it.

Use `iterate()` when you don't need the results of your queries (e.g. mutations) as it saves serialization costs.

The preceding example returns the first two vertices in the graph by using the `g.V().limit(2).toList()` traversal. To query for something else, replace it with another Gremlin traversal with one of the appropriate ending methods.

IAM authentication

Neptune supports IAM authentication to control access to your DB cluster. If you have IAM authentication enabled, you need to use Signature Version 4 signing to authenticate your requests. For detailed instructions and code examples for connecting from a Python client, see Connecting to Amazon Neptune databases using IAM authentication with Gremlin Python.

Using .NET to connect to a Neptune DB instance

> **Important**
>
> Choosing the correct Apache TinkerPop Gremlin driver version is critical for compatibility with your Neptune engine version. Using an incompatible version can result in connection failures or unexpected behavior. For detailed version compatibility information, see Accessing a Neptune graph with Gremlin.

The following section contains a code example written in C# that connects to a Neptune DB instance and performs a Gremlin traversal.

Connections to Amazon Neptune must be from an Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB instance. This sample code was tested on an Amazon EC2 instance running Ubuntu.

Before you begin, do the following:

- Install .NET on the Amazon EC2 instance. To get instructions for installing .NET on multiple operating systems, including Windows, Linux, and macOS, see Get Started with .NET.
- Install Gremlin.NET by running `dotnet add package gremlin.net` for your package. For more information, see Gremlin.NET in the TinkerPop documentation.

To connect to Neptune using Gremlin.NET

1. Create a new .NET project.

```
dotnet new console -o gremlinExample
```

2. Change directories into the new project directory.

```
cd gremlinExample
```

3. Copy the following into the `Program.cs` file. Replace *`your-neptune-endpoint`* with the address of your Neptune DB instance.

   For information about finding the address of your Neptune DB instance, see the Connecting to Amazon Neptune Endpoints section.

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Gremlin.Net;
using Gremlin.Net.Driver;
using Gremlin.Net.Driver.Remote;
using Gremlin.Net.Structure;
using static Gremlin.Net.Process.Traversal.AnonymousTraversalSource;
namespace gremlinExample
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        var endpoint = "your-neptune-endpoint";
```

```
        // This uses the default Neptune and Gremlin port, 8182
        var gremlinServer = new GremlinServer(endpoint, 8182, enableSsl: true );
        var gremlinClient = new GremlinClient(gremlinServer);
        var remoteConnection = new DriverRemoteConnection(gremlinClient, "g");
        var g = Traversal().WithRemote(remoteConnection);
        g.AddV("Person").Property("Name", "Justin").Iterate();
        g.AddV("Custom Label").Property("name", "Custom id vertex 1").Iterate();
        g.AddV("Custom Label").Property("name", "Custom id vertex 2").Iterate();
        var output = g.V().Limit<Vertex>(3).ToList();
        foreach(var item in output) {
            Console.WriteLine(item);
        }
      }
      catch (Exception e)
      {
          Console.WriteLine("{0}", e);
      }
    }
  }
}
```

4. Enter the following command to run the sample:

```
dotnet run
```

The Gremlin query at the end of this example returns the count of a single vertex for testing purposes. It is then printed to the console.

> **Note**
>
> The final part of the Gremlin query, `Next()`, is required to submit the traversal to the server for evaluation. If you don't include that method or another equivalent method, the query is not submitted to the Neptune DB instance.

The following are examples of methods that submit the query to the Neptune DB instance:

- `ToList()`
- `ToSet()`
- `Next()`

- `Iterate()`

These terminal steps behave differently in script mode and bytecode mode. For the canonical list of terminal steps and details about how they affect transactions, see Test Gremlin code in the context where you will deploy it.

Use `Iterate()` when you don't need the results of your queries (e.g. mutations) as it saves serialization costs.

The preceding example returns a list by using the `g.V().Limit(3).ToList()` traversal. To query for something else, replace it with another Gremlin traversal with one of the appropriate ending methods.

IAM authentication

Neptune supports IAM authentication to control access to your DB cluster. If you have IAM authentication enabled, you need to use Signature Version 4 signing to authenticate your requests. For detailed instructions and code examples for connecting from a .NET client, see Connecting to Amazon Neptune databases using IAM authentication with Gremlin .NET.

Using Node.js to connect to a Neptune DB instance

> **Important**
>
> Choosing the correct Apache TinkerPop Gremlin driver version is critical for compatibility with your Neptune engine version. Using an incompatible version can result in connection failures or unexpected behavior. For detailed version compatibility information, see Accessing a Neptune graph with Gremlin.

The following section walks you through the running of a Node.js sample that connects to an Amazon Neptune DB instance and performs a Gremlin traversal.

You must follow these instructions from an Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB instance.

Before you begin, do the following:

- Verify that Node.js version 8.11 or higher is installed. If it is not, download and install Node.js from the Nodejs.org website.

To connect to Neptune using Node.js

1. Enter the following to install the `gremlin-javascript` package:

```
npm install gremlin
```

2. Create a file named `gremlinexample.js` and open it in a text editor.
3. Copy the following into the `gremlinexample.js` file. Replace *`your-neptune-endpoint`* with the address of your Neptune DB instance.

   For information about finding the address of your Neptune DB instance, see the Connecting to Amazon Neptune Endpoints section.

```
const gremlin = require('gremlin');
const DriverRemoteConnection = gremlin.driver.DriverRemoteConnection;
const Graph = gremlin.structure.Graph;

dc = new DriverRemoteConnection('wss://your-neptune-endpoint:8182/gremlin',{});

const graph = new Graph();
const g = graph.traversal().withRemote(dc);

g.V().limit(1).count().next().
    then(data => {
        console.log(data);
        dc.close();
    }).catch(error => {
        console.log('ERROR', error);
        dc.close();
    });
```

4. Enter the following command to run the sample:

```
node gremlinexample.js
```

The preceding example returns the count of a single vertex in the graph by using the `g.V().limit(1).count().next()` traversal. To query for something else, replace it with another Gremlin traversal with one of the appropriate ending methods.

> **Note**
>
> The final part of the Gremlin query, `next()`, is required to submit the traversal to the server for evaluation. If you don't include that method or another equivalent method, the query is not submitted to the Neptune DB instance.

The following are examples of methods that submit the query to the Neptune DB instance:

- `toList()`
- `toSet()`
- `next()`
- `iterate()`

These terminal steps behave differently in script mode and bytecode mode. For the canonical list of terminal steps and details about how they affect transactions, see Test Gremlin code in the context where you will deploy it.

Use `iterate()` when you don't need the results of your queries (e.g. mutations) as it saves serialization costs.

> **Important**
>
> This is a standalone Node.js example. If you are planning to run code like this in an AWS Lambda function, see Lambda function examples for details about using JavaScript efficiently in a Neptune Lambda function.

IAM authentication

Neptune supports IAM authentication to control access to your DB cluster. If you have IAM authentication enabled, you need to use Signature Version 4 signing to authenticate your requests. For detailed instructions and code examples for connecting from a JavaScript client, see Connecting to Amazon Neptune databases using IAM authentication with Gremlin JavaScript.

Using Go to connect to a Neptune DB instance

> **Important**
>
> Choosing the correct Apache TinkerPop Gremlin driver version is critical for compatibility with your Neptune engine version. Using an incompatible version can result in connection failures or unexpected behavior. For detailed version compatibility information, see Accessing a Neptune graph with Gremlin.

> **Note**
>
> The gremlingo 3.5.x versions are backwards compatible with TinkerPop 3.4.x versions as long as you only use 3.4.x features in the Gremlin queries you write.

The following section walks you through the running of a Go sample that connects to an Amazon Neptune DB instance and performs a Gremlin traversal.

You must follow these instructions from an Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB instance.

Before you begin, do the following:

- Download and install Go 1.25 or later from the go.dev website. TinkerPop 3.8.x requires Go 1.25 or later.

To connect to Neptune using Go

1. Starting from an empty directory, initialize a new Go module:

```
go mod init example.com/gremlinExample
```

2. Add gremlin-go as a dependency of your new module:

```
go get github.com/apache/tinkerpop/gremlin-go/v3/driver
```

3. Create a file named `gremlinExample.go` and then open it in a text editor.

4. Copy the following into the `gremlinExample.go` file, replacing *(your neptune endpoint)* with the address of your Neptune DB instance:

```
package main

import (
  "fmt"
  gremlingo "github.com/apache/tinkerpop/gremlin-go/v3/driver"
)

func main() {
  // Creating the connection to the server.
  driverRemoteConnection, err := gremlingo.NewDriverRemoteConnection("wss://(your
 neptune endpoint):8182/gremlin",
    func(settings *gremlingo.DriverRemoteConnectionSettings) {
      settings.TraversalSource = "g"
    })
  if err != nil {
    fmt.Println(err)
    return
  }
  // Cleanup
  defer driverRemoteConnection.Close()

  // Creating graph traversal
  g := gremlingo.Traversal_().WithRemote(driverRemoteConnection)

  // Perform traversal
  results, err := g.V().Limit(2).ToList()
  if err != nil {
    fmt.Println(err)
    return
  }
  // Print results
  for _, r := range results {
    fmt.Println(r.GetString())
  }
}
```

> **Note**
>
> The Neptune TLS certificate format is not currently supported on Go 1.18+ with macOS, and may give a 509 error when trying to initiate a connection. For local testing, this can be skipped by adding "crypto/tls" to the imports and modifying the `DriverRemoteConnection` settings as follows:
>
> ```
> // Creating the connection to the server.
> driverRemoteConnection, err := gremlingo.NewDriverRemoteConnection("wss://
> your-neptune-endpoint:8182/gremlin",
>  func(settings *gremlingo.DriverRemoteConnectionSettings) {
>  settings.TraversalSource = "g"
>  settings.TlsConfig = &tls.Config{InsecureSkipVerify: true}
>  })
> ```

5. Enter the following command to run the sample:

```
go run gremlinExample.go
```

The Gremlin query at the end of this example returns the vertices (`g.V().Limit(2)`) in a slice. This slice is then iterated through and printed with the standard `fmt.Println` function.

> **Note**
>
> The final part of the Gremlin query, `ToList()`, is required to submit the traversal to the server for evaluation. If you don't include that method or another equivalent method, the query is not submitted to the Neptune DB instance.

The following are examples of methods that submit the query to the Neptune DB instance:

- `ToList()`
- `ToSet()`
- `Next()`
- `Iterate()`

These terminal steps behave differently in script mode and bytecode mode. For the canonical list of terminal steps and details about how they affect transactions, see Test Gremlin code in the context where you will deploy it.

Use `Iterate()` when you don't need the results of your queries (e.g. mutations) as it saves serialization costs.

The preceding example returns the first two vertices in the graph by using the `g.V().Limit(2).ToList()` traversal. To query for something else, replace it with another Gremlin traversal with one of the appropriate ending methods.

IAM authentication

Neptune supports IAM authentication to control access to your DB cluster. If you have IAM authentication enabled, you need to use Signature Version 4 signing to authenticate your requests. For detailed instructions and code examples for connecting from a Go client, see Connecting to Amazon Neptune databases using IAM authentication with Gremlin Go.

Using the AWS SDK to run Gremlin queries

With the AWS SDK, you can run Gremlin queries against your Neptune graph using a programming language of your choice. The Neptune data API SDK (service name `neptunedata`) provides the ExecuteGremlinQuery action for submitting Gremlin queries.

You must run these examples from an Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB cluster, or from a location that has network connectivity to your cluster endpoint.

Direct links to the API reference documentation for the `neptunedata` service in each SDK language can be found below:

Programming language	neptunedata API reference
C++	https://sdk.amazonaws.com/cpp/api/LATEST/aws-cpp-sdk-neptunedata/html/annotated.html
Go	https://docs.aws.amazon.com/sdk-for-go/api/service/neptunedata/

Programming language	neptunedata API reference
Java	https://sdk.amazonaws.com/java/api/latest/software/amazon/awssdk/services/neptunedata/package-summary.html
JavaScript	https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/Package/-aws-sdk-client-neptunedata/
Kotlin	https://sdk.amazonaws.com/kotlin/api/latest/neptunedata/index.html
.NET	https://docs.aws.amazon.com/sdkfornet/v3/apidocs/items/Neptunedata/NNeptunedata.html
PHP	https://docs.aws.amazon.com/aws-sdk-php/v3/api/namespace-Aws.Neptunedata.html
Python	https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/neptunedata.html
Ruby	https://docs.aws.amazon.com/sdk-for-ruby/v3/api/Aws/Neptunedata.html
Rust	https://crates.io/crates/aws-sdk-neptunedata
CLI	https://docs.aws.amazon.com/cli/latest/reference/neptunedata/

Gremlin AWS SDK examples

The following examples show how to set up a `neptunedata` client, run a Gremlin query, and print the results. Replace *`YOUR_NEPTUNE_HOST`* and *`YOUR_NEPTUNE_PORT`* with the endpoint and port of your Neptune DB cluster.

Client-side timeout and retry configuration

The SDK client timeout controls how long the *client* waits for a response. It does not control how long the query runs on the server. If the client times out before the server finishes, the query may continue running on Neptune while the client has no way to retrieve the results. We recommend setting the client-side read timeout to `0` (no timeout) or to a value that is at least a few seconds longer than the server-side neptune_query_timeout setting on your Neptune DB cluster. This lets Neptune control when queries time out.
We also recommend setting the maximum retry attempts to 1 (no retries). If the SDK retries a query that is still running on the server, it can result in duplicate operations. This is especially important for mutation queries, where a retry could cause unintended duplicate writes.

Python

1. Follow the installation instructions to install Boto3.
2. Create a file named `gremlinExample.py` and paste the following code:

```
import boto3
import json
from botocore.config import Config

# Disable the client-side read timeout and retries so that
# Neptune's server-side neptune_query_timeout controls query duration.
client = boto3.client(
    'neptunedata',
    endpoint_url=f'https://YOUR_NEPTUNE_HOST:YOUR_NEPTUNE_PORT',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

# Use the untyped GraphSON v3 serializer for a cleaner JSON response.
response = client.execute_gremlin_query(
    gremlinQuery='g.V().limit(1)',
    serializer='application/vnd.gremlin-v3.0+json;types=false'
)

print(json.dumps(response['result'], indent=2))
```

3. Run the example: `python gremlinExample.py`

Java

1. Follow the installation instructions to set up the AWS SDK for Java.
2. Use the following code to set up a `NeptunedataClient`, run a Gremlin query, and print the result:

```
import java.net.URI;
import java.time.Duration;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.core.retry.RetryPolicy;
import software.amazon.awssdk.services.neptunedata.NeptunedataClient;
import
 software.amazon.awssdk.services.neptunedata.model.ExecuteGremlinQueryRequest;
import
 software.amazon.awssdk.services.neptunedata.model.ExecuteGremlinQueryResponse;

// Disable the client-side timeout and retries so that
// Neptune's server-side neptune_query_timeout controls query duration.
NeptunedataClient client = NeptunedataClient.builder()
    .endpointOverride(URI.create("https://YOUR_NEPTUNE_HOST:YOUR_NEPTUNE_PORT"))
    .overrideConfiguration(ClientOverrideConfiguration.builder()
        .apiCallTimeout(Duration.ZERO)
        .retryPolicy(RetryPolicy.none())
        .build())
    .build();

// Use the untyped GraphSON v3 serializer for a cleaner JSON response.
ExecuteGremlinQueryRequest request = ExecuteGremlinQueryRequest.builder()
    .gremlinQuery("g.V().limit(1)")
    .serializer("application/vnd.gremlin-v3.0+json;types=false")
    .build();

ExecuteGremlinQueryResponse response = client.executeGremlinQuery(request);

System.out.println(response.result().toString());
```

JavaScript

1. Follow the installation instructions to set up the AWS SDK for JavaScript. Install the neptunedata client package: `npm install @aws-sdk/client-neptunedata`.
2. Create a file named `gremlinExample.js` and paste the following code:

```
import { NeptunedataClient, ExecuteGremlinQueryCommand } from "@aws-sdk/client-
neptunedata";
import { NodeHttpHandler } from "@smithy/node-http-handler";

const config = {
    endpoint: "https://YOUR_NEPTUNE_HOST:YOUR_NEPTUNE_PORT",
    // Disable the client-side request timeout so that
    // Neptune's server-side neptune_query_timeout controls query duration.
    requestHandler: new NodeHttpHandler({
        requestTimeout: 0
    }),
    maxAttempts: 1
};

const client = new NeptunedataClient(config);

// Use the untyped GraphSON v3 serializer for a cleaner JSON response.
const input = {
    gremlinQuery: "g.V().limit(1)",
    serializer: "application/vnd.gremlin-v3.0+json;types=false"
};

const command = new ExecuteGremlinQueryCommand(input);
const response = await client.send(command);

console.log(JSON.stringify(response, null, 2));
```

3. Run the example: `node gremlinExample.js`

Gremlin query hints

You can use query hints to specify optimization and evaluation strategies for a particular Gremlin query in Amazon Neptune.

Most query hints are specified by adding a `withSideEffect` step to the query with the following syntax.

```
g.withSideEffect(hint, value)
```

- *hint* – Identifies the type of the hint to apply.

- *value* – Determines the behavior of the system aspect under consideration.

For example, the following shows how to include a `repeatMode` hint in a Gremlin traversal.

> **Note**
>
> Query hints that use `withSideEffect` are always prefixed with `Neptune#`.

```
g.withSideEffect('Neptune#repeatMode',
 'DFS').V("3").repeat(out()).times(10).limit(1).path()
```

The preceding query instructs the Neptune engine to traverse the graph *Depth First* (DFS) rather than the default Neptune, *Breadth First* (BFS).

> **Important**
>
> Some query hints use the `with` step instead of `withSideEffect`. The two are not interchangeable. Each query hint section on this page shows the correct syntax to use.

The following sections provide more information about the available query hints and their usage.

Topics

- Gremlin repeatMode query hint
- Gremlin noReordering query hint
- Gremlin typePromotion query hint
- Gremlin useDFE query hint
- Gremlin evaluationTimeoutBehavior query hint
- Gremlin query hints for using the results cache

Gremlin repeatMode query hint

The Neptune `repeatMode` query hint specifies how the Neptune engine evaluates the `repeat()` step in a Gremlin traversal: breadth first, depth first, or chunked depth first.

The evaluation mode of the `repeat()` step is important when it is used to find or follow a path, rather than simply repeating a step a limited number of times.

Syntax

The `repeatMode` query hint is specified by adding a `withSideEffect` step to the query.

```
g.withSideEffect('Neptune#repeatMode', 'mode').gremlin-traversal
```

> **Note**
>
> All Gremlin query hints side effects are prefixed with `Neptune#`.

Available Modes

- `BFS`

  Breadth-First Search

  Default execution mode for the `repeat()` step. This gets all sibling nodes before going deeper along the path.

  This version is memory-intensive and frontiers can get very large. There is a higher risk that the query will run out of memory and be cancelled by the Neptune engine. This most closely matches other Gremlin implementations.
- `DFS`

  Depth-First Search

  Follows each path to the maximum depth before moving on to the next solution.

  This uses less memory. It may provide better performance in situations like finding a single path from a starting point out multiple hops.
- `CHUNKED_DFS`

  Chunked Depth-First Search

  A hybrid approach that explores the graph depth-first in chunks of 1,000 nodes, rather than 1 node (`DFS`) or all nodes (`BFS`).

The Neptune engine will get up to 1,000 nodes at each level before following the path deeper.

This is a balanced approach between speed and memory usage.

It is also useful if you want to use BFS, but the query is using too much memory.

Example

The following section describes the effect of the repeat mode on a Gremlin traversal.

In Neptune the default mode for the `repeat()` step is to perform a breadth-first (BFS) execution strategy for all traversals.

In most cases, the TinkerGraph implementation uses the same execution strategy, but in some cases it alters the execution of a traversal.

For example, the TinkerGraph implementation modifies the following query.

```
g.V("3").repeat(out()).times(10).limit(1).path()
```

The `repeat()` step in this traversal is "unrolled" into the following traversal, which results in a depth-first (DFS) strategy.

```
g.V(<id>).out().out().out().out().out().out().out().out().out().out().limit(1).path()
```

> **Important**
>
> The Neptune query engine does not do this automatically.

Breadth-first (BFS) is the default execution strategy, and is similar to TinkerGraph in most cases. However, there are certain cases where depth-first (DFS) strategies are preferable.

BFS (Default)

Breadth-first (BFS) is the default execution strategy for the `repeat()` operator.

```
g.V("3").repeat(out()).times(10).limit(1).path()
```

The Neptune engine fully explores the first nine-hop frontiers before finding a solution ten hops out. This is effective in many cases, such as a shortest-path query.

However, for the preceding example, the traversal would be much faster using the depth-first (DFS) mode for the `repeat()` operator.

DFS

The following query uses the depth-first (DFS) mode for the `repeat()` operator.

```
g.withSideEffect("Neptune#repeatMode", "DFS").V("3").repeat(out()).times(10).limit(1)
```

This follows each individual solution out to the maximum depth before exploring the next solution.

Gremlin noReordering query hint

When you submit a Gremlin traversal, the Neptune query engine investigates the structure of the traversal and reorders parts of the query, trying to minimize the amount of work required for evaluation and query response time. For example, a traversal with multiple constraints, such as multiple `has()` steps, is typically not evaluated in the given order. Instead it is reordered after the query is checked with static analysis.

The Neptune query engine tries to identify which constraint is more selective and runs that one first. This often results in better performance, but the order in which Neptune chooses to evaluate the query might not always be optimal.

If you know the exact characteristics of the data and want to manually dictate the order of the query execution, you can use the Neptune `noReordering` query hint to specify that the traversal be evaluated in the order given.

Syntax

The `noReordering` query hint is specified by adding a `withSideEffect` step to the query.

```
g.withSideEffect('Neptune#noReordering', true or false).gremlin-traversal
```

> **Note**
>
> All Gremlin query hints side effects are prefixed with `Neptune#`.

Available Values

- `true`
- `false`

Gremlin typePromotion query hint

When you submit a Gremlin traversal that filters on a numerical value or range, the Neptune query engine must normally use type promotion when it executes the query. This means that it has to examine values of every type that could hold the value you are filtering on.

For example, if you are filtering for values equal to 55, the engine must look for integers equal to 55, long integers equal to 55L, floats equal to 55.0, and so forth. Each type promotion requires an additional lookup on storage, which can cause an apparently simple query to take an unexpectedly long time to complete.

Let's say you are searching for all vertexes with a customer-age property greater than 5:

```
g.V().has('customerAge', gt(5))
```

To execute that traversal thoroughly, Neptune must expand the query to examine every numeric type that the value you are querying for could be promoted to. In this case, the `gt` filter has to be applied for any integer over 5, any long over 5L, any float over 5.0, and any double over 5.0. Because each of these type promotions requires an additional lookup on storage, you will see multiple filters per numeric filter when you run the Gremlin `profile` API for this query, and it will take significantly longer to complete than you might expect.

Often type promotion is unnecessary because you know in advance that you only need to find values of one specific type. When this is the case, you can speed up your queries dramatically by using the `typePromotion` query hint to turn off type promotion.

Syntax

The `typePromotion` query hint is specified by adding a `withSideEffect` step to the query.

```
g.withSideEffect('Neptune#typePromotion', true or false).gremlin-traversal
```

> **Note**
>
> All Gremlin query hints side effects are prefixed with `Neptune#`.

Available Values

- `true`
- `false`

To turn off type promotion for the query above, you would use:

```
g.withSideEffect('Neptune#typePromotion', false).V().has('customerAge', gt(5))
```

Gremlin useDFE query hint

Use this query hint to enable use of the DFE for executing the query. By default Neptune does not use the DFE without this query hint being set to `true`, because the neptune_dfe_query_engine instance parameter defaults to `viaQueryHint`. If you set that instance parameter to `enabled`, the DFE engine is used for all queries except those having the `useDFE` query hint set to `false`.

Example of enabling the DFE for a query:

```
g.withSideEffect('Neptune#useDFE', true).V().out()
```

Gremlin `evaluationTimeoutBehavior` query hint

By default, when a Gremlin query exceeds the query timeout, Neptune cancels the query and returns an error. The `evaluationTimeoutBehavior` query hint changes this behavior so that Neptune streams back any partial results that were computed before the timeout occurred, instead of returning an error.

> **Note**
>
> This query hint uses the `with` step, not `withSideEffect`, and does not use the `Neptune#` prefix.

Syntax

```
g.with('evaluationTimeoutBehavior', 'value').gremlin-traversal
```

Available Values

- `partialResults` – When the query times out, Neptune returns any results that were computed before the timeout.
- `exception` – When the query times out, Neptune returns an error. This is the default behavior.

> **Important**
>
> This query hint can only be used with read-only queries. If you use this hint with a mutating query or with DFE enabled (either via the `useDFE` query hint or the `neptune_dfe_query_engine` instance parameter), Neptune returns an error.

Example

The following query returns partial results if it exceeds the query timeout:

```
g.with('evaluationTimeoutBehavior', 'partialResults').V().out().valueMap()
```

Gremlin query hints for using the results cache

The following query hints can be used when the query results cache is enabled.

Gremlin `enableResultCache` query hint

The `enableResultCache` query hint with a value of `true` causes query results to be returned from the cache if they have already been cached. If not, it returns new results and caches them until such time as they are cleared from the cache. For example:

```
g.with('Neptune#enableResultCache', true)
 .V().has('genre','drama').in('likes')
```

Later, you can access the cached results by issuing exactly the same query again.

If the value of this query hint is `false`, or if it isn't present, query results are not cached. However, setting it to `false` does not clear existing cached results. To clear cached results, use the `invalidateResultCache` or `invalidateResultCachekey` hint.

Gremlin `enableResultCacheWithTTL` query hint

The `enableResultCacheWithTTL` query hint also returns cached results if there are any, without affecting the TTL of results already in the cache. If there are currently no cached results, the query returns new results and caches them for the time to live (TTL) specified by the `enableResultCacheWithTTL` query hint. That time to live is specified in seconds. For example, the following query specifies a time to live of sixty seconds:

```
g.with('Neptune#enableResultCacheWithTTL', 60)
 .V().has('genre','drama').in('likes')
```

Before the 60-second time-to-live is over, you can use the same query (here, `g.V().has('genre','drama').in('likes')`) with either the `enableResultCache` or the `enableResultCacheWithTTL` query hint to access the cached results.

> **Note**
>
> The time to live specified with `enableResultCacheWithTTL` does not affect results that have already been cached.
>
> - If results were previously cached using `enableResultCache`, the cache must first be explicitly cleared before `enableResultCacheWithTTL` generates new results and caches them for the TTL that it specifies.
> - If results were previously cached using `enableResultCachewithTTL`, that previous TTL must first expire before `enableResultCacheWithTTL` generates new results and caches them for the TTL that it specifies.

After the time to live has passed, the cached results for the query are cleared, and a subsequent instance of the same query then returns new results. If `enableResultCacheWithTTL` is attached to that subsequent query, the new results are cached with the TTL that it specifies.

Gremlin `invalidateResultCacheKey` query hint

The `invalidateResultCacheKey` query hint can take a `true` or `false` value. A `true` value causes cached results for the the query to which `invalidateResultCacheKey` is attached to be cleared. For example, the following example causes results cached for the query key `g.V().has('genre','drama').in('likes')` to be cleared:

```
g.with('Neptune#invalidateResultCacheKey', true)
 .V().has('genre','drama').in('likes')
```

The example query above does not cause its new results to be cached. You can include `enableResultCache` (or `enableResultCacheWithTTL`) in the same query if you want to cache the new results after clearing the existing cached ones:

```
g.with('Neptune#enableResultCache', true)
 .with('Neptune#invalidateResultCacheKey', true)
 .V().has('genre','drama').in('likes')
```

Gremlin `invalidateResultCache` query hint

The `invalidateResultCache` query hint can take a `true` or `false` value. A `true` value causes all results in the results cache to be cleared. For example:

```
g.with('Neptune#invalidateResultCache', true)
 .V().has('genre','drama').in('likes')
```

The example query above does not cause its results to be cached. You can include `enableResultCache` (or `enableResultCacheWithTTL`) in the same query if you want to cache new results after completely clearing the existing cache:

```
g.with('Neptune#enableResultCache', true)
 .with('Neptune#invalidateResultCache', true)
 .V().has('genre','drama').in('likes')
```

Gremlin `numResultsCached` query hint

The `numResultsCached` query hint can only be used with queries that contain `iterate()`, and it specifies the maximum number of results to cache for the query to which it is attached. Note that the results cached when `numResultsCached` is present are not returned, only cached.

For example, the following query specifies that up to 100 of its results should be cached, but none of those cached results returned:

```
g.with('Neptune#enableResultCache', true)
 .with('Neptune#numResultsCached', 100)
 .V().has('genre','drama').in('likes').iterate()
```

You can then use a query like the following to retrieve a range of the cached results (here, the first ten):

```
g.with('Neptune#enableResultCache', true)
 .with('Neptune#numResultsCached', 100)
 .V().has('genre','drama').in('likes').range(0, 10)
```

Gremlin `noCacheExceptions` query hint

The `noCacheExceptions` query hint can take a `true` or `false` value. A `true` value causes any exceptions related to the results cache to be suppressed. For example:

```
g.with('Neptune#enableResultCache', true)
 .with('Neptune#noCacheExceptions', true)
 .V().has('genre','drama').in('likes')
```

In particular, this suppresses the `QueryLimitExceededException`, which is raised if the results of a query are too large to fit in the results cache.

Gremlin query status API

You can list all active Gremlin queries or get the status of a specific query. The underlying HTTP endpoint for both operations is `https://`*`your-neptune-endpoint`*`:`*`port`*`/gremlin/status`.

Listing active Gremlin queries

To list all active Gremlin queries, call the endpoint with no `queryId` parameter.

Request parameters

- **includeWaiting** (*optional*) – If set to TRUE, the response includes waiting queries in addition to running queries.

Response syntax

```
{
  "acceptedQueryCount": integer,
  "runningQueryCount": integer,
  "queries": [
    {
      "queryId": "guid",
      "queryEvalStats": {
        "waited": integer,
        "elapsed": integer,
        "cancelled": boolean
      },
      "queryString": "string"
    }
  ]
}
```

- **acceptedQueryCount** – The number of queries that have been accepted but not yet completed, including queries in the queue.
- **runningQueryCount** – The number of currently running Gremlin queries.
- **queries** – A list of the current Gremlin queries.

Example

AWS CLI

```
aws neptunedata list-gremlin-queries \
  --endpoint-url https://your-neptune-endpoint:port
```

For more information, see list-gremlin-queries in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
```

```
)

response = client.list_gremlin_queries()

print(response)
```

For AWS SDK examples in other languages like Java, .NET, and more, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/gremlin/status \
  --region us-east-1 \
  --service neptune-db
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

```
curl https://your-neptune-endpoint:port/gremlin/status
```

The following output shows a single running query.

```
{
  "acceptedQueryCount": 9,
  "runningQueryCount": 1,
  "queries": [
    {
      "queryId": "fb34cd3e-f37c-4d12-9cf2-03bb741bf54f",
      "queryEvalStats": {
        "waited": 0,
        "elapsed": 23,
        "cancelled": false
      },
      "queryString": "g.V().out().count()"
```

```
    }
  ]
}
```

Getting the status of a specific Gremlin query

To get the status of a specific Gremlin query, provide the `queryId` parameter.

Request parameters

- **queryId** (*required*) – The ID of the Gremlin query. Neptune automatically assigns this ID value to each query, or you can assign your own ID (see Inject a Custom ID Into a Neptune Gremlin or SPARQL Query).

Response syntax

```
{
  "queryId": "guid",
  "queryString": "string",
  "queryEvalStats": {
    "waited": integer,
    "elapsed": integer,
    "cancelled": boolean,
    "subqueries": document
  }
}
```

- **queryId** – The ID of the query.
- **queryString** – The submitted query. This is truncated to 1024 characters if it is longer than that.
- **queryEvalStats** – Statistics for the query, including `waited` (wait time in milliseconds), `elapsed` (run time in milliseconds), `cancelled` (whether the query was cancelled), and `subqueries` (the number of subqueries).

Example

AWS CLI

```
aws neptunedata get-gremlin-query-status \
```

```
  --endpoint-url https://your-neptune-endpoint:port \
  --query-id "fb34cd3e-f37c-4d12-9cf2-03bb741bf54f"
```

For more information, see get-gremlin-query-status in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_gremlin_query_status(
    queryId='fb34cd3e-f37c-4d12-9cf2-03bb741bf54f'
)

print(response)
```

For AWS SDK examples in other languages like Java, .NET, and more, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/gremlin/status/fb34cd3e-
f37c-4d12-9cf2-03bb741bf54f \
  --region us-east-1 \
  --service neptune-db
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

```
curl https://your-neptune-endpoint:port/gremlin/status/fb34cd3e-
f37c-4d12-9cf2-03bb741bf54f
```

The following is an example response.

```
{
  "queryId": "fb34cd3e-f37c-4d12-9cf2-03bb741bf54f",
  "queryString": "g.V().out().count()",
  "queryEvalStats": {
    "waited": 0,
    "elapsed": 23,
    "cancelled": false
  }
}
```

Gremlin query cancellation

To get the status of Gremlin queries, use HTTP `GET` or `POST` to make a request to the `https://your-neptune-endpoint:port/gremlin/status` endpoint.

Gremlin query cancellation request parameters

- **cancelQuery** – Required for cancellation. This parameter has no corresponding value.
- **queryId** – The ID of the running Gremlin query to cancel.

Gremlin query cancellation example

The following is an example of cancelling a query.

AWS CLI

```
aws neptunedata cancel-gremlin-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --query-id "fb34cd3e-f37c-4d12-9cf2-03bb741bf54f"
```

For more information, see cancel-gremlin-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.cancel_gremlin_query(
    queryId='fb34cd3e-f37c-4d12-9cf2-03bb741bf54f'
)

print(response)
```

For AWS SDK examples in other languages like Java, .NET, and more, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/gremlin/status \
  --region us-east-1 \
  --service neptune-db \
  --data-urlencode "cancelQuery" \
  --data-urlencode "queryId=fb34cd3e-f37c-4d12-9cf2-03bb741bf54f"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

```
curl https://your-neptune-endpoint:port/gremlin/status \
  --data-urlencode "cancelQuery" \
  --data-urlencode "queryId=fb34cd3e-f37c-4d12-9cf2-03bb741bf54f"
```

Successful cancellation returns HTTP `200` OK.

Support for Gremlin script-based sessions

You can use Gremlin sessions with implicit transactions in Amazon Neptune. For information about Gremlin sessions, see Considering Sessions in the Apache TinkerPop documentation. The sections below describe how to use Gremlin sessions with Java.

> **Important**
>
> Currently, the longest time Neptune can keep a script-based session open is 10 minutes. If you don't close a session before that, the session times out and everything in it is rolled back.

Topics

- Gremlin sessions on the Gremlin console
- Gremlin sessions in the Gremlin Language Variant

Gremlin sessions on the Gremlin console

If you create a remote connection on the Gremlin Console without the `session` parameter, the remote connection is created in *sessionless* mode. In this mode, each request that is submitted to the server is treated as a complete transaction in itself, and no state is saved between requests. If a request fails, only that request is rolled back.

If you create a remote connection that *does* use the `session` parameter, you create a script-based session that lasts until you close the remote connection. Every session is identified by a unique UUID that the console generates and returns to you.

The following is an example of one console call that creates a session. After queries are submitted, another call closes the session and commits the queries.

> **Note**
>
> The Gremlin client must always be closed to release server side resources.

```
gremlin> :remote connect tinkerpop.server conf/neptune-remote.yaml session
  . . .
  . . .
gremlin> :remote close
```

For more information and examples, see Sessions in the TinkerPop documentation.

All the queries that you run during a session form a single transaction that isn't committed until all the queries succeed and you close the remote connection. If a query fails, or if you don't close the connection within the maximum session lifetime that Neptune supports, the session transaction is not committed, and all the queries in it are rolled back.

Gremlin sessions in the Gremlin Language Variant

In the Gremlin language variant (GLV), you need to create a `SessionedClient` object to issue multiple queries in a single transaction, as in the following example.

```
try {                               // line 1
  Cluster cluster = Cluster.open();                    // line 2
  Client client = cluster.connect("sessionName");      // line 3
   ...
   ...
} finally {
  // Always close. If there are no errors, the transaction is committed; otherwise,
 it's rolled back.
  client.close();
}
```

Line 3 in the preceding example creates the `SessionedClient` object according to the configuration options set for the cluster in question. The *`sessionName`* string that you pass to the connect method becomes the unique name of the session. To avoid collisions, use a UUID for the name.

The client starts a session transaction when it is initialized. All the queries that you run during the session form are committed only when you call `client.close( )`. Again, if a single query fails, or if you don't close the connection within the maximum session lifetime that Neptune supports, the session transaction fails, and all the queries in it are rolled back.

> **Note**
>
> The Gremlin client must always be closed to release server side resources.

```
GraphTraversalSource g = traversal().withRemote(conn);

Transaction tx = g.tx();

// Spawn a GraphTraversalSource from the Transaction.
// Traversals spawned from gtx are executed within a single transaction.
GraphTraversalSource gtx = tx.begin();
try {
  gtx.addV('person').iterate();
  gtx.addV('software').iterate();

  tx.commit();
} finally {
    if (tx.isOpen()) {
        tx.rollback();
    }
}
```

Gremlin transactions in Neptune

There are several contexts within which Gremlin transactions are executed. When working with Gremlin it is important to understand the context you are working within and what its implications are:

- **`Script-based`** – Requests are made using text-based Gremlin strings, like this:
  - Using the Java driver and `Client.submit(string)`.
  - Using the Gremlin console and `:remote connect`.
  - Using the HTTP API.
- **`Bytecode-based`** – Requests are made using serialized Gremlin bytecode typical of Gremlin Language Variants(GLV).

  For example, using the Java driver, `g = traversal().withRemote(...)`.

For either of the above contexts, there is the additional context of the request being sent as sessionless or as bound to a session.

> **Note**
>
> Gremlin transactions must always either be committed or rolled back, so that server-side resources can be released. In the event of an error during the transaction, it is important to retry the entire transaction and not just the particular request that failed.

Sessionless requests

When sessionless, a request is equivalent to a single transaction.

For scripts, the implication is that one or more Gremlin statements sent in a single request will commit or rollback as a single transaction. For example:

```
Cluster cluster = Cluster.open();
Client client = cluster.connect(); // sessionless
// 3 vertex additions in one request/transaction:
client.submit("g.addV();g.addV();g.addV()").all().get();
```

For bytecode, a sessionless request is made for each traversal spawned and executed from g:

```
GraphTraversalSource g = traversal().withRemote(...);

// 3 vertex additions in three individual requests/transactions:
g.addV().iterate();
g.addV().iterate();
g.addV().iterate();

// 3 vertex additions in one single request/transaction:
g.addV().addV().addV().iterate();
```

Requests bound to a session

When bound to a session, multiple requests can be applied within the context of a single transaction.

For scripts, the implication is that there is no need to concatenate together all of the graph operations into a single embedded string value:

```
Cluster cluster = Cluster.open();
Client client = cluster.connect(sessionName); // session
try {
    // 3 vertex additions in one request/transaction:
    client.submit("g.addV();g.addV();g.addV()").all().get();
} finally {
    client.close();
}

try {
    // 3 vertex additions in three requests, but one transaction:
    client.submit("g.addV()").all().get(); // starts a new transaction with the same
 sessionName
    client.submit("g.addV()").all().get();
    client.submit("g.addV()").all().get();
} finally {
    client.close();
}
```

For script-based sessions, closing the client with `client.close()` commits the transaction. There is no explicit rollback command available in script-based sessions. To force a rollback, you can cause the transaction to fail by issuing a query such as `g.inject(0).fail('rollback')` before closing the client.

> **Note**
>
> A query like `g.inject(0).fail('rollback')`, used to intentionally throw an error to force a rollback, produces an exception on the client. Catch and discard the resulting exception before closing the client.

For bytecode, the transaction can be explicitly controlled and the session managed transparently. Gremlin Language Variants (GLV) support Gremlin's `tx()` syntax to `commit()` or `rollback()` a transaction as follows:

```
GraphTraversalSource g = traversal().withRemote(conn);

Transaction tx = g.tx();

// Spawn a GraphTraversalSource from the Transaction.
```

```
// Traversals spawned from gtx are executed within a single transaction.
GraphTraversalSource gtx = tx.begin();
try {
    gtx.addV('person').iterate();
    gtx.addV('software').iterate();

    tx.commit();
} finally {
    if (tx.isOpen()) {
        tx.rollback();
    }
}
```

Although the example above is written in Java, you can also use this `tx()` syntax in other languages. For language-specific transaction syntax, see the Transactions section of the Apache TinkerPop documentation for Java, Python, Javascript, .NET, and Go.

> **Warning**
>
> Sessionless read-only queries are executed under SNAPSHOT isolation, but read-only queries run within an explicit transaction are executed under SERIALIZABLE isolation. The read-only queries executed under `SERIALIZABLE` isolation incur higher overhead and can block or get blocked by concurrent writes, unlike those run under `SNAPSHOT` isolation.

Timeout behavior for bytecode commit and rollback

When you use bytecode-based transactions with the `tx()` syntax, the `commit()` and `rollback()` operations are not subject to query timeout settings. Neither the global `neptune_query_timeout` parameter nor per-query timeout values set through `evaluationTimeout` apply to these operations. On the server, `commit()` and `rollback()` run without a time limit until they complete or encounter an error.

On the client side, the Gremlin driver's `tx.commit()` and `tx.rollback()` calls will not complete until the server responds. Depending on the language, this might manifest as a blocking call or an unresolved async operation. No driver provides a built-in timeout setting that bounds these calls. Consult the API documentation for your specific Gremlin Language Variant for details on concurrency behavior around these transaction features.

> **Important**
>
> If a `commit()` or `rollback()` call takes longer than expected, it might be blocked by lock contention from a concurrent transaction. For more information about lock conflicts, see Conflict Resolution Using Lock-Wait Timeouts.

If you need to bound the time your application waits for a `commit()` or `rollback()`, you can use your language's concurrency features to apply a client-side timeout. If the client-side timeout fires, the server continues processing the operation. The server-side operation holds a worker thread until it completes. After a client-side timeout, close the connection and create a new one rather than reusing the existing connection, because the transaction state is indeterminate.

Server-side transaction cleanup

If a client disconnects or abandons a transaction without committing or rolling back, Neptune has server-side mechanisms that eventually clean up the orphaned transaction:

- **Session timeout** – Bytecode-based sessions that remain idle for longer than the maximum session lifetime (10 minutes) are closed, and any open transaction is rolled back.
- **Connection idle timeout** – Neptune closes WebSocket connections that are idle for approximately 20 minutes. When the connection closes, the server rolls back any open transaction associated with that connection.

These cleanup mechanisms are safety nets. We recommend that you always explicitly commit or roll back transactions when you are finished with them.

Streaming query results with Gremlin

When you run a Gremlin traversal that returns a large number of results, Neptune streams them back to the client in batches over the WebSocket connection. Neptune sends result batches as they are produced, without waiting for the client to request more. This can be advantageous if you want to process results as they are being returned from the server, but requires using lazy iteration patterns to avoid collecting the full result set into memory.

Neptune sends results in batches of 64 per WebSocket frame by default. You cannot change this server-side default, but the batch size can be overridden on a per-request basis from the client

using the `batchSize` request option (called `Tokens.ARGS_BATCH_SIZE` in the Java driver, or `connectionPool.resultIterationBatchSize` as a driver-level default).

For details on configuring `batchSize` in other language drivers, see the Configuration section for each driver in the Apache TinkerPop Gremlin Drivers and Variants documentation.

Because the server pushes results automatically, client-side backpressure is handled implicitly through TCP and WebSocket flow control. If the client is slow to read from the socket, the server's writes will eventually block until the client catches up.

> **Important**
>
> Streaming is most effective with traversals that can produce results incrementally. Traversals that include `order()`, `groupCount()`, `group()`, `dedup()`, or other steps that require the full traversal to complete before emitting results will cause Neptune to materialize the entire result set in memory before streaming begins. In these cases, batching still reduces per-frame serialization overhead, but does not reduce server-side memory usage.

Consuming results incrementally

To process results as they arrive, iterate lazily using `hasNext()` / `next()` or equivalent APIs rather than collecting all results into a list. You can use `next(batchSize)` to pull results in application-level batches, allowing you to perform intermediate work between batches while the server continues producing results.

Example Java (GLV bytecode)

```
GraphTraversalSource g = traversal().withRemote(connection);

int batchSize = 10;
int batchNum = 0;
var traversal = g.V().hasLabel("movie").values("title").limit(1000);
while (traversal.hasNext()) {
    var batch = traversal.next(batchSize);
    batchNum++;
    for (var title : batch) {
        System.out.println("  " + title);
    }
```

```
    // Do other intermediary work here between batch calls
    System.out.println("Batch " + batchNum + " processing complete\n");
}
```

Example Python

```
g = traversal().with_remote(connection)

BATCH_SIZE = 10
batch_num = 0
t = g.V().has_label('movie').values('title').limit(1000)
while t.has_next():
    batch = t.next(BATCH_SIZE)
    batch_num += 1
    for title in batch:
        print(f"  {title}")

    # Do other intermediary work here between batch calls
    print(f"Batch {batch_num} processing complete\n")
```

Example Go

```
// The Go driver does not support next(n), so batches are accumulated manually.
g := gremlingo.Traversal_().WithRemote(connection)

resultSet, err := g.V().HasLabel("movie").Values("title").Limit(1000).GetResultSet()
if err != nil {
    log.Fatal(err)
}

batchSize := 10
batchNum := 0
for {
    var batch []interface{}
    for i := 0; i < batchSize; i++ {
        result, ok, err := resultSet.One() // returns (value, ok, error); ok is false
 when results are exhausted
        if err != nil {
            log.Fatal(err)
        }
        if !ok {
            break
```

```
        }
        batch = append(batch, result)
    }
    if len(batch) == 0 {
        break
    }
    batchNum++
    for _, v := range batch {
        fmt.Printf("  %v\n", v)
    }

    // Do other intermediary work here between batch calls
    fmt.Printf("Batch %d processing complete\n\n", batchNum)
}
```

Example.NET

```
var g = Traversal().WithRemote(connection);

var batchSize = 10;
var batchNum = 0;
var traversal = g.V().HasLabel("movie").Values<string>("title").Limit<string>(1000);
while (traversal.HasNext())
{
    var batch = traversal.Next(batchSize);
    batchNum++;
    foreach (var title in batch)
    {
        Console.WriteLine($"  {title}");
    }

    // Do other intermediary work here between batch calls
    Console.WriteLine($"Batch {batchNum} processing complete\n");
}
```

Example Node.js

```
// The Node.js driver does not support next(n), so batches are accumulated manually.
const g = traversal().withRemote(connection);

const batchSize = 10;
let batchNum = 0;
const t = g.V().hasLabel('movie').values('title').limit(1000);
```

```
while (true) {
    const batch = [];
    for (let i = 0; i < batchSize; i++) {
        const result = await t.next();
        if (result.done) break;
        batch.push(result.value);
    }
    if (batch.length === 0) break;
    batchNum++;
    for (const title of batch) {
        console.log(`  ${title}`);
    }

    // Do other intermediary work here between batch calls
    console.log(`Batch ${batchNum} processing complete\n`);
}
```

Eager vs. incremental consumption

Streaming allows you to process results incrementally as additional data is being fetched and returned. The following methods block until the entire result set is collected into memory, preventing your application from acting on results as they arrive:

- **Java:** `toList()` or `toSet()`
- **Python:** `toList()` or `toSet()`
- **Go:** `ToList()`, `ToSet()`, or `GetResultSet().GetAll()`
- **.NET:** `ToList()` or `Promise()`
- **Node.js:** `toList()`

> **Note**
>
> Data still flows incrementally over the WebSocket connection even when using these methods. The difference is that your application cannot process individual results until the entire collection is complete. To process results as they arrive, use the lazy iteration or batch patterns shown in the examples above.

Using the Gremlin API with Amazon Neptune

> **Note**
>
> Amazon Neptune does not support the `bindings` property.

Gremlin HTTPS requests all use a single endpoint: `https://your-neptune-endpoint:port/gremlin`. All Neptune connections must use HTTPS.

You can connect the Gremlin Console to a Neptune graph directly through WebSockets.

For more information about connecting to the Gremlin endpoint, see Accessing a Neptune graph with Gremlin.

The Amazon Neptune implementation of Gremlin has specific details and differences that you need to consider. For more information, see Gremlin standards compliance in Amazon Neptune.

For information about the Gremlin language and traversals, see The Traversal in the Apache TinkerPop documentation.

Caching query results in Amazon Neptune Gremlin

Amazon Neptune supports a results cache for Gremlin queries.

You can enable the query results cache and then use a query hint to cache the results of a Gremlin read-only query.

Any re-run of the query then retrieves the cached results with low latency and no I/O costs, as long as they are still in the cache. This works for queries submitted both on an HTTP endpoint and using Websockets, either as byte-code or in string form.

> **Note**
>
> Queries sent to the profile endpoint are not cached even when the query cache is enabled.

You can control how the Neptune query results cache behaves in several ways. For example:

- You can get cached results paginated, in blocks.
- You can specify the time-to-live (TTL) for specified queries.

- You can clear the cache for specified queries.
- You can clear the entire cache.
- You can set up to be notified if results exceed the cache size.

The cache is maintained using a least-recently-used (LRU) policy, meaning that once the space allotted to the cache is full, the least-recently-used results are removed to make room when new results are being cached.

> **Important**
>
> The query-results cache is not available on `t3.medium` or `t4.medium` instance types.

Enabling the query results cache in Neptune

The query results cache can be enabled across all instances in a cluster or per-instance. To enable the results cache on all instances in a cluster, set the `neptune_result_cache` parameter in the cluster's `cluster-parameter-group` to 1. To enable this on a specific instance, set the `neptune_result_cache` parameter in the instance's `instance-parameter-group` to 1. The cluster parameter group setting will override the instance parameter group value.

A restart is required on any affected instances for the results cache parameter settings to be applied. While you can enable the results cache across all instances in a cluster via the `cluster-parameter-group`, each instance maintains its own cache. The query results cache feature is not a cluster-wide cache.

Once the results cache is enabled, Neptune sets aside a portion of current memory for caching query results. The larger the instance type you're using and the more memory is available, the more memory Neptune sets aside for the cache.

If the results cache memory fills up, Neptune automatically drops least-recently-used (LRU) cached results to make way for new ones.

You can check the current status of the results cache using the Instance Status command.

Using hints to cache query results

Once the query results cache is enabled, you use query hints to control query caching. All the examples below apply to the same query traversal, namely:

```
g.V().has('genre','drama').in('likes')
```

Using `enableResultCache`

With the query results cache enabled, you can cache the results of a Gremlin query using the `enableResultCache` query hint, as follows:

```
g.with('Neptune#enableResultCache', true)
 .V().has('genre','drama').in('likes')
```

Neptune then returns the query results to you, and also caches them. Later, you can access the cached results by issuing exactly the same query again:

```
g.with('Neptune#enableResultCache', true)
 .V().has('genre','drama').in('likes')
```

The cache key that identifies the cached results is the query string itself, namely:

```
g.V().has('genre','drama').in('likes')
```

Using `enableResultCacheWithTTL`

You can specify how long the query results should be cached for by using the `enableResultCacheWithTTL` query hint. For example, the following query specifies that the query results should expire after 120 seconds:

```
g.with('Neptune#enableResultCacheWithTTL', 120)
 .V().has('genre','drama').in('likes')
```

Again, the cache key that identifies the cached results is the base query string:

```
g.V().has('genre','drama').in('likes')
```

And again, you can access the cached results using that query string with the `enableResultCache` query hint:

```
g.with('Neptune#enableResultCache', true)
```

```
 .V().has('genre','drama').in('likes')
```

If 120 or more seconds have passed since the results were cached, that query will return new results, and cache them, without any time-to-live.

You can also access the cached results by issuing the same query again with the `enableResultCacheWithTTL` query hint. For example:

```
g.with('Neptune#enableResultCacheWithTTL', 140)
 .V().has('genre','drama').in('likes')
```

Until 120 seconds have passed (that is, the TTL currently in effect), this new query using the `enableResultCacheWithTTL` query hint returns the cached results. After 120 seconds, it would return new results and cache them with a time-to-live of 140 seconds.

> **Note**
>
> If results for a query key are already cached, then the same query key with `enableResultCacheWithTTL` does not generate new results and has no effect on the time-to-live of the currently cached results.
>
> - If results were previously cached using `enableResultCache`, the cache must first be cleared before `enableResultCacheWithTTL` generates new results and caches them for the TTL that it specifies.
> - If results were previously cached using `enableResultCachewithTTL`, that previous TTL must first expire before `enableResultCacheWithTTL` generates new results and caches them for the TTL that it specifies.

Using `invalidateResultCacheKey`

You can use the `invalidateResultCacheKey` query hint to clear cached results for one particular query. For example:

```
g.with('Neptune#invalidateResultCacheKey', true)
 .V().has('genre','drama').in('likes')
```

That query clears the cache for the query key, `g.V().has('genre','drama').in('likes')`, and returns new results for that query.

You can also combine `invalidateResultCacheKey` with `enableResultCache` or `enableResultCacheWithTTL`. For example, the following query clears the current cached results, caches new results, and returns them:

```
g.with('Neptune#enableResultCache', true)
 .with('Neptune#invalidateResultCacheKey', true)
 .V().has('genre','drama').in('likes')
```

Using `invalidateResultCache`

You can use the `invalidateResultCache` query hint to clear all cached results in the query result cache. For example:

```
g.with('Neptune#invalidateResultCache', true)
 .V().has('genre','drama').in('likes')
```

That query clears the entire result cache and returns new results for the query.

You can also combine `invalidateResultCache` with `enableResultCache` or `enableResultCacheWithTTL`. For example, the following query clears the entire results cache, caches new results for this query, and returns them:

```
g.with('Neptune#enableResultCache', true)
 .with('Neptune#invalidateResultCache', true)
 .V().has('genre','drama').in('likes')
```

Paginating cached query results

Suppose you have already cached a large number of results like this:

```
g.with('Neptune#enableResultCache', true)
 .V().has('genre','drama').in('likes')
```

Now suppose you issue the following range query:

```
g.with('Neptune#enableResultCache', true)
 .V().has('genre','drama').in('likes').range(0,10)
```

Neptune first looks for the full cache key, namely `g.V().has('genre','drama').in('likes').range(0,10)`. If that key doesn't exist,

Neptune next looks to see if there is a key for that query string without the range (namely `g.V().has('genre','drama').in('likes')`). When it finds that key, Neptune then fetches the first ten results from its cache, as the range specifies.

> **Note**
>
> If you use the `invalidateResultCacheKey` query hint with a query that has a range at the end, Neptune clears the cache for a query without the range if it doesn't find an exact match for the query with the range.

Using `numResultsCached` with `.iterate()`

Using the `numResultsCached` query hint, you can populate the results cache without returning all the results being cached, which can be useful when you prefer to paginate a large number of results.

The `numResultsCached` query hint only works with queries that end with `iterate()`.

For example, if you want to cache the first 50 results of the sample query:

```
g.with("Neptune#enableResultCache", true)
 .with("Neptune#numResultsCached", 50)
 .V().has('genre','drama').in('likes').iterate()
```

In this case the query key in the cache is: `g.with("Neptune#numResultsCached", 50).V().has('genre','drama').in('likes')`. You can now retrieve the first ten of the cached results with this query:

```
g.with("Neptune#enableResultCache", true)
 .with("Neptune#numResultsCached", 50)
 .V().has('genre','drama').in('likes').range(0, 10)
```

And, you can retrieve the next ten results from the query as follows:

```
g.with("Neptune#enableResultCache", true)
 .with("Neptune#numResultsCached", 50)
 .V().has('genre','drama').in('likes').range(10, 20)
```

Don't forget to include the `numResultsCached` hint! It is an essential part of the query key and must therefore be present in order to access the cached results.

Some things to keep in mind when using `numResultsCached`

- **The number you supply with `numResultsCached` is applied at the end of the query.** This means, for example, that the following query actually caches results in the range `(1000, 1500)`:

```
g.with("Neptune#enableResultCache", true)
 .with("Neptune#numResultsCached", 500)
 .V().range(1000, 2000).iterate()
```

- **The number you supply with `numResultsCached` specifies the maximum number of results to cache.** This means, for example, that the following query actually caches results in the range `(1000, 2000)`:

```
g.with("Neptune#enableResultCache", true)
 .with("Neptune#numResultsCached", 100000)
 .V().range(1000, 2000).iterate()
```

- **Results cached by queries that end with `.range().iterate()` have their own range.** For example, suppose you cache results using a query like this:

```
g.with("Neptune#enableResultCache", true)
 .with("Neptune#numResultsCached", 500)
 .V().range(1000, 2000).iterate()
```

  To retrieve the first 100 results from the cache, you would write a query like this:

```
g.with("Neptune#enableResultCache", true)
 .with("Neptune#numResultsCached", 500)
 .V().range(1000, 2000).range(0, 100)
```

  Those hundred results would be equivalent to results from the base query in the range `(1000, 1100)`.

The query cache keys used to locate cached results

After the results of a query have been cached, subsequent queries with the same *query cache key* retrieve results from the cache rather than generating new ones. The query cache key of a query is evaluated as follows:

1. All the cache-related query hints are ignored, except for `numResultsCached`.
2. A final `iterate()` step is ignored.
3. The rest of the query is ordered according to its byte-code representation.

The resulting string is matched against an index of the query results already in the cache to determine whether there is a cache hit for the query.

For example, take this query:

```
g.withSideEffect('Neptune#typePromotion', false).with("Neptune#enableResultCache",
 true)
 .with("Neptune#numResultsCached", 50)
 .V().has('genre','drama').in('likes').iterate()
```

It will be stored as the byte-code version of this:

```
g.withSideEffect('Neptune#typePromotion', false)
 .with("Neptune#numResultsCached", 50)
 .V().has('genre','drama').in('likes')
```

Exceptions related to the results cache

If the results of a query that you are trying to cache are too large to fit in the cache memory even after removing everything previously cached, Neptune raises a `QueryLimitExceededException` fault. No results are returned, and the exception generates the following error message:

```
The result size is larger than the allocated cache,
      please refer to results cache best practices for options to rerun the query.
```

You can supress this message using the `noCacheExceptions` query hint, as follows:

```
g.with('Neptune#enableResultCache', true)
 .with('Neptune#noCacheExceptions', true)
```

```
.V().has('genre','drama').in('likes')
```

Making efficient upserts with Gremlin `mergeV()` and `mergeE()` steps

An upsert (or conditional insert) reuses a vertex or edge if it already exists, or creates it if it doesn't. Efficient upserts can make a significant difference in the performance of Gremlin queries.

Upserts allow you to write idempotent insert operations: no matter how many times you run such an operation, the overall outcome is the same. This is useful in highly concurrent write scenarios where concurrent modifications to the same part of the graph can force one or more transactions to roll back with a `ConcurrentModificationException`, thereby necessitating retries.

> **Note**
>
> The `mergeV()` and `mergeE()` steps require Neptune engine version 1.2.1.0 or later, which introduced TinkerPop 3.6 support. If your cluster is running an older engine version, see Efficient upserts before 3.6.x instead. You can check your current engine version using the instance status endpoint.

For example, the following query upserts a vertex by using the supplied `Map` to first try to find a vertex with a `T.id` of `"v-1"`. If that vertex is found then it is returned. If it is not found then a vertex with that `id` and property are created through the `onCreate` clause.

```
g.mergeV([(id):'v-1']).
  option(onCreate, [(label): 'PERSON', 'email': 'person-1@example.org'])
```

Batching upserts to improve throughput

For high throughput write scenarios, you can chain `mergeV()` and `mergeE()` steps together to upsert vertices and edges in batches. Batching reduces the transactional overhead of upserting large numbers of vertices and edges. You can then further improve throughput by upserting batch requests in parallel using multiple clients.

As a rule of thumb we recommend upserting approximately 200 records per batch request. A record is a single vertex or edge label or property. A vertex with a single label and 4 properties, for example, creates 5 records. An edge with a label and a single property creates 2 records. If you wanted to upsert batches of vertices, each with a single label and 4 properties, you should start with a batch size of 40, because `200 / (1 + 4) = 40`.

You can experiment with the batch size. 200 records per batch is a good starting point, but the ideal batch size may be higher or lower depending on your workload. Note, however, that Neptune may limit the overall number of Gremlin steps per request. This limit is not documented, but to be on the safe side, try to ensure that your requests contain no more than 1,500 Gremlin steps. Neptune may reject large batch requests with more than 1,500 steps.

To increase throughput, you can upsert batches in parallel using multiple clients (see Creating Efficient Multithreaded Gremlin Writes). The number of clients should be the same as the number of worker threads on your Neptune writer instance, which is typically 2 x the number of vCPUs on the server. For instance, an `r5.8xlarge` instance has 32 vCPUs and 64 worker threads. For high-throughput write scenarios using an `r5.8xlarge`, you would use 64 clients writing batch upserts to Neptune in parallel.

Each client should submit a batch request and wait for the request to complete before submitting another request. Although the multiple clients run in parallel, each individual client submits requests in a serial fashion. This ensures that the server is supplied with a steady stream of requests that occupy all the worker threads without flooding the server-side request queue (see Sizing DB instances in a Neptune DB cluster).

Try to avoid steps that generate multiple traversers

When a Gremlin step executes, it takes an incoming traverser, and emits one or more output traversers. The number of traversers emitted by a step determines the number of times the next step is executed.

Typically, when performing batch operations you want each operation, such as upsert vertex A, to execute once, so that the sequence of operations looks like this: upsert vertex A, then upsert vertex B, then upsert vertex C, and so on. As long as a step creates or modifies only one element, it emits only one traverser, and the steps that represent the next operation are executed only once. If, on the other hand, an operation creates or modifies more than one element, it emits multiple traversers, which in turn cause the subsequent steps to be executed multiple times, once per emitted traverser. This can result in the database performing unnecessary additional work, and in some cases can result in the creation of unwanted additional vertices, edges or property values.

An example of how things can go wrong is with a query like `g.V().addV()`. This simple query adds a vertex for every vertex found in the graph, because `V()` emits a traverser for each vertex in the graph and each of those traversers triggers a call to `addV()`.

See Mixing upserts and inserts for ways to deal with operations that can emit multiple traversers.

Upserting vertices

The `mergeV()` step is specifically designed for upserting vertices. It takes as an argument a `Map` that represents elements to match for existing vertices in the graph, and if an element is not found, uses that `Map` to create a new vertex. The step also allows you to alter the behavior in the event of a creation or a match, where the `option()` modulator can be applied with `Merge.onCreate` and `Merge.onMatch` tokens to control those respective behaviors. See the TinkerPop Reference Documentation for further information about how to use this step.

You can use a vertex ID to determine whether a specific vertex exists. This is the preferred approach, because Neptune optimizes upserts for highly concurrent use cases around IDs. As an example, the following query creates a vertex with a given vertex ID if it doesn't already exist, or reuses it if it does:

```
g.mergeV([(T.id): 'v-1']).
    option(onCreate, [(T.label): 'PERSON', email: 'person-1@example.org', age: 21]).
    option(onMatch, [age: 22]).
  id()
```

Note that this query ends with an `id()` step. While not strictly necessary for the purpose of upserting the vertex, an `id()` step to the end of an upsert query ensures that the server doesn't serialize all the vertex properties back to the client, which helps reduce the locking cost of the query.

Alternatively, you can use a vertex property to identify a vertex:

```
g.mergeV([email: 'person-1@example.org']).
    option(onCreate, [(T.label): 'PERSON', age: 21]).
    option(onMatch, [age: 22]).
  id()
```

If possible, use your own user-supplied IDs to create vertices, and use these IDs to determine whether a vertex exists during an upsert operation. This lets Neptune optimize the upserts. An ID-based upsert can be significantly more efficient than a property-based upsert when concurrent modifications are common.

Chaining vertex upserts

You can chain vertex upserts together to insert them in a batch:

```
g.V('v-1')
 .fold()
 .coalesce(unfold(),
          addV('Person').property(id, 'v-1')
                        .property('email', 'person-1@example.org'))
 .V('v-2')
 .fold()
 .coalesce(unfold(),
          addV('Person').property(id, 'v-2')
                        .property('email', 'person-2@example.org'))
 .V('v-3')
 .fold()
 .coalesce(unfold(),
          addV('Person').property(id, 'v-3')
                        .property('email', 'person-3@example.org'))
 .id()
```

Alternatively, you can also use this `mergeV()` syntax:

```
g.mergeV([(T.id): 'v-1', (T.label): 'PERSON', email: 'person-1@example.org']).
  mergeV([(T.id): 'v-2', (T.label): 'PERSON', email: 'person-2@example.org']).
  mergeV([(T.id): 'v-3', (T.label): 'PERSON', email: 'person-3@example.org'])
```

However, because this form of the query includes elements in the search criteria that are superfluous to the basic lookup by `id`, it isn't as efficient as the previous query.

Upserting edges

The `mergeE()` step is specifically designed for upserting edges. It takes a `Map` as an argument that represents elements to match for existing edges in the graph and if an element is not found, uses that `Map` to create a new edge. The step also allows you to alter the behavior in the event of a creation or a match, where the `option()` modulator can be applied with `Merge.onCreate` and `Merge.onMatch` tokens to control those respective behaviors. See the TinkerPop Reference Documentation for further information about how to use this step.

You can use edge IDs to upsert edges in the same way you upsert vertices using custom vertex IDs. Again, this is the preferred approach because it allows Neptune to optimize the query. For example, the following query creates an edge based on its edge ID if it doesn't already exist, or reuses it if it does. The query also uses the IDs of the `Direction.from` and `Direction.to` vertices if it needs to create a new edge:

```
g.mergeE([(T.id): 'e-1']).
    option(onCreate, [(from): 'v-1', (to): 'v-2', weight: 1.0]).
    option(onMatch, [weight: 0.5]).
  id()
```

Note that this query ends with an `id()` step. While not strictly necessary for the purpose of upserting the edge, adding an `id()` step to the end of an upsert query ensures that the server doesn't serialize all the edge properties back to the client, which helps reduce the locking cost of the query.

Many applications use custom vertex IDs, but leave Neptune to generate edge IDs. If you don't know the ID of an edge, but you do know the `from` and `to` vertex IDs, you can use this kind of query to upsert an edge:

```
g.mergeE([(from): 'v-1', (to): 'v-2', (T.label): 'KNOWS']).
  id()
```

All vertices referenced by `mergeE()` must exist for the step to create the edge.

Chaining edge upserts

As with vertex upserts, it's straightforward to chain `mergeE()` steps together for batch requests:

```
g.mergeE([(from): 'v-1', (to): 'v-2', (T.label): 'KNOWS']).
  mergeE([(from): 'v-2', (to): 'v-3', (T.label): 'KNOWS']).
  mergeE([(from): 'v-3', (to): 'v-4', (T.label): 'KNOWS']).
  id()
```

Combining vertex and edge upserts

Sometimes you may want to upsert both vertices and the edges that connect them. You can mix the batch examples presented here. The following example upserts 3 vertices and 2 edges:

```
g.mergeV([(id):'v-1']).
    option(onCreate, [(label): 'PERSON', 'email': 'person-1@example.org']).
  mergeV([(id):'v-2']).
    option(onCreate, [(label): 'PERSON', 'email': 'person-2@example.org']).
  mergeV([(id):'v-3']).
    option(onCreate, [(label): 'PERSON', 'email': 'person-3@example.org']).
  mergeE([(from): 'v-1', (to): 'v-2', (T.label): 'KNOWS']).
```

```
  mergeE([(from): 'v-2', (to): 'v-3', (T.label): 'KNOWS']).
 id()
```

Mixing upserts and inserts

Sometimes you may want to upsert both vertices and the edges that connect them. You can mix the batch examples presented here. The following example upserts 3 vertices and 2 edges:

Upserts typically proceed one element at a time. If you stick to the upsert patterns presented here, each upsert operation emits a single traverser, which causes the subsequent operation to be executed just once.

However, sometimes you may want to mix upserts with inserts. This can be the case, for example, if you use edges to represent instances of actions or events. A request might use upserts to ensure that all necessary vertices exist, and then use inserts to add edges. With requests of this kind, pay attention to the potential number of traversers being emitted from each operation.

> **Neptune 1.5.0.0: mergeV()/mergeE() callback change**
>
> Starting with TinkerPop 3.8.0, the `onCreate` and `onMatch` callbacks for `mergeV()` and `mergeE()` are passed the matched or newly created element (the vertex or edge) instead of the incoming traverser. This change affects mid-traversal `mergeV()` and `mergeE()` usage like the chained examples in this section. The upsert patterns shown here continue to work, but if you reference the incoming traverser inside an `onCreate` or `onMatch` child traversal, update your code to use the element directly. For more information, see the TinkerPop 3.8.0 upgrade documentation on the Apache TinkerPop website.

Consider the following example, which mixes upserts and inserts to add edges that represent events into the graph:

```
// Fully optimized, but inserts too many edges
g.mergeV([(id):'v-1']).
    option(onCreate, [(label): 'PERSON', 'email': 'person-1@example.org']).
  mergeV([(id):'v-2']).
    option(onCreate, [(label): 'PERSON', 'email': 'person-2@example.org']).
  mergeV([(id):'v-3']).
    option(onCreate, [(label): 'PERSON', 'email': 'person-3@example.org']).
  mergeV([(T.id): 'c-1', (T.label): 'CITY', name: 'city-1']).
  V('p-1', 'p-2').
```

```
  addE('FOLLOWED').to(V('p-1')).
  V('p-1', 'p-2', 'p-3').
  addE('VISITED').to(V('c-1')).
  id()
```

The query should insert 5 edges: 2 FOLLOWED edges and 3 VISITED edges. However, the query as written inserts 8 edges: 2 FOLLOWED and 6 VISITED. The reason for this is that the operation that inserts the 2 FOLLOWED edges emits 2 traversers, causing the subsequent insert operation, which inserts 3 edges, to be executed twice.

The fix is to add a `fold()` step after each operation that can potentially emit more than one traverser:

```
g.mergeV([(T.id): 'v-1', (T.label): 'PERSON', email: 'person-1@example.org']).
  mergeV([(T.id): 'v-2', (T.label): 'PERSON', email: 'person-2@example.org']).
  mergeV([(T.id): 'v-3', (T.label): 'PERSON', email: 'person-3@example.org']).
  mergeV([(T.id): 'c-1', (T.label): 'CITY', name: 'city-1']).
  V('p-1', 'p-2').
  addE('FOLLOWED').
    to(V('p-1')).
  fold().
  V('p-1', 'p-2', 'p-3').
  addE('VISITED').
    to(V('c-1')).
  id()
```

Here we've inserted a `fold()` step after the operation that inserts FOLLOWED edges. This results in a single traverser, which then causes the subsequent operation to be executed only once.

The downside of this approach is that the query is now not fully optimized, because `fold()` is not optimized. The insert operation that follows `fold()` will now also not be optimized.

If you need to use `fold()` to reduce the number of traversers on behalf of subsequent steps, try to order your operations so that the least expensive ones occupy the non-optimized part of the query.

Setting Cardinality

The default cardinality for vertex properties in Neptune is set, which means that when using mergeV() the values supplied in the map are all going to be given that cardinality. To use single cardinality, you must be explicit in its usage. Starting in TinkerPop 3.7.0, there is a new syntax that allows the cardinality to be supplied as part of the map as shown in the following example:

```
g.mergeV([(T.id): '1234']).
  option(onMatch, ['age': single(20), 'name': single('alice'), 'city': set('miami')])
```

Alternatively, you may set the cardinality as a default for that `option` as follows:

```
// age and name are set to single cardinality by default
g.mergeV([(T.id): '1234']).
  option(onMatch, ['age': 22, 'name': 'alice', 'city': set('boston')], single)
```

There are fewer options for setting cardinality in `mergeV()` prior to version 3.7.0. The general approach is to fall back to the `property()` step as follows:

```
g.mergeV([(T.id): '1234']).
  option(onMatch, sideEffect(property(single,'age', 20).
  property(set,'city','miami')).constant([:]))
```

> **Note**
>
> This approach will only work with `mergeV()` when it is used with a start step. You would therefore not be able to chain `mergeV()` within a single traversal as the first `mergeV()` after the start step that uses this syntax will produce an error should the incoming traverser be a graph element. In this case, you would want to break up your `mergeV()` calls into multiple requests where each can be a start step.

Making efficient Gremlin upserts with `fold()/coalesce()/unfold()`

An upsert (or conditional insert) reuses a vertex or edge if it already exists, or creates it if it doesn't. Efficient upserts can make a significant difference in the performance of Gremlin queries.

This page shows how use the `fold()/coalesce()/unfold()` Gremlin pattern to make efficient upserts. However, with the release of TinkerPop version 3.6.x introduced in Neptune in engine version 1.2.1.0, the new `mergeV()` and `mergeE()` steps are preferable in most cases. The `fold()/coalesce()/unfold()` pattern described here may still be useful in a some complex situations, but in general use `mergeV()` and `mergeE()` if you can, as described in Making efficient upserts with Gremlin `mergeV()` and `mergeE()` steps.

Upserts allow you to write idempotent insert operations: no matter how many times you run such an operation, the overall outcome is the same. This is useful in highly concurrent write scenarios

where concurrent modifications to the same part of the graph can force one or more transactions to roll back with a `ConcurrentModificationException`, thereby necessitating a retry.

For example, the following query upserts a vertex by first looking for the specified vertex in the dataset, and then folding the results into a list. In the first traversal supplied to the `coalesce()` step, the query then unfolds this list. If the unfolded list is not empty, the results are emitted from the `coalesce()`. If, however, the `unfold()` returns an empty collection because the vertex does not currently exist, `coalesce()` moves on to evaluate the second traversal with which it has been supplied, and in this second traversal the query creates the missing vertex.

```
g.V('v-1').fold()
          .coalesce(
             unfold(),
             addV('Person').property(id, 'v-1')
                           .property('email', 'person-1@example.org')
           )
```

Use an optimized form of `coalesce()` for upserts

Neptune can optimize the `fold().coalesce(unfold(), ...)` idiom to make high-throughput updates, but this optimization only works if both parts of the `coalesce()` return either a vertex or an edge but nothing else. If you try to return something different, such as a property, from any part of the `coalesce()`, the Neptune optimization does not occur. The query may succeed, but it will not perform as well as an optimized version, particularly against large datasets.

Because unoptimized upsert queries increase execution times and reduce throughput, it's worth using the Gremlin `explain` endpoint to determine whether an upsert query is fully optimized. When reviewing `explain` plans, look for lines that begin with `+ not converted into Neptune steps` and `WARNING: >>`. For example:

```
+ not converted into Neptune steps: [FoldStep, CoalesceStep([[UnfoldStep],
 [AddEdgeSte...
WARNING: >> FoldStep << is not supported natively yet
```

These warnings can help you identify the parts of a query that are preventing it from being fully optimized.

Sometimes it isn't possible to optimize a query fully. In these situations you should try to put the steps that cannot be optimized at the end of the query, thereby allowing the engine to optimize as many steps as possible. This technique is used in some of the batch upsert examples, where all

optimized upserts for a set of vertices or edges are performed before any additional, potentially unoptimized modifications are applied to the same vertices or edges.

Batching upserts to improve throughput

For high throughput write scenarios, you can chain upsert steps together to upsert vertices and edges in batches. Batching reduces the transactional overhead of upserting large numbers of vertices and edges. You can then further improve throughput by upserting batch requests in parallel using multiple clients.

As a rule of thumb we recommend upserting approximately 200 records per batch request. A record is a single vertex or edge label or property. A vertex with a single label and 4 properties, for example, creates 5 records. An edge with a label and a single property creates 2 records. If you wanted to upsert batches of vertices, each with a single label and 4 properties, you should start with a batch size of 40, because `200 / (1 + 4) = 40`.

You can experiment with the batch size. 200 records per batch is a good starting point, but the ideal batch size may be higher or lower depending on your workload. Note, however, that Neptune may limit the overall number of Gremlin steps per request. This limit is not documented, but to be on the safe side try to ensure that your requests contain no more than 1500 Gremlin steps. Neptune may reject large batch requests with more than 1500 steps.

To increase throughput, you can upsert batches in parallel using multiple clients (see Creating Efficient Multithreaded Gremlin Writes). The number of clients should be the same as the number of worker threads on your Neptune writer instance, which is typically 2 x the number of vCPUs on the server. For instance, an `r5.8xlarge` instance has 32 vCPUs and 64 worker threads. For high-throughput write scenarios using an `r5.8xlarge`, you would use 64 clients writing batch upserts to Neptune in parallel.

Each client should submit a batch request and wait for the request to complete before submitting another request. Although the multiple clients run in parallel, each individual client submits requests in a serial fashion. This ensures that the server is supplied with a steady stream of requests that occupy all the worker threads without flooding the server-side request queue (see Sizing DB instances in a Neptune DB cluster).

Try to avoid steps that generate multiple traversers

When a Gremlin step executes, it takes an incoming traverser, and emits one or more output traversers. The number of traversers emitted by a step determines the number of times the next step is executed.

Typically, when performing batch operations you want each operation, such as upsert vertex A, to execute once, so that the sequence of operations looks like this: upsert vertex A, then upsert vertex B, then upsert vertex C, and so on. As long as a step creates or modifies only one element, it emits only one traverser, and the steps that represent the next operation are executed only once. If, on the other hand, an operation creates or modifies more than one element, it emits multiple traversers, which in turn cause the subsequent steps to be executed multiple times, once per emitted traverser. This can result in the database performing unnecessary additional work, and in some cases can result in the creation of unwanted additional vertices, edges or property values.

An example of how things can go wrong is with a query like `g.V().addV()`. This simple query adds a vertex for every vertex found in the graph, because `V()` emits a traverser for each vertex in the graph and each of those traversers triggers a call to `addV()`.

See Mixing upserts and inserts for ways to deal with operations that can emit multiple traversers.

Upserting vertices

You can use a vertex ID to determine whether a corresponding vertex exists. This is the preferred approach, because Neptune optimizes upserts for highly concurrent use cases around IDs. As an example, the following query creates a vertex with a given vertex ID if it doesn't already exist, or reuses it if it does:

```
g.V('v-1')
 .fold()
  .coalesce(unfold(),
            addV('Person').property(id, 'v-1')
                          .property('email', 'person-1@example.org'))
  .id()
```

Note that this query ends with an `id()` step. While not strictly necessary for the purpose of upserting the vertex, adding an `id()` step to the end of an upsert query ensures that the server doesn't serialize all the vertex properties back to the client, which helps reduce the locking cost of the query.

Alternatively, you can use a vertex property to determine whether the vertex exists:

```
g.V()
 .hasLabel('Person')
 .has('email', 'person-1@example.org')
```

```
 .fold()
 .coalesce(unfold(),
           addV('Person').property('email', 'person-1@example.org'))
 .id()
```

If possible, use your own user-supplied IDs to create vertices, and use these IDs to determine whether a vertex exists during an upsert operation. This lets Neptune optimize upserts around the IDs. An ID-based upsert can be significantly more efficient than a property-based upsert in highly concurrent modification scenarios.

Chaining vertex upserts

You can chain vertex upserts together to insert them in a batch:

```
g.V('v-1')
 .fold()
 .coalesce(unfold(),
           addV('Person').property(id, 'v-1')
                         .property('email', 'person-1@example.org'))
 .V('v-2')
 .fold()
 .coalesce(unfold(),
           addV('Person').property(id, 'v-2')
                         .property('email', 'person-2@example.org'))
 .V('v-3')
 .fold()
 .coalesce(unfold(),
           addV('Person').property(id, 'v-3')
                         .property('email', 'person-3@example.org'))
 .id()
```

Upserting edges

You can use edge IDs to upsert edges in the same way you upsert vertices using custom vertex IDs. Again, this is the preferred approach because it allows Neptune to optimize the query. For example, the following query creates an edge based on its edge ID if it doesn't already exist, or reuses it if it does. The query also uses the IDs of the `from` and `to` vertices if it needs to create a new edge.

```
g.E('e-1')
 .fold()
 .coalesce(unfold(),
```

```
            addE('KNOWS').from(V('v-1'))
                         .to(V('v-2'))
                         .property(id, 'e-1'))
  .id()
```

Many applications use custom vertex IDs, but leave Neptune to generate edge IDs. If you don't know the ID of an edge, but you do know the `from` and `to` vertex IDs, you can use this formulation to upsert an edge:

```
g.V('v-1')
 .outE('KNOWS')
 .where(inV().hasId('v-2'))
 .fold()
 .coalesce(unfold(),
           addE('KNOWS').from(V('v-1'))
                        .to(V('v-2')))
 .id()
```

Note that the vertex step in the `where()` clause should be `inV()` (or `outV()` if you've used `inE()` to find the edge), not `otherV()`. Do not use `otherV()`, here, or the query will not be optimized and performance will suffer. For example, Neptune would not optimize the following query:

```
// Unoptimized upsert, because of otherV()
g.V('v-1')
 .outE('KNOWS')
 .where(otherV().hasId('v-2'))
 .fold()
 .coalesce(unfold(),
           addE('KNOWS').from(V('v-1'))
                        .to(V('v-2')))
 .id()
```

If you don't know the edge or vertex IDs up front, you can upsert using vertex properties:

```
g.V()
 .hasLabel('Person')
 .has('name', 'person-1')
 .outE('LIVES_IN')
 .where(inV().hasLabel('City').has('name', 'city-1'))
 .fold()
 .coalesce(unfold(),
```

```
            addE('LIVES_IN').from(V().hasLabel('Person')
                                      .has('name', 'person-1'))
                                .to(V().hasLabel('City')
                                       .has('name', 'city-1')))
  .id()
```

As with vertex upserts, it's preferable to use ID-based edge upserts using either an edge ID or `from` and `to` vertex IDs, rather than property-based upserts, so that Neptune can fully optimize the upsert.

Checking for `from` and `to` vertex existence

Note the construction of the steps that create a new edge: `addE().from().to()`. This construction ensures that the query checks the existence of both the `from` and the `to` vertex. If either of these does not exist, the query returns an error as follows:

```
{
  "detailedMessage": "Encountered a traverser that does not map to a value for child...
  "code": "IllegalArgumentException",
  "requestId": "..."
}
```

If it's possible that either the `from` or the `to` vertex doesn't exist, you should attempt to upsert them before upserting the edge between them. See Combining vertex and edge upserts.

There's an alternative construction for creating an edge that you shouldn't use: `V().addE().to()`. It only adds an edge if the `from` vertex exists. If the `to` vertex doesn't exist the, query generates an error, as described previously, but if the `from` vertex doesn't exist, it silently fails to insert an edge, without generating any error. For example, the following upsert completes without upserting an edge if the `from` vertex doesn't exist:

```
// Will not insert edge if from vertex does not exist
g.V('v-1')
 .outE('KNOWS')
 .where(inV().hasId('v-2'))
 .fold()
 .coalesce(unfold(),
           V('v-1').addE('KNOWS')
                   .to(V('v-2')))
 .id()
```

Chaining edge upserts

If you want to chain edge upserts together to create a batch request, you must begin each upsert with a vertex lookup, even if you already know the edge IDs.

If you do already know the IDs of the edges you want to upsert, and the IDs of the `from` and `to` vertices, you can use this formulation:

```
g.V('v-1')
 .outE('KNOWS')
 .hasId('e-1')
 .fold()
 .coalesce(unfold(),
          V('v-1').addE('KNOWS')
                  .to(V('v-2'))
                  .property(id, 'e-1'))
 .V('v-3')
 .outE('KNOWS')
 .hasId('e-2').fold()
 .coalesce(unfold(),
          V('v-3').addE('KNOWS')
                  .to(V('v-4'))
                  .property(id, 'e-2'))
 .V('v-5')
 .outE('KNOWS')
 .hasId('e-3')
 .fold()
 .coalesce(unfold(),
          V('v-5').addE('KNOWS')
                  .to(V('v-6'))
                  .property(id, 'e-3'))
 .id()
```

Perhaps the most common batch edge upsert scenario is that you know the `from` and `to` vertex IDs, but don't know the IDs of the edges you want to upsert. In that case, use the following formulation:

```
g.V('v-1')
 .outE('KNOWS')
 .where(inV().hasId('v-2'))
 .fold()
 .coalesce(unfold(),
```

```
          V('v-1').addE('KNOWS')
                  .to(V('v-2')))

 .V('v-3')
 .outE('KNOWS')
 .where(inV().hasId('v-4'))
 .fold()
 .coalesce(unfold(),
          V('v-3').addE('KNOWS')
                  .to(V('v-4')))
 .V('v-5')
 .outE('KNOWS')
 .where(inV().hasId('v-6'))
 .fold()
 .coalesce(unfold(),
          V('v-5').addE('KNOWS').to(V('v-6')))
 .id()
```

If you know IDs of the edges you want to upsert, but don't know the IDs of the `from` and `to` vertices (this is unusual), you can use this formulation:

```
g.V()
 .hasLabel('Person')
 .has('email', 'person-1@example.org')
 .outE('KNOWS')
 .hasId('e-1')
 .fold()
 .coalesce(unfold(),
          V().hasLabel('Person')
             .has('email', 'person-1@example.org')
             .addE('KNOWS')
             .to(V().hasLabel('Person')
                    .has('email', 'person-2@example.org'))
                    .property(id, 'e-1'))
 .V()
 .hasLabel('Person')
 .has('email', 'person-3@example.org')
 .outE('KNOWS')
 .hasId('e-2')
 .fold()
 .coalesce(unfold(),
          V().hasLabel('Person')
             .has('email', 'person-3@example.org')
```

```
              .addE('KNOWS')
              .to(V().hasLabel('Person')
                     .has('email', 'person-4@example.org'))
              .property(id, 'e-2'))
 .V()
 .hasLabel('Person')
 .has('email', 'person-5@example.org')
 .outE('KNOWS')
 .hasId('e-1')
 .fold()
 .coalesce(unfold(),
           V().hasLabel('Person')
              .has('email', 'person-5@example.org')
              .addE('KNOWS')
              .to(V().hasLabel('Person')
                     .has('email', 'person-6@example.org'))
                     .property(id, 'e-3'))
 .id()
```

Combining vertex and edge upserts

Sometimes you may want to upsert both vertices and the edges that connect them. You can mix the batch examples presented here. The following example upserts 3 vertices and 2 edges:

```
g.V('p-1')
 .fold()
 .coalesce(unfold(),
           addV('Person').property(id, 'p-1')
                         .property('email', 'person-1@example.org'))
 .V('p-2')
 .fold()
 .coalesce(unfold(),
           addV('Person').property(id, 'p-2')
                         .property('name', 'person-2@example.org'))
 .V('c-1')
 .fold()
 .coalesce(unfold(),
           addV('City').property(id, 'c-1')
                       .property('name', 'city-1'))
 .V('p-1')
 .outE('LIVES_IN')
 .where(inV().hasId('c-1'))
 .fold()
```

```
 .coalesce(unfold(),
          V('p-1').addE('LIVES_IN')
                  .to(V('c-1')))
 .V('p-2')
 .outE('LIVES_IN')
 .where(inV().hasId('c-1'))
 .fold()
 .coalesce(unfold(),
          V('p-2').addE('LIVES_IN')
                  .to(V('c-1')))
 .id()
```

Mixing upserts and inserts

Sometimes you may want to upsert both vertices and the edges that connect them. You can mix the batch examples presented here. The following example upserts 3 vertices and 2 edges:

Upserts typically proceed one element at a time. If you stick to the upsert patterns presented here, each upsert operation emits a single traverser, which causes the subsequent operation to be executed just once.

However, sometimes you may want to mix upserts with inserts. This can be the case, for example, if you use edges to represent instances of actions or events. A request might use upserts to ensure that all necessary vertices exist, and then use inserts to add edges. With requests of this kind, pay attention to the potential number of traversers being emitted from each operation.

Consider the following example, which mixes upserts and inserts to add edges that represent events into the graph:

```
// Fully optimized, but inserts too many edges
g.V('p-1')
 .fold()
 .coalesce(unfold(),
          addV('Person').property(id, 'p-1')
                        .property('email', 'person-1@example.org'))
 .V('p-2')
 .fold()
 .coalesce(unfold(),
          addV('Person').property(id, 'p-2')
                        .property('name', 'person-2@example.org'))
 .V('p-3')
```

```
 .fold()
 .coalesce(unfold(),
          addV('Person').property(id, 'p-3')
                        .property('name', 'person-3@example.org'))
 .V('c-1')
 .fold()
 .coalesce(unfold(),
          addV('City').property(id, 'c-1')
                      .property('name', 'city-1'))
 .V('p-1', 'p-2')
 .addE('FOLLOWED')
 .to(V('p-1'))
 .V('p-1', 'p-2', 'p-3')
 .addE('VISITED')
 .to(V('c-1'))
 .id()
```

The query should insert 5 edges: 2 FOLLOWED edges and 3 VISITED edges. However, the query as written inserts 8 edges: 2 FOLLOWED and 6 VISITED. The reason for this is that the operation that inserts the 2 FOLLOWED edges emits 2 traversers, causing the subsequent insert operation, which inserts 3 edges, to be executed twice.

The fix is to add a `fold()` step after each operation that can potentially emit more than one traverser:

```
g.V('p-1')
 .fold()
 .coalesce(unfold(),
          addV('Person').property(id, 'p-1')
                        .property('email', 'person-1@example.org'))
 .V('p-2')
 .fold()
 .coalesce(unfold(),
          addV('Person').property(id, 'p-2').
                        .property('name', 'person-2@example.org'))
 .V('p-3')
 .fold()
 .coalesce(unfold(),
          addV('Person').property(id, 'p-3').
                        .property('name', 'person-3@example.org'))
 .V('c-1')
 .fold().
 .coalesce(unfold(),
```

```
            addV('City').property(id, 'c-1').
                        .property('name', 'city-1'))
 .V('p-1', 'p-2')
 .addE('FOLLOWED')
 .to(V('p-1'))
 .fold()
 .V('p-1', 'p-2', 'p-3')
 .addE('VISITED')
 .to(V('c-1')).
 .id()
```

Here we've inserted a `fold()` step after the operation that inserts FOLLOWED edges. This results in a single traverser, which then causes the subsequent operation to be executed only once.

The downside of this approach is that the query is now not fully optimized, because `fold()` is not optimized. The insert operation that follows `fold()` will now not be optimized.

If you need to use `fold()` to reduce the number of traversers on behalf of subsequent steps, try to order your operations so that the least expensive ones occupy the non-optimized part of the query.

Upserts that modify existing vertices and edges

Sometimes you want to create a vertex or edge if it doesn't exist, and then add or update a property to it, regardless of whether it is a new or existing vertex or edge.

To add or modify a property, use the `property()` step. Use this step outside the `coalesce()` step. If you try to modify the property of an existing vertex or edge inside the `coalesce()` step, the query may not be optimized by the Neptune query engine.

The following query adds or updates a counter property on each upserted vertex. Each `property()` step has single cardinality to ensure that the new values replace any existing values, rather than being added to a set of existing values.

```
g.V('v-1')
 .fold()
 .coalesce(unfold(),
          addV('Person').property(id, 'v-1')
                        .property('email', 'person-1@example.org'))
 .property(single, 'counter', 1)
 .V('v-2')
 .fold()
```

```
 .coalesce(unfold(),
           addV('Person').property(id, 'v-2')
                         .property('email', 'person-2@example.org'))
 .property(single, 'counter', 2)
 .V('v-3')
 .fold()
 .coalesce(unfold(),
           addV('Person').property(id, 'v-3')
                         .property('email', 'person-3@example.org'))
 .property(single, 'counter', 3)
 .id()
```

If you have a property value, such as a `lastUpdated` timestamp value, that applies to all upserted elements, you can add or update it at the end of the query:

```
g.V('v-1')
 .fold()
 .coalesce(unfold(),
           addV('Person').property(id, 'v-1')
                         .property('email', 'person-1@example.org'))
 .V('v-2').
 .fold().
 .coalesce(unfold(),
           addV('Person').property(id, 'v-2')
                         .property('email', 'person-2@example.org'))
 .V('v-3')
 .fold()
 .coalesce(unfold(),
           addV('Person').property(id, 'v-3')
                         .property('email', 'person-3@example.org'))
 .V('v-1', 'v-2', 'v-3')
 .property(single, 'lastUpdated', datetime('2020-02-08'))
 .id()
```

If there are additional conditions that determine whether or not a vertex or edge should be further modified, you can use a `has()` step to filter the elements to which a modification will be applied. The following example uses a `has()` step to filter upserted vertices based on the value of their `version` property. The query then updates to 3 the `version` of any vertex whose `version` is less than 3:

```
g.V('v-1')
 .fold()
 .coalesce(unfold(),
           addV('Person').property(id, 'v-1')
                         .property('email', 'person-1@example.org')
                         .property('version', 3))
 .V('v-2')
 .fold()
 .coalesce(unfold(),
           addV('Person').property(id, 'v-2')
                         .property('email', 'person-2@example.org')
                         .property('version', 3))
 .V('v-3')
 .fold()
 .coalesce(unfold(),
           addV('Person').property(id, 'v-3')
                         .property('email', 'person-3@example.org')
                         .property('version', 3))
 .V('v-1', 'v-2', 'v-3')
 .has('version', lt(3))
 .property(single, 'version', 3)
 .id()
```

Analyzing Neptune query execution using Gremlin `explain`

Amazon Neptune has added a Gremlin feature named *explain*. This feature is a self-service tool for understanding the execution approach taken by the Neptune engine. You invoke it by adding an `explain` parameter to an HTTP call that submits a Gremlin query.

The `explain` feature provides information about the logical structure of query execution plans. You can use this information to identify potential evaluation and execution bottlenecks and tune your query, as explained in Tuning Gremlin queries. You can also use query hints to improve query execution plans.

Topics

- Understanding how Gremlin queries work in Neptune
- Using the Gremlin explain API in Neptune
- Gremlin profile API in Neptune
- Tuning Gremlin queries using explain and profile
- Native Gremlin step support in Amazon Neptune

Understanding how Gremlin queries work in Neptune

To take full advantage of the Gremlin `explain` and `profile` reports in Amazon Neptune, it is helpful to understand some background information about Gremlin queries.

Topics

- Gremlin statements in Neptune
- How Neptune processes Gremlin queries using statement indexes
- How Gremlin queries are processed in Neptune

Gremlin statements in Neptune

Property graph data in Amazon Neptune is composed of four-position (quad) statements. Each of these statements represents an individual atomic unit of property graph data. For more information, see Neptune Graph Data Model. Similar to the Resource Description Framework (RDF) data model, these four positions are as follows:

- `subject (S)`
- `predicate (P)`
- `object (O)`
- `graph (G)`

Each statement is an assertion about one or more resources. For example, a statement can assert the existence of a relationship between two resources, or it can attach a property (key-value pair) to some resource.

You can think of the predicate as the verb of the statement, describing the type of relationship or property. The object is the target of the relationship, or the value of the property. The graph position is optional and can be used in many different ways. For the Neptune property graph (PG) data, it is either unused (null graph) or it is used to represent the identifier for an edge. A set of statements with shared resource identifiers creates a graph.

There are three classes of statements in the Neptune property graph data model:

Topics

- Gremlin Vertex Label Statements

- Gremlin Edge Statements
- Gremlin Property Statements

Gremlin Vertex Label Statements

Vertex label statements in Neptune serve two purposes:

- They track the labels for a vertex.
- The presence of at least one of these statements is what implies the existence of a particular vertex in the graph.

The subject of these statements is a vertex identifier, and the object is a label, both of which are specified by the user. You use a special fixed predicate for these statements, displayed as `<~label>`, and a default graph identifier (the null graph), displayed as `<~>`.

For example, consider the following `addV` traversal.

```
g.addV("Person").property(id, "v1")
```

This traversal results in the following statement being added to the graph.

```
StatementEvent[Added(<v1> <~label> <Person> <~>) .]
```

Gremlin Edge Statements

A Gremlin edge statement is what implies the existence of an edge between two vertices in a graph in Neptune. The subject (S) of an edge statement is the source `from` vertex. The predicate (P) is a user-supplied edge label. The object (O) is the target `to` vertex. The graph (G) is a user-supplied edge identifier.

For example, consider the following `addE` traversal.

```
g.addE("knows").from(V("v1")).to(V("v2")).property(id, "e1")
```

The traversal results in the following statement being added to the graph.

```
StatementEvent[Added(<v1> <knows> <v2> <e1>) .]
```

Gremlin Property Statements

A Gremlin property statement in Neptune asserts an individual property value for a vertex or edge. The subject is a user-supplied vertex or edge identifier. The predicate is the property name (key), and the object is the individual property value. The graph (G) is again the default graph identifier, the null graph, displayed as <~>.

Consider the following vertex property example.

```
g.V("v1").property("name", "John")
```

This statement results in the following.

```
StatementEvent[Added(<v1> <name> "John" <~>) .]
```

Property statements differ from others in that their object is a primitive value (a `string`, `date`, `byte`, `short`, `int`, `long`, `float`, or `double`). Their object is not a resource identifier that could be used as the subject of another assertion.

For multi-properties, each individual property value in the set receives its own statement.

```
g.V("v1").property(set, "phone", "956-424-2563").property(set, "phone", "956-354-3692
 (tel:9563543692)")
```

This results in the following.

```
StatementEvent[Added(<v1> <phone> "956-424-2563" <~>) .]
StatementEvent[Added(<v1> <phone> "956-354-3692" <~>) .]
```

Edge properties are handled similarly to vertex properties, but use the edge identifier in the (S) position. For example, adding a property to an edge:

```
g.E("e1").property("weight", 0.8)
```

This results in the following statement being added to the graph.

```
StatementEvent[Added(<e1> <weight> 0.8 <~>) .]
```

How Neptune processes Gremlin queries using statement indexes

Statements are accessed in Amazon Neptune by way of three statement indexes, as detailed in How Statements Are Indexed in Neptune. Neptune extracts a statement *pattern* from a Gremlin query in which some positions are known, and the rest are left for discovery by index search.

Neptune assumes that the size of the property graph schema is not large. This means that the number of distinct edge labels and property names is fairly low, resulting in a low total number of distinct predicates. Neptune tracks distinct predicates in a separate index. It uses this cache of predicates to do a union scan of `{ all P x POGS }` rather than use an OSGP index. Avoiding the need for a reverse traversal OSGP index saves both storage space and load throughput.

The Neptune Gremlin Explain/Profile API lets you obtain the predicate count in your graph. You can then determine whether your application invalidates the Neptune assumption that your property graph schema is small.

The following examples help illustrate how Neptune uses indexes to process Gremlin queries.

Question: What are the labels of vertex v1?

```
Gremlin code:      g.V('v1').label()
Pattern:           (<v1>, <~label>, ?, ?)
Known positions:   SP
Lookup positions:  OG
Index:             SPOG
Key range:         <v1>:<~label>:*
```

Question: What are the 'knows' out-edges of vertex v1?

```
Gremlin code:      g.V('v1').out('knows')
Pattern:           (<v1>, <knows>, ?, ?)
Known positions:   SP
Lookup positions:  OG
Index:             SPOG
Key range:         <v1>:<knows>:*
```

Question: Which vertices have a `Person` vertex label?

```
Gremlin code:      g.V().hasLabel('Person')
Pattern:           (?, <~label>, <Person>, <~>)
```

```
Known positions:   POG
Lookup positions:  S
Index:             POGS
Key range:         <~label>:<Person>:<~>:*
```

Question: What are the from/to vertices of a given edge e1?

```
Gremlin code:      g.E('e1').bothV()
Pattern:           (?, ?, ?, <e1>)
Known positions:   G
Lookup positions:  SPO
Index:             GPSO
Key range:         <e1>:*
```

One statement index that Neptune does **not** have is a reverse traversal OSGP index. This index could be used to gather all incoming edges across all edge labels, as in the following example.

Question: What are the incoming adjacent vertices v1?

```
Gremlin code:      g.V('v1').in()
Pattern:           (?, ?, <v1>, ?)
Known positions:   O
Lookup positions:  SPG
Index:             OSGP  // <-- Index does not exist
```

How Gremlin queries are processed in Neptune

In Amazon Neptune, more complex traversals can be represented by a series of patterns that create a relation based on the definition of named variables that can be shared across patterns to create joins. This is shown in the following example.

Question: What is the two-hop neighborhood of vertex v1?

```
Gremlin code:      g.V(‘v1’).out('knows').out('knows').path()
Pattern:           (?1=<v1>, <knows>, ?2, ?) X Pattern(?2, <knows>, ?3, ?)

The pattern produces a three-column relation (?1, ?2, ?3) like this:
                   ?1     ?2     ?3
                   ================
                   v1     v2     v3
                   v1     v2     v4
```

```
                        v1      v5      v6
```

By sharing the `?2` variable across the two patterns (at the O position in the first pattern and the S position of the second pattern), you create a join from the first hop neighbors to the second hop neighbors. Each Neptune solution has bindings for the three named variables, which can be used to re-create a TinkerPop Traverser (including path information).

The first step in Gremlin query processing is to parse the query into a TinkerPop Traversal object, composed of a series of TinkerPop steps. These steps, which are part of the open-source Apache TinkerPop project, are both the logical and physical operators that compose a Gremlin traversal in the reference implementation. They are both used to represent the model of the query. They are executable operators that can produce solutions according to the semantics of the operator that they represent. For example, `.V()` is both represented and executed by the TinkerPop GraphStep.

Because these off-the-shelf TinkerPop steps are executable, such a TinkerPop Traversal can execute any Gremlin query and produce the correct answer. However, when executed against a large graph, TinkerPop steps can sometimes be very inefficient and slow. Instead of using them, Neptune tries to convert the traversal into a declarative form composed of groups of patterns, as described previously.

Neptune doesn't currently support all Gremlin operators (steps) in its native query engine. So it tries to collapse as many steps as possible down into a single `NeptuneGraphQueryStep`, which contains the declarative logical query plan for all the steps that have been converted. Ideally, all steps are converted. But when a step is encountered that can't be converted, Neptune breaks out of native execution and defers all query execution from that point forward to the TinkerPop steps. It doesn't try to weave in and out of native execution.

After the steps are translated into a logical query plan, Neptune runs a series of query optimizers that rewrite the query plan based on static analysis and estimated cardinalities. These optimizers do things like reorder operators based on range counts, prune unnecessary or redundant operators, rearrange filters, push operators into different groups, and so on.

After an optimized query plan is produced, Neptune creates a pipeline of physical operators that do the work of executing the query. This includes reading data from the statement indices, performing joins of various types, filtering, ordering, and so on. The pipeline produces a solution stream that is then converted back into a stream of TinkerPop Traverser objects.

Serialization of query results

Amazon Neptune currently relies on the TinkerPop response message serializers to convert query results (TinkerPop Traversers) into the serialized data to be sent over the wire back to the client. These serialization formats tend to be quite verbose.

For example, to serialize the result of a vertex query such as `g.V().limit(1)`, the Neptune query engine must perform a single search to produce the query result. However, the `GraphSON` serializer would perform a large number of additional searches to package the vertex into the serialization format. It would have to perform one search to get the label, one to get the property keys, and one search per property key for the vertex to get all the values for each key.

Some of the serialization formats are more efficient, but all require additional searches. Additionally, the TinkerPop serializers don't try to avoid duplicated searches, often resulting in many searches being repeated unnecessarily.

This makes it very important to write your queries so that they ask specifically just for the information they need. For example, `g.V().limit(1).id()` would return just the vertex ID and eliminate all the additional serializer searches. The Gremlin `profile` API in Neptune allows you to see how many search calls are made during query execution and during serialization.

Using the Gremlin `explain` API in Neptune

The Amazon Neptune Gremlin `explain` API returns the query plan that would be executed if a specified query were run. Because the API doesn't actually run the query, the plan is returned almost instantaneously.

It differs from the TinkerPop .explain() step so as to be able to report information specific to the Neptune engine.

Information contained in a Gremlin `explain` report

An `explain` report contains the following information:

- The query string as requested.
- **The original traversal.** This is the TinkerPop Traversal object produced by parsing the query string into TinkerPop steps. It is equivalent to the original query produced by running `.explain()` on the query against the TinkerPop TinkerGraph.
- **The converted traversal.** This is the Neptune Traversal produced by converting the TinkerPop Traversal into the Neptune logical query plan representation. In many cases the entire

TinkerPop traversal is converted into two Neptune steps: one that executes the entire query (`NeptuneGraphQueryStep`) and one that converts the Neptune query engine output back into TinkerPop Traversers (`NeptuneTraverserConverterStep`).

- **The optimized traversal.** This is the optimized version of the Neptune query plan after it has been run through a series of static work-reducing optimizers that rewrite the query based on static analysis and estimated cardinalities. These optimizers do things like reorder operators based on range counts, prune unnecessary or redundant operators, rearrange filters, push operators into different groups, and so on.
- **The predicate count.** Because of the Neptune indexing strategy described earlier, having a large number of different predicates can cause performance problems. This is especially true for queries that use reverse traversal operators with no edge label (`.in` or `.both`). If such operators are used and the predicate count is high enough, the `explain` report displays a warning message.
- **DFE information.** When the DFE alternative engine is enabled, the following traversal components may show up in the optimized traversal:
  - **`DFEStep`** – A Neptune optimized DFE step in the traversal that contains a child `DFENode`. `DFEStep` represents the part of the query plan that is executed in the DFE engine.
  - **`DFENode`** – Contains the intermediate representation as one or more child `DFEJoinGroupNodes`.
  - **`DFEJoinGroupNode`** – Represents a join of one or more `DFENode` or `DFEJoinGroupNode` elements.
  - **`NeptuneInterleavingStep`** – A Neptune optimized DFE step in the traversal that contains a child `DFEStep`.

    Also contains a `stepInfo` element that contains information about the traversal, such as the frontier element, the path elements used, and so on. This information is used to process the child `DFEStep`.

An easy way to find out if your query is being evaluated by DFE is to check whether the `explain` output contains a `DFEStep`. Any part of the traversal that is not part of the `DFEStep` will not be executed by DFE and will be executed by the TinkerPop engine.

See Example with DFE enabled for a sample report.

Gremlin `explain` syntax

The syntax of the `explain` API is the same as that for the HTTP API for query, except that it uses `/gremlin/explain` as the endpoint instead of `/gremlin`, as in the following examples.

AWS CLI

```
aws neptunedata execute-gremlin-explain-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --gremlin-query "g.V().limit(1)"
```

For more information, see execute-gremlin-explain-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_gremlin_explain_query(
    gremlinQuery='g.V().limit(1)'
)

print(response['output'])
```

For AWS SDK examples in other languages like Java, .NET, and more, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/gremlin/explain \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d '{"gremlin":"g.V().limit(1)"}'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

```
curl -X POST https://your-neptune-endpoint:port/gremlin/explain \
  -d '{"gremlin":"g.V().limit(1)"}'
```

The preceding query would produce the following output.

```
*******************************************************
                Neptune Gremlin Explain
*******************************************************

Query String
============
g.V().limit(1)

Original Traversal
==================
[GraphStep(vertex,[]), RangeGlobalStep(0,1)]

Converted Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
            PatternNode[(?1, <~label>, ?2, <~>) . project distinct ?1 .]
        }, finishers=[limit(1)], annotations={path=[Vertex(?1):GraphStep], maxVarId=3}
    },
    NeptuneTraverserConverterStep
]

Optimized Traversal
```

```
===================
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
            PatternNode[(?1, <~label>, ?2, <~>) . project distinct ?1 .],
 {estimatedCardinality=INFINITY}
        }, finishers=[limit(1)], annotations={path=[Vertex(?1):GraphStep], maxVarId=3}
    },
    NeptuneTraverserConverterStep
]

Predicates
==========
# of predicates: 18
```

Unconverted TinkerPop Steps

Ideally, all TinkerPop steps in a traversal have native Neptune operator coverage. When this isn't the case, Neptune falls back on TinkerPop step execution for gaps in its operator coverage. If a traversal uses a step for which Neptune does not yet have native coverage, the `explain` report displays a warning showing where the gap occurred.

When a step without a corresponding native Neptune operator is encountered, the entire traversal from that point forward is run using TinkerPop steps, even if subsequent steps do have native Neptune operators.

The exception to this is when Neptune full-text search is invoked. The NeptuneSearchStep implements steps without native equivalents as full-text search steps.

Example of `explain` output where all steps in a query have native equivalents

The following is an example `explain` report for a query where all steps have native equivalents:

```
*******************************************************
                Neptune Gremlin Explain
*******************************************************

Query String
============
g.V().out()

Original Traversal
```

```
==================
[GraphStep(vertex,[]), VertexStep(OUT,vertex)]

Converted Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
            PatternNode[(?1, <~label>, ?2, <~>) . project distinct ?1 .]
            PatternNode[(?1, ?5, ?3, ?6) . project ?1,?3 . IsEdgeIdFilter(?6) .]
            PatternNode[(?3, <~label>, ?4, <~>) . project ask .]
        }, annotations={path=[Vertex(?1):GraphStep, Vertex(?3):VertexStep], maxVarId=7}
    },
    NeptuneTraverserConverterStep
]

Optimized Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
            PatternNode[(?1, ?5, ?3, ?6) . project ?1,?3 . IsEdgeIdFilter(?6) .],
 {estimatedCardinality=INFINITY}
        }, annotations={path=[Vertex(?1):GraphStep, Vertex(?3):VertexStep], maxVarId=7}
    },
    NeptuneTraverserConverterStep
]

Predicates
==========
# of predicates: 18
```

Example where some steps in a query do not have native equivalents

Neptune handles both `GraphStep` and `VertexStep` natively, but if you introduce a `FoldStep` and `UnfoldStep`, the resulting `explain` output is different:

```
*******************************************************
                Neptune Gremlin Explain
*******************************************************
```

```
Query String
============
g.V().fold().unfold().out()

Original Traversal
==================
[GraphStep(vertex,[]), FoldStep, UnfoldStep, VertexStep(OUT,vertex)]

Converted Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
            PatternNode[(?1, <~label>, ?2, <~>) . project distinct ?1 .]
        }, annotations={path=[Vertex(?1):GraphStep], maxVarId=3}
    },
    NeptuneTraverserConverterStep
]
+ not converted into Neptune steps: [FoldStep, UnfoldStep, VertexStep(OUT,vertex)]

Optimized Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
            PatternNode[(?1, <~label>, ?2, <~>) . project distinct ?1 .],
 {estimatedCardinality=INFINITY}
        }, annotations={path=[Vertex(?1):GraphStep], maxVarId=3}
    },
    NeptuneTraverserConverterStep,
    NeptuneMemoryTrackerStep
]
+ not converted into Neptune steps: [FoldStep, UnfoldStep, VertexStep(OUT,vertex)]

WARNING: >> FoldStep << is not supported natively yet
```

In this case, the `FoldStep` breaks you out of native execution. But even the subsequent `VertexStep` is no longer handled natively because it appears downstream of the `Fold`/`Unfold` steps.

For performance and cost-savings, it's important that you try to formulate traversals so that the maximum amount of work possible is done natively inside the Neptune query engine, instead of by the TinkerPop step implementations.

Example of a query that uses Neptune full-text-search

The following query uses Neptune full-text search:

```
g.withSideEffect("Neptune#fts.endpoint", "some_endpoint")
  .V()
  .tail(100)
  .has("Neptune#fts mark*")
  -------
  .has("name", "Neptune#fts mark*")
  .has("Person", "name", "Neptune#fts mark*")
```

The `.has("name", "Neptune#fts mark*")` part limits the search to vertexes with `name`, while `.has("Person", "name", "Neptune#fts mark*")` limits the search to vertexes with `name` and the label `Person`. This results in the following traversal in the `explain` report:

```
Final Traversal
[NeptuneGraphQueryStep(Vertex) {
    JoinGroupNode {
        PatternNode[(?1, termid(1,URI), ?2, termid(0,URI)) . project distinct ?1 .],
 {estimatedCardinality=INFINITY}
    }, annotations={path=[Vertex(?1):GraphStep], maxVarId=4}
}, NeptuneTraverserConverterStep, NeptuneTailGlobalStep(10),
 NeptuneTinkerpopTraverserConverterStep, NeptuneSearchStep {
    JoinGroupNode {
        SearchNode[(idVar=?3, query=mark*, field=name) . project ask .],
 {endpoint=some_endpoint}
    }
    JoinGroupNode {
        SearchNode[(idVar=?3, query=mark*, field=name) . project ask .],
 {endpoint=some_endpoint}
    }
}]
```

Example of using `explain` when the DFE is enabled

The following is an example of an `explain` report when the DFE alternative query engine is enabled:

```
*******************************************************
                Neptune Gremlin Explain
*******************************************************

Query String
============

g.V().as("a").out().has("name", "josh").out().in().where(eq("a"))


Original Traversal
==================
[GraphStep(vertex,[])@[a], VertexStep(OUT,vertex), HasStep([name.eq(josh)]),
 VertexStep(OUT,vertex), VertexStep(IN,vertex), WherePredicateStep(eq(a))]

Converted Traversal
===================
Neptune steps:
[
    DFEStep(Vertex) {
      DFENode {
        DFEJoinGroupNode[ children={
          DFEPatternNode[(?1, <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>, ?2,
 <http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph>) . project DISTINCT[?1]
 {rangeCountEstimate=unknown}],
          DFEPatternNode[(?1, ?3, ?4, ?5) . project ALL[?1, ?4] graphFilters=(!
= <http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph> . ),
 {rangeCountEstimate=unknown}]
        }, {rangeCountEstimate=unknown}
        ]
      } [Vertex(?1):GraphStep@[a], Vertex(?4):VertexStep]
    } ,
    NeptuneTraverserConverterDFEStep
]
+ not converted into Neptune steps: HasStep([name.eq(josh)]),
Neptune steps:
[
    NeptuneInterleavingStep {
      StepInfo[joinVars=[?7, ?1], frontierElement=Vertex(?7):HasStep,
 pathElements={a=(last,Vertex(?1):GraphStep@[a])}, listPathElement={}, indexTime=0ms],
      DFEStep(Vertex) {
        DFENode {
          DFEJoinGroupNode[ children={
```

```
            DFEPatternNode[(?7, ?8, ?9, ?10) . project ALL[?7, ?9]
 graphFilters=(!= <http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph> . ),
 {rangeCountEstimate=unknown}],
            DFEPatternNode[(?12, ?11, ?9, ?13) . project ALL[?9, ?12]
 graphFilters=(!= <http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph> . ),
 {rangeCountEstimate=unknown}]
          }, {rangeCountEstimate=unknown}
          ]
        } [Vertex(?9):VertexStep, Vertex(?12):VertexStep]
      }
    }
]
+ not converted into Neptune steps: WherePredicateStep(eq(a)),
Neptune steps:
[
    DFECleanupStep
]


Optimized Traversal
===================
Neptune steps:
[
    DFEStep(Vertex) {
      DFENode {
        DFEJoinGroupNode[ children={
          DFEPatternNode[(?1, ?3, ?4, ?5) . project ALL[?1, ?4] graphFilters=(!=
 defaultGraph[526] . ), {rangeCountEstimate=9223372036854775807}]
        }, {rangeCountEstimate=unknown}
        ]
      } [Vertex(?1):GraphStep@[a], Vertex(?4):VertexStep]
    } ,
    NeptuneTraverserConverterDFEStep
]
+ not converted into Neptune steps: NeptuneHasStep([name.eq(josh)]),
Neptune steps:
[
    NeptuneMemoryTrackerStep,
    NeptuneInterleavingStep {
      StepInfo[joinVars=[?7, ?1], frontierElement=Vertex(?7):HasStep,
 pathElements={a=(last,Vertex(?1):GraphStep@[a])}, listPathElement={}, indexTime=0ms],
      DFEStep(Vertex) {
        DFENode {
          DFEJoinGroupNode[ children={
```

```
            DFEPatternNode[(?7, ?8, ?9, ?10) . project ALL[?7, ?9] graphFilters=(!=
 defaultGraph[526] . ), {rangeCountEstimate=9223372036854775807}],
            DFEPatternNode[(?12, ?11, ?9, ?13) . project ALL[?9, ?12] graphFilters=(!=
 defaultGraph[526] . ), {rangeCountEstimate=9223372036854775807}]
         }, {rangeCountEstimate=unknown}
         ]
       } [Vertex(?9):VertexStep, Vertex(?12):VertexStep]
      }
    }
]
+ not converted into Neptune steps: WherePredicateStep(eq(a)),
Neptune steps:
[
    DFECleanupStep
]


WARNING: >> [NeptuneHasStep([name.eq(josh)]), WherePredicateStep(eq(a))] << (or one of
 the children for each step) is not supported natively yet

Predicates
==========
# of predicates: 8
```

See Information in `explain` for a description of the DFE-specific sections in the report.

Gremlin `profile` API in Neptune

The Neptune Gremlin `profile` API runs a specified Gremlin traversal, collects various metrics about the run, and produces a profile report as output.

It differs from the TinkerPop .profile() step so as to be able to report information specific to the Neptune engine.

The profile report includes the following information about the query plan:

- The physical operator pipeline
- The index operations for query execution and serialization
- The size of the result

The `profile` API uses an extended version of the HTTP API syntax for query, with `/gremlin/profile` as the endpoint instead of `/gremlin`.

Parameters specific to Neptune Gremlin `profile`

- **profile.results** – `boolean`, allowed values: `TRUE` and `FALSE`, default value: `TRUE`.

  If true, the query results are gathered and displayed as part of the `profile` report. If false, only the result count is displayed.

- **profile.chop** – `int`, default value: 250.

  If non-zero, causes the results string to be truncated at that number of characters. This does not keep all results from being captured. It simply limits the size of the string in the profile report. If set to zero, the string contains all the results.

- **profile.serializer** – `string`, default value: `<null>`.

  If non-null, the gathered results are returned in a serialized response message in the format specified by this parameter. The number of index operations necessary to produce that response message is reported along with the size in bytes to be sent to the client.

  Allowed values are `<null>` or any of the valid MIME type or TinkerPop driver "Serializers" enum values.

  ```
  "application/json" or "GRAPHSON"
  "application/vnd.gremlin-v1.0+json" or "GRAPHSON_V1"
  "application/vnd.gremlin-v1.0+json;types=false" or "GRAPHSON_V1_UNTYPED"
  "application/vnd.gremlin-v2.0+json" or "GRAPHSON_V2"
  "application/vnd.gremlin-v2.0+json;types=false" or "GRAPHSON_V2_UNTYPED"
  "application/vnd.gremlin-v3.0+json" or "GRAPHSON_V3"
  "application/vnd.gremlin-v3.0+json;types=false" or "GRAPHSON_V3_UNTYPED"
  "application/vnd.graphbinary-v1.0" or "GRAPHBINARY_V1"
  ```

- **profile.indexOps** – `boolean`, allowed values: `TRUE` and `FALSE`, default value: `FALSE`.

  If true, shows a detailed report of all index operations that took place during query execution and serialization. Warning: This report can be verbose.

Sample output of Neptune Gremlin `profile`

The following is a sample `profile` query.

AWS CLI

```
aws neptunedata execute-gremlin-profile-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --gremlin-query 'g.V().hasLabel("airport").has("code",
 "AUS").emit().repeat(in().simplePath()).times(2).limit(100)' \
  --serializer "application/vnd.gremlin-v3.0+json"
```

For more information, see execute-gremlin-profile-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_gremlin_profile_query(
    gremlinQuery='g.V().hasLabel("airport").has("code",
 "AUS").emit().repeat(in().simplePath()).times(2).limit(100)',
    serializer='application/vnd.gremlin-v3.0+json'
)

print(response['output'])
```

For AWS SDK examples in other languages like Java, .NET, and more, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/gremlin/profile \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d '{"gremlin":"g.V().hasLabel(\"airport\").has(\"code\",
 \"AUS\").emit().repeat(in().simplePath()).times(2).limit(100)",
 "profile.serializer":"application/vnd.gremlin-v3.0+json"}'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

```
curl -X POST https://your-neptune-endpoint:port/gremlin/profile \
  -d '{"gremlin":"g.V().hasLabel(\"airport\").has(\"code\",
 \"AUS\").emit().repeat(in().simplePath()).times(2).limit(100)",
 "profile.serializer":"application/vnd.gremlin-v3.0+json"}'
```

This query generates the following `profile` report when executed on the air-routes sample graph from the blog post, Let Me Graph That For You – Part 1 – Air Routes.

```
*******************************************************
                Neptune Gremlin Profile
*******************************************************

Query String
==================
g.V().hasLabel("airport").has("code",
 "AUS").emit().repeat(in().simplePath()).times(2).limit(100)

Original Traversal
==================
[GraphStep(vertex,[]), HasStep([~label.eq(airport), code.eq(AUS)]),
 RepeatStep(emit(true),[VertexStep(IN,vertex), PathFilterStep(simple),
 RepeatEndStep],until(loops(2))), RangeGlobalStep(0,100)]

Optimized Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
```

```
            PatternNode[(?1, <code>, "AUS", ?) . project ?1 .],
 {estimatedCardinality=1, indexTime=84, hashJoin=true, joinTime=3, actualTotalOutput=1}
            PatternNode[(?1, <~label>, ?2=<airport>, <~>) . project ask .],
 {estimatedCardinality=3374, indexTime=29, hashJoin=true, joinTime=0,
 actualTotalOutput=61}
            RepeatNode {
                Repeat {
                    PatternNode[(?3, ?5, ?1, ?6) . project ?1,?3 . IsEdgeIdFilter(?6) .
 SimplePathFilter(?1, ?3)) .], {hashJoin=true, estimatedCardinality=50148, indexTime=0,
 joinTime=3}
                }
                Emit {
                    Filter(true)
                }
                LoopsCondition {
                    LoopsFilter([?1, ?3],eq(2))
                }
            }, annotations={repeatMode=BFS, emitFirst=true, untilFirst=false, leftVar=?
1, rightVar=?3}
        }, finishers=[limit(100)], annotations={path=[Vertex(?1):GraphStep,
 Repeat[Vertex(?3):VertexStep]], joinStats=true, optimizationTime=495, maxVarId=7,
 executionTime=323}
    },
    NeptuneTraverserConverterStep
]

Physical Pipeline
=================
NeptuneGraphQueryStep
    |-- StartOp
    |-- JoinGroupOp
        |-- SpoolerOp(100)
        |-- DynamicJoinOp(PatternNode[(?1, <code>, "AUS", ?) . project ?1 .],
 {estimatedCardinality=1, indexTime=84, hashJoin=true})
        |-- SpoolerOp(100)
        |-- DynamicJoinOp(PatternNode[(?1, <~label>, ?2=<airport>, <~>) . project
 ask .], {estimatedCardinality=3374, indexTime=29, hashJoin=true})
        |-- RepeatOp
            |-- <upstream input> (Iteration 0) [visited=1, output=1 (until=0, emit=1),
 next=1]
            |-- BindingSetQueue (Iteration 1) [visited=61, output=61 (until=0,
 emit=61), next=61]
                |-- SpoolerOp(100)
```

```
                |-- DynamicJoinOp(PatternNode[(?3, ?5, ?1, ?6) . project ?
1,?3 . IsEdgeIdFilter(?6) . SimplePathFilter(?1, ?3)) .], {hashJoin=true,
 estimatedCardinality=50148, indexTime=0})
            |-- BindingSetQueue (Iteration 2) [visited=38, output=38 (until=38,
 emit=0), next=0]
                |-- SpoolerOp(100)
                |-- DynamicJoinOp(PatternNode[(?3, ?5, ?1, ?6) . project ?
1,?3 . IsEdgeIdFilter(?6) . SimplePathFilter(?1, ?3)) .], {hashJoin=true,
 estimatedCardinality=50148, indexTime=0})
        |-- LimitOp(100)

Runtime (ms)
============
Query Execution:  392.686
Serialization:   2636.380

Traversal Metrics
=================
Step                                                               Count  Traversers
    Time (ms)    % Dur
---------------------------------------------------------------------------------------------------
NeptuneGraphQueryStep(Vertex)                                        100         100
      314.162    82.78
NeptuneTraverserConverterStep                                        100         100
       65.333    17.22
                                                >TOTAL                     -           -
      379.495        -

Repeat Metrics
==============
Iteration  Visited   Output    Until     Emit     Next
-------------------------------------------------------
        0        1        1        0        1        1
        1       61       61        0       61       61
        2       38       38       38        0        0
-------------------------------------------------------
               100      100       38       62       62

Predicates
==========
# of predicates: 16

WARNING: reverse traversal with no edge label(s) - .in() / .both() may impact query
 performance
```

```
Results
=======
Count: 100
Output: [v[3], v[3600], v[3614], v[4], v[5], v[6], v[7], v[8], v[9], v[10], v[11],
 v[12], v[47], v[49], v[136], v[13], v[15], v[16], v[17], v[18], v[389], v[20], v[21],
 v[22], v[23], v[24], v[25], v[26], v[27], v[28], v[416], v[29], v[30], v[430], v[31],
 v[9...
Response serializer: GRYO_V3D0
Response size (bytes): 23566

Index Operations
================
Query execution:
    # of statement index ops: 3
    # of unique statement index ops: 3
    Duplication ratio: 1.0
    # of terms materialized: 0
Serialization:
    # of statement index ops: 200
    # of unique statement index ops: 140
    Duplication ratio: 1.43
    # of terms materialized: 393
```

In addition to the query plans returned by a call to Neptune `explain`, the `profile` results include runtime statistics around query execution. Each Join operation is tagged with the time it took to perform its join as well as the actual number of solutions that passed through it.

The `profile` output includes the time taken during the core query execution phase, as well as the serialization phase if the `profile.serializer` option was specified.

The breakdown of the index operations performed during each phase is also included at the bottom of the `profile` output.

Note that consecutive runs of the same query may show different results in terms of run-time and index operations because of caching.

For queries using the `repeat()` step, a breakdown of the frontier on each iteration is available if the `repeat()` step was pushed down as part of a `NeptuneGraphQueryStep`.

Differences in `profile` reports when DFE is enabled

When the Neptune DFE alternative query engine is enabled, `profile` output is somewhat different:

Optimized Traversal: This section is similar to the one in `explain` output, but contains additional information. This includes the type of DFE operators that were considered in planning, and the associated worst case and best case cost estimates.

Physical Pipeline: This section captures the operators that are used to execute the query. `DFESubQuery` elements abstract the physical plan that is used by DFE to execute the portion of the plan it is responsible for. The `DFESubQuery` elements are unfolded in the following section where DFE statistics are listed.

DFEQueryEngine Statistics: This section shows up only when at least part of the query is executed by DFE. It outlines various runtime statistics that are specific to DFE, and contains a detailed breakdown of the time spent in the various parts of the query execution, by `DFESubQuery`.

Nested subqueries in different `DFESubQuery` elements are flattened in this section, and unique identifiers are marked with a header that starts with `subQuery=`.

Traversal metrics: This section shows step-level traversal metrics, and when the DFE engine runs all or part of the query, displays metrics for `DFEStep` and/or `NeptuneInterleavingStep`. See Tuning Gremlin queries using `explain` and `profile`.

Sample `profile` output when the Neptune Dataflow engine (DFE) is enabled

When the DFE engine is being used to run Gremlin queries, output of the Gremlin `profile` API is formatted as shown in the example below.

Query:

AWS CLI

```
aws neptunedata execute-gremlin-profile-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --gremlin-query "g.withSideEffect('Neptune#useDFE', true).V().has('code',
 'ATL').out()"
```

For more information, see execute-gremlin-profile-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_gremlin_profile_query(
    gremlinQuery="g.withSideEffect('Neptune#useDFE', true).V().has('code',
 'ATL').out()"
)

print(response['output'])
```

For AWS SDK examples in other languages like Java, .NET, and more, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/gremlin/profile \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d '{"gremlin":"g.withSideEffect('"'"'Neptune#useDFE'"'"',
 true).V().has('"'"'code'"'"', '"'"'ATL'"'"').out()"}'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

```
curl -X POST https://your-neptune-endpoint:port/gremlin/profile \
```

```
 -d '{"gremlin":"g.withSideEffect('"'"'Neptune#useDFE'"'"',
true).V().has('"'"'code'"'"', '"'"'ATL'"'"').out()"}'
```

```
*******************************************************
                    Neptune Gremlin Profile
    *******************************************************

    Query String
    ==================
    g.withSideEffect('Neptune#useDFE', true).V().has('code', 'ATL').out()

    Original Traversal
    ==================
    [GraphStep(vertex,[]), HasStep([code.eq(ATL)]), VertexStep(OUT,vertex)]

    Optimized Traversal
    ===================
    Neptune steps:
    [
        DFEStep(Vertex) {
          DFENode {
            DFEJoinGroupNode[null](
              children=[
                DFEPatternNode((?1, vp://code[419430926], ?4, defaultGraph[526]) .
 project DISTINCT[?1] objectFilters=(in(ATL[452987149]) . ), {rangeCountEstimate=1},
                  opInfo=(type=PipelineJoin,
 cost=(exp=(in=1.00,out=1.00,io=0.00,comp=0.00,mem=0.00),wc=(in=1.00,out=1.00,io=0.00,comp=0.00
                    disc=(type=PipelineScan,
 cost=(exp=(in=1.00,out=1.00,io=0.00,comp=0.00,mem=34.00),wc=(in=1.00,out=1.00,io=0.00,comp=0.0
                DFEPatternNode((?1, ?5, ?6, ?7) . project ALL[?1, ?6] graphFilters=(!=
 defaultGraph[526] . ), {rangeCountEstimate=9223372036854775807})],
              opInfo=[
                OperatorInfoWithAlternative[
                  rec=(type=PipelineJoin,
 cost=(exp=(in=1.00,out=27.76,io=0.00,comp=0.00,mem=0.00),wc=(in=1.00,out=27.76,io=0.00,comp=0.
                    disc=(type=PipelineScan,
 cost=(exp=(in=1.00,out=27.76,io=Infinity,comp=0.00,mem=295147905179352830000.00),wc=(in=1.00,o
                  alt=(type=PipelineScan,
 cost=(exp=(in=1.00,out=27.76,io=Infinity,comp=0.00,mem=295147905179352830000.00),wc=(in=1.00,o
          } [Vertex(?1):GraphStep, Vertex(?6):VertexStep]
        } ,
        NeptuneTraverserConverterDFEStep,
```

```
        DFECleanupStep
    ]


    Physical Pipeline
    =================
    DFEStep
        |-- DFESubQuery1

    DFEQueryEngine Statistics
    =================
    DFESubQuery1

 ##########################################################################################
    # ID # Out #1 # Out #2 # Name                  # Arguments
                                                                                  # Mode #
 Units In # Units Out # Ratio  # Time (ms) #

 ##########################################################################################
    # 0  # 1      # -      # DFESolutionInjection  # solutions=[]
                                                                                # -    # 0
      # 1         # 0.00   # 0.01      #
    #    #        #        #                       # outSchema=[]
                                                                                #      #
      #           #        #           #

 ##########################################################################################
    # 1  # 2      # -      # DFEChunkLocalSubQuery # subQuery=http://aws.amazon.com/
neptune/vocab/v01/dfe/past/graph#089f43e3-4d71-4259-8d19-254ff63cee04/graph_1 # -    #
 1        # 1         # 1.00   # 0.02      #

 ##########################################################################################
    # 2  # 3      # -      # DFEChunkLocalSubQuery # subQuery=http://aws.amazon.com/
neptune/vocab/v01/dfe/past/graph#089f43e3-4d71-4259-8d19-254ff63cee04/graph_2 # -    #
 1        # 242       # 242.00 # 0.02      #

 ##########################################################################################
    # 3  # 4      # -      # DFEMergeChunks        # -
                                                                                # -    # 242
      # 242       # 1.00   # 0.01      #

 ##########################################################################################
```

```
    # 4  # -      # -      # DFEDrain              # -
                                                                                    # -    # 242
      # 0         # 0.00   # 0.01      #

 ##########################################################################################


    subQuery=http://aws.amazon.com/neptune/vocab/v01/dfe/past/
graph#089f43e3-4d71-4259-8d19-254ff63cee04/graph_1

 ##########################################################################################
    # ID # Out #1 # Out #2 # Name                 # Arguments
                         # Mode # Units In # Units Out # Ratio # Time (ms) #

 ##########################################################################################
    # 0  # 1      # -      # DFEPipelineScan      # pattern=Node(?1) with property
 'code' as ?4 and label 'ALL' # -    # 0        # 1         # 0.00  # 0.22      #
    #    #        #        #                     # inlineFilters=[(?4 IN ["ATL"])]
                         #      #          #           #       #           #
    #    #        #        #                     # patternEstimate=1
                         #      #          #           #       #           #

 ##########################################################################################
    # 1  # 2      # -      # DFEMergeChunks       # -
                         # -    # 1        # 1         # 1.00  # 0.02      #

 ##########################################################################################
    # 2  # 4      # -      # DFERelationalJoin    # joinVars=[]
                         # -    # 2        # 1         # 0.50  # 0.09      #

 ##########################################################################################
    # 3  # 2      # -      # DFESolutionInjection # solutions=[]
                         # -    # 0        # 1         # 0.00  # 0.01      #
    #    #        #        #                     # outSchema=[]
                         #      #          #           #       #           #

 ##########################################################################################
    # 4  # -      # -      # DFEDrain             # -
                         # -    # 1        # 0         # 0.00  # 0.01      #

 ##########################################################################################
```

```
    subQuery=http://aws.amazon.com/neptune/vocab/v01/dfe/past/
graph#089f43e3-4d71-4259-8d19-254ff63cee04/graph_2

 ##########################################################################################
    # ID # Out #1 # Out #2 # Name                 # Arguments
 # Mode # Units In # Units Out # Ratio  # Time (ms) #

 ##########################################################################################
    # 0  # 1      # -      # DFESolutionInjection # solutions=[]
 # -    # 0        # 1         # 0.00   # 0.01      #
    #    #        #        #                     # outSchema=[?1]
 #      #          #           #        #           #

 ##########################################################################################
    # 1  # 2      # 3      # DFETee               # -
 # -    # 1        # 2         # 2.00   # 0.01      #

 ##########################################################################################
    # 2  # 4      # -      # DFEDistinctColumn    # column=?1
 # -    # 1        # 1         # 1.00   # 0.21      #
    #    #        #        #                     # ordered=false
 #      #          #           #        #           #

 ##########################################################################################
    # 3  # 5      # -      # DFEHashIndexBuild    # vars=[?1]
 # -    # 1        # 1         # 1.00   # 0.03      #

 ##########################################################################################
    # 4  # 5      # -      # DFEPipelineJoin      # pattern=Edge((?1)-[?7:?5]->(?6))
 # -    # 1        # 242       # 242.00 # 0.51      #
    #    #        #        #                     # constraints=[]
 #      #          #           #        #           #
    #    #        #        #                     # patternEstimate=9223372036854775807
 #      #          #           #        #           #

 ##########################################################################################
    # 5  # 6      # 7      # DFESync              # -
 # -    # 243      # 243       # 1.00   # 0.02      #

 ##########################################################################################
    # 6  # 8      # -      # DFEForwardValue      # -
 # -    # 1        # 1         # 1.00   # 0.01      #

 ##########################################################################################
```

```
   # 7  # 8      # -      # DFEForwardValue      # -
# -    # 242      # 242       # 1.00   # 0.02      #

##########################################################################################
   # 8  # 9      # -      # DFEHashIndexJoin     # -
# -    # 243      # 242       # 1.00   # 0.31      #

##########################################################################################
   # 9  # -      # -      # DFEDrain             # -
# -    # 242      # 0         # 0.00   # 0.01      #

##########################################################################################


   Runtime (ms)
   ============
   Query Execution: 11.744

   Traversal Metrics
   =================
   Step                                                               Count
Traversers       Time (ms)    % Dur

------------------------------------------------------------------------------------------
   DFEStep(Vertex)                                                      242
242          10.849    95.48
   NeptuneTraverserConverterDFEStep                                     242
242           0.514     4.52
                                                     >TOTAL                     -
-          11.363        -

   Predicates
   ==========
   # of predicates: 18

   Results
   =======
   Count: 242


   Index Operations
   ================
   Query execution:
       # of statement index ops: 0
```

```
        # of terms materialized: 0
```

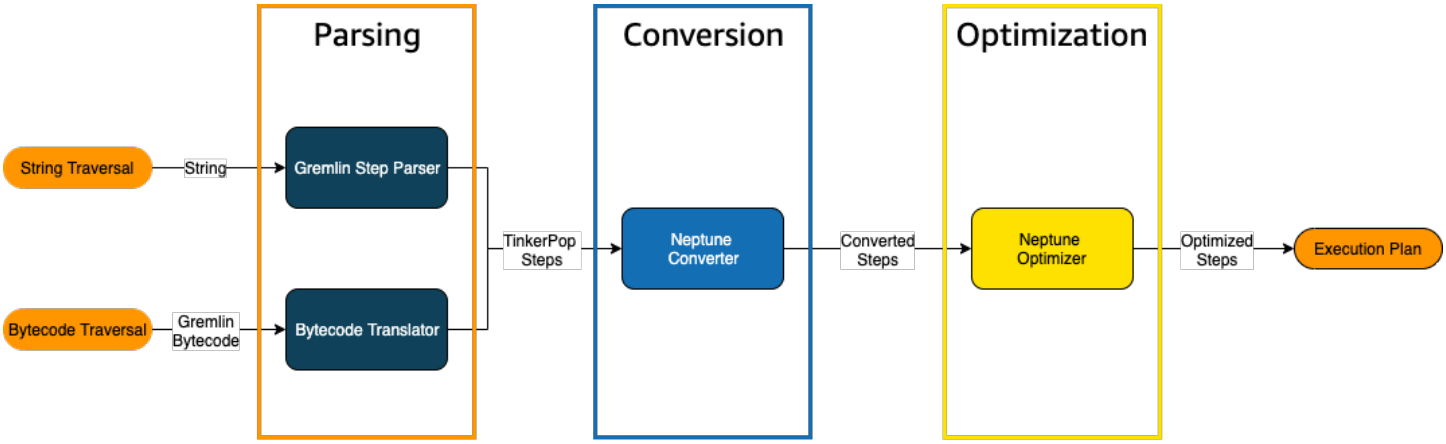
Tuning Gremlin queries using `explain` and `profile`

You can often tune your Gremlin queries in Amazon Neptune to get better performance, using the information available to you in the reports you get from the Neptune explain and profile APIs. To do so, it helps to understand how Neptune processes Gremlin traversals.

> **Important**
>
> A change was made in TinkerPop version 3.4.11 that improves correctness of how queries are processed, but for the moment can sometimes seriously impact query performance. For example, a query of this sort may run significantly slower:
>
> ```
> g.V().hasLabel('airport').
>  order().
>  by(out().count(),desc).
>  limit(10).
>  out()
> ```
>
> The vertices after the limit step are now fetched in a non-optimal way beause of the TinkerPop 3.4.11 change. To avoid this, you can modify the query by adding the barrier() step at any point after the `order().by()`. For example:
>
> ```
> g.V().hasLabel('airport').
>  order().
>  by(out().count(),desc).
>  limit(10).
>  barrier().
>  out()
> ```
>
> TinkerPop 3.4.11 was enabled in Neptune engine version 1.0.5.0.

Understanding Gremlin traversal processing in Neptune

When a Gremlin traversal is sent to Neptune, there are three main processes that transform the traversal into an underlying execution plan for the engine to execute. These are parsing, conversion, and optimization:



The traversal parsing process

The first step in processing a traversal is to parse it into a common language. In Neptune, that common language is the set of TinkerPop steps that are part of the TinkerPop API. Each of these steps represents a unit of computation within the traversal.

You can send a Gremlin traversal to Neptune either as a string or as bytecode. The REST endpoint and the Java client driver `submit()` method send traversals as strings, as in this example:

```
client.submit("g.V()")
```

Applications and language drivers using Gremlin language variants (GLV) send traversals in bytecode.

The traversal conversion process

The second step in processing a traversal is to convert its TinkerPop steps into a set of converted and non-converted Neptune steps. Most steps in the Apache TinkerPop Gremlin query language are converted to Neptune-specific steps that are optimized to run on the underlying Neptune engine. When a TinkerPop step without a Neptune equivalent is encountered in a traversal, that step and all subsequent steps in the traversal are processed by the TinkerPop query engine.

For more information about what steps can be converted under what circumstances, see Gremlin step support.

The traversal optimization process

The final step in traversal processing is to run the series of converted and non-converted steps through the optimizer, to try to determine the best execution plan. The output of this optimization is the execution plan that the Neptune engine processes.

Using the Neptune Gremlin `explain` API to tune queries

The Neptune explain API is not the same as the Gremlin `explain()` step. It returns the final execution plan that the Neptune engine would process when executing the query. Because it does not perform any processing, it returns the same plan regardless of the parameters used, and its output contains no statistics about actual execution.

Consider the following simple traversal that finds all the airport vertices for Anchorage:

```
g.V().has('code','ANC')
```

There are two ways you can run this traversal through the Neptune `explain` API. The first way is to make a REST call to the explain endpoint, like this:

```
curl -X POST https://your-neptune-endpoint:port/gremlin/explain -d
 '{"gremlin":"g.V().has('code','ANC')"}'
```

The resulting `explain` API output describes Neptune's execution plan for the traversal. As you can see in the image below, the plan includes each of the 3 steps in the processing pipeline:

```
Explain

*******************************************************
                Neptune Gremlin Explain
*******************************************************

Query String
============

g.V().has('code','ANC')

Original Traversal
==================
[GraphStep(vertex,[]), HasStep([code.eq(ANC)])]

Converted Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
            PatternNode[(?1, <~label>, ?2, <->) . project distinct ?1 .]
            PatternNode[(?1, <code>, "ANC", ?) . project ask .]
        }, annotations={path=[Vertex(?1):GraphStep], maxVarId=3}
    },
    NeptuneTraverserConverterStep
]

Optimized Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
            PatternNode[(?1, <code>, "ANC", ?) . project ?1 .], {estimatedCardinality=1}
        }, annotations={path=[Vertex(?1):GraphStep], maxVarId=3}
    },
    NeptuneTraverserConverterStep
]

Predicates
==========
# of predicates: 22
```

Parsing

Conversion

Optimization

Tuning a traversal by looking at steps that are not converted

One of the first things to look for in the Neptune `explain` API output is for Gremlin steps that are not converted to Neptune native steps. In a query plan, when a step is encountered that cannot be converted to a Neptune native step, it and all subsequent steps in the plan are processed by the Gremlin server.

In the example above, all steps in the traversal were converted. Let's examine `explain` API output for this traversal:

```
g.V().has('code','ANC').out().choose(hasLabel('airport'), values('code'), constant('Not
 an airport'))
```

As you can see in the image below, Neptune could not convert the `choose()` step:

Explain

```
*******************************************************
                Neptune Gremlin Explain
*******************************************************

Query String
============

g.V().has('code','ANC').out().choose(hasLabel('airport'), values('code'), constant('Not an airport'))

Original Traversal
==================
[GraphStep(vertex,[]), HasStep([code.eq(ANC)]), VertexStep(OUT,vertex), ChooseStep([HasStep([~label.eq(airport)]), HasNex

Converted Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
            PatternNode[(?1, <~label>, ?2, <~>) . project distinct ?1 .]
            PatternNode[(?1, <code>, "ANC", ?) . project ask .]
            PatternNode[(?1, ?5, ?3, ?6) . project ?1,?3 . IsEdgeIdFilter(?6) .]
            PatternNode[(?3, <~label>, ?4, <~>) . project ask .]
        }, annotations={path=[Vertex(?1):GraphStep, Vertex(?3):VertexStep], maxVarId=7}
    },
    NeptuneTraverserConverterStep
]
+ not converted into Neptune steps: [ChooseStep([HasStep([~label.eq(airport)]), HasNextStep],[[(eq(true)), [PropertiesSte

Optimized Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
            PatternNode[(?1, <code>, "ANC", ?) . project ?1 .], {estimatedCardinality=1}
            PatternNode[(?1, ?5, ?3, ?6) . project ?1,?3 . IsEdgeIdFilter(?6) .], {estimatedCardinality=INFINITY}
        }, annotations={path=[Vertex(?1):GraphStep, Vertex(?3):VertexStep], maxVarId=7}
    },
    NeptuneTraverserConverterStep
]
+ not converted into Neptune steps: [ChooseStep([NeptuneHasStep([~label.eq(airport)]), HasNextStep],[[(eq(true)), [Proper

WARNING: >> ChooseStep([NeptuneHasStep([~label.eq(airport)]), HasNextStep],[[(eq(true)), [PropertiesStep([code],value), E

Predicates
==========
# of predicates: 26
```

There are several things you could do to tune the performance of the traversal. The first would be to rewrite it in such a way as to eliminate the step that could not be converted. Another would be to move the step to the end of the traversal so that all other steps can be converted to native ones.

A query plan with steps that are not converted does not always need to be tuned. If the steps that cannot be converted are at the end of the traversal, and are related to how output is formatted rather than how the graph is traversed, they may have little effect on performance.

Another thing to look for when examining output from the Neptune `explain` API is steps that do not use indexes. The following traversal finds all airports with flights that land in Anchorage:

```
g.V().has('code','ANC').in().values('code')
```

Output from the explain API for this traversal is:

```
*******************************************************
                Neptune Gremlin Explain
*******************************************************

Query String
============

g.V().has('code','ANC').in().values('code')

Original Traversal
==================
[GraphStep(vertex,[]), HasStep([code.eq(ANC)]), VertexStep(IN,vertex),
 PropertiesStep([code],value)]

Converted Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(PropertyValue) {
        JoinGroupNode {
            PatternNode[(?1, <~label>, ?2, <~>) . project distinct ?1 .]
            PatternNode[(?1, <code>, "ANC", ?) . project ask .]
            PatternNode[(?3, ?5, ?1, ?6) . project ?1,?3 . IsEdgeIdFilter(?6) .]
            PatternNode[(?3, <~label>, ?4, <~>) . project ask .]
            PatternNode[(?3, ?7, ?8, <~>) . project ?3,?8 . ContainsFilter(?7 in
 (<code>)) .]
        }, annotations={path=[Vertex(?1):GraphStep, Vertex(?3):VertexStep,
 PropertyValue(?8):PropertiesStep], maxVarId=9}
    },
    NeptuneTraverserConverterStep
]

Optimized Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(PropertyValue) {
        JoinGroupNode {
```

```
            PatternNode[(?1, <code>, "ANC", ?) . project ?1 .],
 {estimatedCardinality=1}
            PatternNode[(?3, ?5, ?1, ?6) . project ?1,?3 . IsEdgeIdFilter(?6) .],
 {estimatedCardinality=INFINITY}
            PatternNode[(?3, ?7=<code>, ?8, <~>) . project ?3,?8 .],
 {estimatedCardinality=7564}
        }, annotations={path=[Vertex(?1):GraphStep, Vertex(?3):VertexStep,
 PropertyValue(?8):PropertiesStep], maxVarId=9}
    },
    NeptuneTraverserConverterStep
]

Predicates
==========
# of predicates: 26

WARNING: reverse traversal with no edge label(s) - .in() / .both() may impact query
 performance
```

The `WARNING` message at the bottom of the output occurs because the `in()` step in the traversal cannot be handled using one of the 3 indexes that Neptune maintains (see How Statements Are Indexed in Neptune and Gremlin statements in Neptune). Because the `in()` step contains no edge filter, it cannot be resolved using the `SPOG`, `POGS` or `GPSO` index. Instead, Neptune must perform a union scan to find the requested vertices, which is much less efficient.

There are two ways to tune the traversal in this situation. The first is to add one or more filtering criteria to the `in()` step so that an indexed lookup can be used to resolve the query. For the example above, this might be:

```
g.V().has('code','ANC').in('route').values('code')
```

Output from the Neptune `explain` API for the revised traversal no longer contains the `WARNING` message:

```
*******************************************************
                Neptune Gremlin Explain
*******************************************************

Query String
============

g.V().has('code','ANC').in('route').values('code')
```

```
Original Traversal
==================
[GraphStep(vertex,[]), HasStep([code.eq(ANC)]), VertexStep(IN,[route],vertex),
 PropertiesStep([code],value)]

Converted Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(PropertyValue) {
        JoinGroupNode {
            PatternNode[(?1, <~label>, ?2, <~>) . project distinct ?1 .]
            PatternNode[(?1, <code>, "ANC", ?) . project ask .]
            PatternNode[(?3, ?5, ?1, ?6) . project ?1,?3 . IsEdgeIdFilter(?6) .
 ContainsFilter(?5 in (<route>)) .]
            PatternNode[(?3, <~label>, ?4, <~>) . project ask .]
            PatternNode[(?3, ?7, ?8, <~>) . project ?3,?8 . ContainsFilter(?7 in
 (<code>)) .]
        }, annotations={path=[Vertex(?1):GraphStep, Vertex(?3):VertexStep,
 PropertyValue(?8):PropertiesStep], maxVarId=9}
    },
    NeptuneTraverserConverterStep
]

Optimized Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(PropertyValue) {
        JoinGroupNode {
            PatternNode[(?1, <code>, "ANC", ?) . project ?1 .],
 {estimatedCardinality=1}
            PatternNode[(?3, ?5=<route>, ?1, ?6) . project ?1,?3 . IsEdgeIdFilter(?
6) .], {estimatedCardinality=32042}
            PatternNode[(?3, ?7=<code>, ?8, <~>) . project ?3,?8 .],
 {estimatedCardinality=7564}
        }, annotations={path=[Vertex(?1):GraphStep, Vertex(?3):VertexStep,
 PropertyValue(?8):PropertiesStep], maxVarId=9}
    },
    NeptuneTraverserConverterStep
]

Predicates
```

```
==========
# of predicates: 26
```

Another option if you are running many traversals of this kind is to run them in a Neptune DB cluster that has the optional OSGP index enabled (see Enabling an OSGP Index). Enabling an OSGP index has drawbacks:

- It must be enabled in a DB cluster before any data is loaded.
- Insertion rates for vertices and edges may slow by up to 23%.
- Storage usage will increase by around 20%.
- Read queries that scatter requests across all indexes may have increased latencies.

Having an OSGP index makes a lot of sense for a restricted set of query patterns, but unless you are running those frequently, it is usually preferable to try to ensure that the traversals you write can be resolved using the three primary indexes.

Using a large number of predicates

Neptune treats each edge label and each distinct vertex or edge property name in your graph as a predicate, and is designed by default to work with a relatively low number of distinct predicates. When you have more than a few thousand predicates in your graph data, performance can degrade.

Neptune `explain` output will warn you if this is the case:

```
Predicates
==========
# of predicates: 9549
WARNING: high predicate count (# of distinct property names and edge labels)
```

If it is not convenient to rework your data model to reduce the number of labels and properties, and therefore the number of predicates, the best way to tune traversals is to run them in a DB cluster that has the OSGP index enabled, as discussed above.

Using the Neptune Gremlin `profile` API to tune traversals

The Neptune `profile` API is quite different from the Gremlin `profile()` step. Like the `explain` API, its output includes the query plan that the Neptune engine uses when executing the traversal.

In addition, the `profile` output includes actual execution statistics for the traversal, given how its parameters are set.

Again, take the simple traversal that finds all airport vertices for Anchorage:

```
g.V().has('code','ANC')
```

As with the `explain` API, you can invoke the `profile` API using a REST call:

```
curl -X POST https://your-neptune-endpoint:port/gremlin/profile -d
 '{"gremlin":"g.V().has('code','ANC')"}'
```

The resulting `profile` API output contains both Neptune's execution plan for the traversal and statistics about the plan's execution, as you can see in this image:

```
Profile

*******************************************************
                 Neptune Gremlin Profile
*******************************************************

Query String
==================

g.V().has('code','ANC')

Original Traversal
==================
[GraphStep(vertex,[]), HasStep([code.eq(ANC)])]

Optimized Traversal
===================
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
            PatternNode[(?1, <code>, "ANC", ?) . project ?1 .], {estimatedCardinality=1, indexTime=0, joinTime=0, numSear
        }, annotations={path=[Vertex(?1):GraphStep], joinStats=true, optimizationTime=1, maxVarId=3, executionTime=3}
    },
    NeptuneTraverserConverterStep
]

Physical Pipeline
=================
NeptuneGraphQueryStep
    |-- StartOp
    |-- JoinGroupOp
        |-- SpoolerOp(1000)
        |-- DynamicJoinOp(PatternNode[(?1, <code>, "ANC", ?) . project ?1 .], {estimatedCardinality=1})

Runtime (ms)
============
Query Execution: 5.096

Traversal Metrics
=================
Step                                                               Count  Traversers       Time (ms)    % Dur
---------------------------------------------------------------------------------------------------------------
NeptuneGraphQueryStep(Vertex)                                          1           1           0.956    90.62
NeptuneTraverserConverterStep                                          1           1           0.099     9.38
                                            >TOTAL                     -           -           1.055        -

Predicates
==========
# of predicates: 26

Results
=======
Count: 1
Output: [v[2]]


Index Operations
================
Query execution:
    # of statement index ops: 1
    # of unique statement index ops: 1
    Duplication ratio: 1.0
    # of terms materialized: 0
```

Execution Plan

Pipeline

Statistics and Results

In `profile` output, the execution plan section only contains the final execution plan for the traversal, not the intermediate steps. The pipeline section contains the physical pipeline operations that were performed as well as the actual time (in milliseconds) that traversal execution took.

The runtime metric is extremely helpful in comparing the times that two different versions of a traversal take as you are optimizing them.

> **Note**
>
> The initial runtime of a traversal is generally longer than subsequent runtimes, because the first one causes the relevant data to be cached.

The third section of the `profile` output contains execution statistics and the results of the traversal. To see how this information can be useful in tuning a traversal, consider the following traversal, which finds every airport whose name begins with "Anchora", and all the airports reachable in two hops from those airports, returning airport codes, flight routes, and distances:

```
%%gremlin profile

g.withSideEffect("Neptune#fts.endpoint", "{your-OpenSearch-endpoint-URL").
    V().has("city", "Neptune#fts Anchora~").
    repeat(outE('route').inV().simplePath()).times(2).
    project('Destination', 'Route').
        by('code').
        by(path().by('code').by('dist'))
```

Traversal metrics in Neptune `profile` API output

The first set of metrics that is available in all `profile` output is the traversal metrics. These are similar to the Gremlin `profile()` step metrics, with a few differences:

```
Traversal Metrics
=================
Step                                                               Count  Traversers
    Time (ms)    % Dur
-------------------------------------------------------------------------------------------------
NeptuneGraphQueryStep(Vertex)                                       3856        3856
       91.701     9.09
NeptuneTraverserConverterStep                                       3856        3856
       38.787     3.84
ProjectStep([Destination, Route],[value(code), ...                  3856        3856
      878.786    87.07
  PathStep([value(code), value(dist)])                              3856        3856
      601.359
```

```
                                                >TOTAL                     -           -
    1009.274        -
```

The first column of the traversal-metrics table lists the steps executed by the traversal. The first two steps are generally the Neptune-specific steps, `NeptuneGraphQueryStep` and `NeptuneTraverserConverterStep`.

`NeptuneGraphQueryStep` represents the execution time for the entire portion of the traversal that could be converted and executed natively by the Neptune engine.

`NeptuneTraverserConverterStep` represents the process of converting the output of those converted steps into TinkerPop traversers which allow steps that could not be converted steps, if any, to be processed, or to return the results in a TinkerPop-compatible format.

In the example above, we have several non-converted steps, so we see that each of these TinkerPop steps (`ProjectStep`, `PathStep`) then appears as a row in the table.

The second column in the table, `Count`, reports the number of *represented* traversers that passed through the step, while the third column, `Traversers`, reports the number of traversers which passed through that step, as explained in the TinkerPop profile step documentation.

In our example there are 3,856 vertices and 3,856 traversers returned by the `NeptuneGraphQueryStep`, and these numbers remain the same throughout the remaining processing because `ProjectStep` and `PathStep` are formatting the results, not filtering them.

> **Note**
>
> Unlike TinkerPop, the Neptune engine does not optimize performance by *bulking* in its `NeptuneGraphQueryStep` and `NeptuneTraverserConverterStep` steps. Bulking is the TinkerPop operation that combines traversers on the same vertex to reduce operational overhead, and that is what causes the `Count` and `Traversers` numbers to differ. Because bulking only occurs in steps that Neptune delegates to TinkerPop, and not in steps that Neptune handles natively, the `Count` and `Traverser` columns seldom differ.

The Time column reports the number of milliseconds that the step took, and the the `% Dur` column reports what percent of the total processing time the step took. These are the metrics that tell you where to focus your tuning efforts by showing the steps that took the most time.

Index operation metrics in Neptune `profile` API output

Another set of metrics in the output of the Neptune profile API is the index operations:

```
Index Operations
================
Query execution:
    # of statement index ops: 23191
    # of unique statement index ops: 5960
    Duplication ratio: 3.89
    # of terms materialized: 0
```

These report:

- The total number of index lookups.
- The number of unique index lookups performed.
- The ratio of total index lookups to unique ones. A lower ratio indicates less redundancy.
- The number of terms materialized from the term dictionary.

Repeat metrics in Neptune `profile` API output

If your traversal uses a `repeat()` step as in the example above, then a section containing repeat metrics appears in the `profile` output:

```
Repeat Metrics
==============
Iteration  Visited   Output    Until     Emit     Next
-------------------------------------------------------
        0        2        0        0        0        2
        1       53        0        0        0       53
        2     3856     3856     3856        0        0
-------------------------------------------------------
              3911     3856     3856        0       55
```

These report:

- The loop count for a row (the `Iteration` column).
- The number of elements visited by the loop (the `Visited` column).
- The number of elements output by the loop (the `Output` column).

- The last element output by the loop (the `Until` column).
- The number of elements emitted by the loop (the `Emit` column).
- The number of elements passed from the loop to the subsequent loop (the `Next` column).

These repeat metrics are very helpful in understanding the branching factor of your traversal, to get a feeling for how much work is being done by the database. You can use these numbers to diagnose performance problems, especially when the same traversal performs dramatically differently with different parameters.

Full-text search metrics in Neptune `profile` API output

When a traversal uses a full-text search lookup, as in the example above, then a section containing the full-text search (FTS) metrics appears in the `profile` output:

```
FTS Metrics
==============
SearchNode[(idVar=?1, query=Anchora~, field=city) . project ?1 .],
    {endpoint=your-OpenSearch-endpoint-URL, incomingSolutionsThreshold=1000,
 estimatedCardinality=INFINITY,
    remoteCallTimeSummary=[total=65, avg=32.500000, max=37, min=28],
    remoteCallTime=65, remoteCalls=2, joinTime=0, indexTime=0, remoteResults=2}

    2 result(s) produced from SearchNode above
```

This shows the query sent to the ElasticSearch (ES) cluster and reports several metrics about the interaction with ElasticSearch that can help you pinpoint performance problems relating to full-text search:

- Summary information about the calls into the ElasticSearch index:
  - The total number of milliseconds required by all remoteCalls to satisfy the query (`total`).
  - The average number of milliseconds spent in a remoteCall (`avg`).
  - The minimum number of milliseconds spent in a remoteCall (`min`).
  - The maximum number of milliseconds spent in a remoteCall (`max`).
- Total time consumed by remoteCalls to ElasticSearch (`remoteCallTime`).
- The number of remoteCalls made to ElasticSearch (`remoteCalls`).
- The number of milliseconds spent in joins of ElasticSearch results (`joinTime`).
- The number of milliseconds spent in index lookups (`indexTime`).

- The total number of results returned by ElasticSearch (`remoteResults`).

Native Gremlin step support in Amazon Neptune

The Amazon Neptune engine does not currently have full native support for all Gremlin steps, as explained in Tuning Gremlin queries. Current support falls into four categories:

- Gremlin steps that can always be converted to native Neptune engine operations
- Gremlin steps that can be converted to native Neptune engine operations in some cases
- Gremlin steps that are never converted to native Neptune engine operations
- Gremlin steps that are not supported in Neptune at all

> **Note**
>
> In Neptune engine release 1.5.0.0 and later, DFE natively supports most Gremlin steps. If you use engine release 1.5.0.0 or later, see Neptune engine release 1.5.0.0 and later for the current DFE step coverage. The native-engine step support described below applies to workloads that do not use DFE.

> **Neptune 1.5.0.0: step behavior changes**
>
> TinkerPop 3.8.0 changed the behavior of several steps. Notes throughout this page describe each change next to the affected entry. The highlights are:
>
> - The `store()` step and the `aggregate(Scope)` overload (for example, `aggregate(local)`) have been removed. Use `aggregate(global)` (the default) for global accumulation, or `local(aggregate())` if you need local-scope aggregation.
> - The `choose()` step now matches only the first matching option, traversers that do not match any option are passed through automatically, and the `option(Traversal, v)` variant has been removed. A new `Pick.unproductive` token is available for handling traversers that do not produce a result.
> - The `valueMap()`, `propertyMap()`, `groupCount()`, `dedup()`, `sack()`, `sample()`, and `aggregate()` steps now throw an error if more than one `by()` modulator is supplied.
> - Inside `repeat()`, `cap()` and `inject()` are no longer allowed. Test any `repeat()` that uses these steps before upgrading to 3.8.0.

- The `none()` step has been renamed to `discard()`. A new predicate-filter step, `none(P)`, reuses the none name and drops traversers that match the predicate.

For more information, see the TinkerPop 3.8.0 upgrade documentation on the Apache TinkerPop website.

Gremlin steps that can always be converted to native Neptune engine operations

Many Gremlin steps can be converted to native Neptune engine operations as long as they meet the following conditions:

- They are not preceded in the query by a step that cannot be converted.
- Their parent step, if any, can be converted,
- All their child traversals, if any, can be converted.

The following Gremlin steps are always converted to native Neptune engine operations if they meet those conditions:

- and( )
- as( )
- count( )
- E( )
- emit( )
- explain( )
- group( )
- groupCount( )
- identity( )
- is( )
- key( )
- label( )
- limit( )
- local( )

- loops( )
- not( )
- or( )
- profile( )
- properties( )
- subgraph( )
- until( )
- V( )
- value( )
- valueMap( )
- values( )

Gremlin steps that can be converted to native Neptune engine operations in some cases

Some Gremlin steps can be converted to native Neptune engine operations in some situations but not in others:

- addE( ) – The `addE()` step can generally be converted to a native Neptune engine operation, unless it is immediately followed by a `property()` step containing a traversal as a key.
- addV( ) – The `addV()` step can generally be converted to a native Neptune engine operation, unless it is immediately followed by a `property()` step containing a traversal as a key, or unless multiple labels are assigned.
- aggregate( ) – The `aggregate()` step can generally be converted to a native Neptune engine operation, unless the step is used in a child traversal or sub-traversal, or unless the value being stored is something other than a vertex, edge, id, label or property value.

> **Neptune 1.5.0.0: store() and aggregate(Scope) removed**
>
> The `store()` step and the `aggregate(Scope)` overload (for example, `aggregate(local)`) have been removed in TinkerPop 3.8.0. Existing Neptune queries that use the default `aggregate()` form continue to work. For migration guidance, see the page-level note at the top of this page.

In example below, `aggregate()` is not converted because it is being used in a child traversal:

```
g.V().has('code','ANC').as('a')
     .project('flights').by(select('a')
     .outE().aggregate('x'))
```

In this example, aggregate() is not converted because what is stored is the `min()` of a value:

```
g.V().has('code','ANC').outE().aggregate('x').by(values('dist').min())
```

- barrier() – The `barrier()` step can generally be converted to a native Neptune engine operation, unless the step following it is not converted.
- cap() – The only case in which the `cap()` step is converted is when it is combined with the `unfold()` step to return an unfolded version of an aggregate of vertex, edge, id, or poperty values. In this example, `cap()` will be converted because it is followed by `.unfold()`:

```
g.V().has('airport','country','IE').aggregate('airport').limit(2)
     .cap('airport').unfold()
```

However, if you remove the `.unfold()`, `cap()` will not be converted:

```
g.V().has('airport','country','IE').aggregate('airport').limit(2)
     .cap('airport')
```

- coalesce() – The only case where the `coalesce()` step is converted is when it follows the Upsert pattern recommended on the TinkerPop recipes page. Other coalesce() patterns are not allowed. Conversion is limited to the case where all child traversals can be converted, they all produce the same type as output (vertex, edge, id, value, key, or label), they all traverse to a new element, and they do not contain the `repeat()` step.
- constant() – The constant() step is currently only converted if it is used within a `sack().by()` part of a traversal to assign a constant value, like this:

```
g.V().has('code','ANC').sack(assign).by(constant(10)).out().limit(2)
```

- cyclicPath() – The `cyclicPath()` step can generally be converted to a native Neptune engine operation, unless the step is used with `by()`, `from()`, or `to()` modulators. In the following queries, for example, `cyclicPath()` is not converted:

```
g.V().has('code','ANC').as('a').out().out().cyclicPath().by('code')
```

```
g.V().has('code','ANC').as('a').out().out().cyclicPath().from('a')
g.V().has('code','ANC').as('a').out().out().cyclicPath().to('a')
```

- drop() – The `drop()` step can generally be converted to a native Neptune engine operation, unless the step is used inside a `sideEffect()` or `optional()` step.
- fold() – There are only two situations where the fold() step can be converted, namely when it is used in the Upsert pattern recommended on the TinkerPop recipes page, and when it is used in a `group().by()` context like this:

```
g.V().has('code','ANC').out().group().by().by(values('code', 'city').fold())
```

- has() – The `has()` step can generally be converted to a native Neptune engine operation provided queries with `T` use the predicate `P.eq`, `P.neq` or `P.contains`. Expect variations of `has()` that imply those instances of `P` to convert to native as well, such as `hasId('id1234')` which is equivalent to `has(eq, T.id, 'id1234')`.
- id() – The `id()` step is converted unless it is used on a property, like this:

```
g.V().has('code','ANC').properties('code').id()
```

- mergeE() – The `mergeE()` step can be converted to a native Neptune engine operation if the parameters (the merge condition, the `onCreate` and `onMatch`) are constant (either `null`, a constant `Map`, or `select()` of a `Map`). All examples in upserting edges can be converted.
- mergeV() – The mergeV() step can be converted to a native Neptune engine operation if the parameters (the merge condition, the `onCreate` and `onMatch`) are constant (either `null`, a constant `Map`, or `select()` of a `Map`). All examples in upserting vertices can be converted.
- order() – The `order()` step can generally be converted to a native Neptune engine operation, unless one of the following is true:
  - The `order()` step is within a nested child traversal, like this:

    ```
    g.V().has('code','ANC').where(V().out().order().by(id))
    ```

  - Local ordering is being used, as for example with `order(local)`.
  - A custom comparator is being used in the `by()` modulation to order by. An example is this use of `sack()`:

    ```
    g.withSack(0).
      V().has('code','ANC').
          repeat(outE().sack(sum).by('dist').inV()).times(2).limit(10).
    ```

```
        order().by(sack())
```

  - There are multiple orderings on the same element.

- project( ) – The `project()` step can generally be converted to a native Neptune engine operation, unless the number of `by()` statements following the `project()` does not match the number of labels specified, as here:

```
g.V().has('code','ANC').project('x', 'y').by(id)
```

- range( ) – The `range()` step is only converted when the lower end of the range in question is zero (for example, `range(0,3)`).
- repeat( ) – The `repeat()` step can generally be converted to a native Neptune engine operation, unless it is nested within another `repeat()` step, like this:

```
g.V().has('code','ANC').repeat(out().repeat(out()).times(2)).times(2)
```

> **Neptune 1.5.0.0: repeat() restrictions**
>
> Starting with TinkerPop 3.8.0, `cap()` and `inject()` are no longer allowed inside a `repeat()` traversal. Retest any `repeat()` traversal that uses `cap()` or `inject()` before you upgrade.

- sack( ) – The `sack()` step can generally be converted to a native Neptune engine operation, except in the following cases:
  - If a non-numeric sack operator is being used.
  - If a numeric sack operator other than `+`, `-`, `mult`, `div`, `min` and `max` is being used.
  - If `sack()` is used inside a `where()` step to filter based on a sack value, as here:

```
g.V().has('code','ANC').sack(assign).by(values('code')).where(sack().is('ANC'))
```

- sum( ) – The `sum()` step can generally be converted to a native Neptune engine operation, but not when used to calculate a global summation, like this:

```
g.V().has('code','ANC').outE('routes').values('dist').sum()
```

- union( ) – The `union()` step can be converted to a native Neptune engine operation as long as it is the last step in the query aside from the terminal step.

- unfold( ) – The `unfold( )` step can only be converted to a native Neptune engine operation when it is used in the Upsert pattern recommended on the TinkerPop recipes page, and when it is used together with `cap( )` like this:

```
g.V().has('airport','country','IE').aggregate('airport').limit(2)
     .cap('airport').unfold()
```

- where( ) – The `where( )` step can generally be converted to a native Neptune engine operation, except in the following cases:
  - When by() modulations are used, like this:

    ```
    g.V().hasLabel('airport').as('a')
         .where(gt('a')).by('runways')
    ```

  - When comparison operators other than `eq`, `neq`, `within`, and `without` are used.
  - When user-supplied aggregations are used.

Gremlin steps that are never converted to native Neptune engine operations

The following Gremlin steps are supported in Neptune but are never converted to native Neptune engine operations. Instead, they are executed by the Gremlin server.

> **Neptune 1.5.0.0: choose() step behavior change**
>
> The `choose( )` step behavior changed in TinkerPop 3.8.0. For migration guidance, see the page-level note at the top of this page.

- choose( )
- coin( )
- fail( )
- inject( )
- match( )
- math( )
- max( )
- mean( )

- min( )
- option( )
- optional( )
- path( )
- propertyMap( )
- sample( )
- skip( )
- tail( )
- timeLimit( )
- tree( )

Gremlin steps that are not supported in Neptune at all

The following Gremlin steps are not supported at all in Neptune. In most cases this is because they require a `GraphComputer`, which Neptune does not currently support.

- connectedComponent( )
- io( )
- shortestPath( )
- withComputer( )
- pageRank( )
- peerPressure( )
- program( )

The `io()` step is actually partially supported, in that it can be used to `read()` from a URL but not to `write()`.

Using Gremlin with the Amazon Neptune DFE query engine

You can enable Neptune's alternative query engine, known as DFE, in two ways:

- Set the `neptune_dfe_query_engine` instance parameter to `enabled` for all queries on the instance.

- Use the useDFE query hint on individual queries (requires `neptune_dfe_query_engine` to be set to its default value of `viaQueryHint`).

When DFE is enabled, Neptune translates read-only Gremlin queries and traversals into an intermediate logical representation and runs them on DFE whenever possible.

Starting with Neptune engine release 1.5.0.0, DFE natively supports most Gremlin steps. When a step isn't supported by DFE, Neptune falls back to TinkerPop to run the step. The `explain` and `profile` reports include warnings when this happens.

Gremlin step coverage in DFE

Gremlin DFE can be used by either enabling the instance parameter or using the `Neptune#useDFE` query hint. For more information please refer to Using Gremlin with the Neptune DFE query engine.

The following sections describe DFE step coverage for Neptune engine release 1.5.0.0 and later, and for Neptune engine releases before 1.5.0.0.

Neptune engine release 1.5.0.0 and later

Starting with Amazon Neptune engine release 1.5.0.0, DFE natively supports most Gremlin steps. The following steps are not natively supported on DFE:

- **Mutation** – `addV()`, `addE()`, `drop()`, `mergeV()`, `mergeE()`, and `property()`.
- **Filters and predicates** – `all()`, `any()`, `none()`, `TextP.regex`, and `TextP.notRegex`.
- **Side-effects** – `barrier()`, `subgraph()`, `timeLimit()`, and `tree()`.
- **Map** – `asBool()`, `asNumber()`, `element()`, `format()`, `math()`, `merge()`, and `propertyMap()`.

If you use one of these steps, Neptune falls back to TinkerPop processing for that part of the query.

See also Gremlin standards compliance for Neptune-wide differences from Apache TinkerPop that also apply under DFE.

A Gremlin query can contain a set of entirely supported steps and still include step patterns that do not translate fully to DFE. If you use a step in a more complex or less common form, that part of

the query falls back to TinkerPop for processing. To check whether a specific query runs natively in DFE, use the explain or profile output.

Neptune engine releases before 1.5.0.0

The following steps are available to use in Gremlin DFE.

Path and traversal steps:

asDate(), barrier(), call(), cap(), dateAdd(), dateDiff(), disjunct(), drop(), fail(), filter(), flatMap(), id(), identity(), index(), intersect(), inject(), label(), length(), loops(), map(), order(), order(local), path(), project(), range(), repeat(), reverse(), sack(), sample(), select(), sideEffect(), split(), unfold(), union()

Aggregate and collection steps:

aggregate(global), combine(), count(), dedup(), dedup(local), fold(), group(), groupCount(),

Mathematical steps:

max(), mean(), min(), sum()

Element steps:

otherV(), elementMap(), element(), v(), out(), in(), both(), outE(), inE(), bothE(), outV(), inV(), bothV(), otherV()

Property steps:

properties(), key(), valueMap(), value()

Filter steps:

and(), coalesce(), coin(), discard(), has(), is(), local(), none(P), not(), or(), where()

String manipulation steps:

concat(), lTrim(), rTrim(), substring(), toLower(), toUpper(), trim()

Predicates:

- Compare: eq, neq, lt, lte, gt, gte

- Contains: within, without
- TextP: endingWith, containing, notStartingWith, notEndingWith, notContaining
- P: and, or, between, outside, inside

Limitations

Repeat with Limit, Labels inside repeat traversal and dedup are not supported in DFE yet.

```
// With Limit inside the repeat traversal
  g.V().has('code','AGR').repeat(out().limit(5)).until(has('code','FRA'))

  // With Labels inside the repeat traversal
  g.V().has('code','AGR').repeat(out().as('a')).until(has('code','FRA'))

  // With Dedup inside the repeat traversal
  g.V().has('code','AGR').repeat(out().dedup()).until(has('code','FRA'))
```

Path with nested repeats, or branching steps are not supported yet.

```
// Path with branching steps
  g.V().has('code','AGR').union(identity, outE().inV()).path().by('code')


  // With nested repeat
  g.V().has('code','AGR').repeat(out().union(identity(), out())).path().by('code')
```

Query planning interleaving

When Neptune encounters a Gremlin step without a corresponding native DFE operator, it runs as much of the traversal on DFE as possible. It then falls back to TinkerPop only for the unsupported step.

Neptune represents any intermediate, non-prefix query translation as a `NeptuneInterleavingStep` in the `explain` and `profile` outputs.

For performance comparison, you can turn off interleaving in a query while still using the DFE engine to run the prefix part. You can also use only the TinkerPop engine for non-prefix query execution. To do this, use the `disableInterleaving` query hint.

Just as the useDFE query hint with a value of `false` prevents a query from being run on the DFE at all, the `disableInterleaving` query hint with a value of `true` turns off DFE interleaving for translation of a query. For example:

```
g.with('Neptune#disableInterleaving', true)
 .V().has('genre','drama').in('likes')
```

Updated Gremlin `explain` and `profile` output

Gremlin explain provides details about the optimized traversal that Neptune uses to run a query. See the sample DFE `explain` output for an example of what `explain` output looks like when DFE is enabled.

The Gremlin `profile` API runs a specified Gremlin traversal, collects various metrics about the run, and produces a profile report that contains details about the optimized query plan and the runtime statistics of various operators. See sample DFE `profile` output for an example of what `profile` output looks like when DFE is enabled.

> **Note**
>
> As of Neptune engine release 1.5.0.0, DFE support for Gremlin is no longer considered experimental. In engine releases before 1.5.0.0, DFE support for Gremlin was experimental and the exact format of the `explain` and `profile` output was subject to change. See Neptune engine releases before 1.5.0.0 for the limitations that applied to earlier releases.

Accessing the Neptune Graph with openCypher

Neptune supports building graph applications using openCypher, currently one of the most popular query languages for developers working with graph databases. Developers, business analysts, and data scientists like openCypher's SQL-inspired syntax because it provides a familiar structure to compose queries for graph applications.

openCypher is a declarative query language for property graphs that was originally developed by Neo4j, then open-sourced in 2015, and contributed to the openCypher project under an Apache 2 open-source license. Its syntax is documented in the Cypher Query Language Reference, Version 9.

For the limitations and differences in Neptune support of the openCypher specification, see openCypher specification compliance in Amazon Neptune.

> **Note**
>
> The current Neo4j implementation of the Cypher query language has diverged in some ways from the openCypher specification. If you are migrating current Neo4j Cypher code to Neptune, see Neptune compatibility with Neo4j and Rewriting Cypher queries to run in openCypher on Neptune for help.

Starting with engine release 1.1.1.0, openCypher is available for production use in Neptune.

Gremlin vs. openCypher: similarities and differences

Gremlin and openCypher are both property-graph query languages, and they are complementary in many ways.

Gremlin was designed to appeal to programmers and fit seamlessly into code. As a result, Gremlin is imperative by design, whereas openCypher's declarative syntax may feel more familiar for people with SQL or SPARQL experience. Gremlin might seem more natural to a data scientist using Python in a Jupyter notebook, whereas openCypher might seem more intuitive to a business user with some SQL background.

The nice thing is that **you don't have to choose** between Gremlin and openCypher in Neptune. Queries in either language can operate on the same graph regardless of which of the two language was used to enter that data. You may find it more convenient to use Gremlin for some things and openCypher for others, depending on what you're doing.

Gremlin uses an imperative syntax that lets you control how you move through your graph in a series of steps, each of which takes in a stream of data, performs some action on it (using a filter, map, and so forth), and then outputs the results to the next step. A Gremlin query commonly takes the form, `g.V()`, followed by additional steps.

In openCypher, you use a declarative syntax, inspired by SQL, that specifies a pattern of nodes and relationships to find in your graph using a motif syntax (like `()-[]->()`). An openCypher query often starts with a `MATCH` clause, followed by other clauses such as `WHERE`, `WITH`, and `RETURN`.

Getting started using openCypher

You can query property-graph data in Neptune using openCypher regardless of how it was loaded, but you can't use openCypher to query data loaded as RDF.

The Neptune bulk loader accepts property-graph data in a CSV format for Gremlin, and in a CSV format for openCypher. Also, of course, you can add property data to your graph using Gremlin and/or openCypher queries.

Data processed by openCypher takes the form of an unordered series of key/value maps. The main way to refine, manipulate, and augment these maps is to use clauses that perform tasks such as pattern matching, insertion, update, and deletion on the key/value pairs.

There are several clauses in openCypher for finding data patterns in the graph, of which `MATCH` is the most common. `MATCH` lets you specify the pattern of nodes, relationships, and filters that you want to look for in your graph. For example:

- **Get all nodes**

```
MATCH (n) RETURN n
```

- **Find connected nodes**

```
MATCH (n)-[r]->(d) RETURN n, r, d
```

- **Find a path**

```
MATCH p=(n)-[r]->(d) RETURN p
```

- **Get all nodes with a label**

```
MATCH (n:airport) RETURN n
```

Note that the first query above returns every single node in your graph, and the next two return every node that has a relationship— this is not generally recommended! In almost all cases, you want to narrow down the data being returned, which you can do by specifying node or relationship labels and properties, as in the fourth example.

You can find a handy cheat-sheet for openCypher syntax in the Neptune github sample repository.

Neptune openCypher status servlet and status endpoint

The openCypher status endpoint provides access to information about queries that are currently running on the server or waiting to run. It also lets you cancel those queries. The endpoint is:

```
https://(the server):(the port number)/openCypher/status
```

You can use the HTTP GET and POST methods to get current status from the server, or to cancel a query. You can also use the DELETE method to cancel a running or waiting query.

Parameters for status requests

Status query parameters

- **includeWaiting** (true or false) – When set to true and other parameters are not present, causes status information for waiting queries to be returned as well as for running queries.
- **cancelQuery** – Used only with GET and POST methods, to indicate that this is a cancelation request. The DELETE method does not need this parameter.

  The value of the cancelQuery parameter is not used, but when cancelQuery is present, the queryId parameter is required, to identify which query to cancel.
- **queryId** – Contains the ID of a specific query.

  When used with the GET or POST method and the cancelQuery parameter is not present, queryId causes status information to be returned for the specific query it identifies. If the cancelQuery parameter is present, then the specific query that queryId identifies is canceled.

  When used with the DELETE method, queryId always indicates a specific query to be canceled.
- **silent** – Only used when canceling a query. If set to true, causes the cancelation to happen silently.

Status request response fields

Status response fields if the ID of a specific query is not provided

- **acceptedQueryCount** – The number of queries that have been accepted but not yet completed, including queries in the queue.
- **runningQueryCount** – The number of currently running openCypher queries.
- **queries** – A list of the current openCypher queries.

Status response fields for a specific query

- **queryId** – A GUID id for the query. Neptune automatically assigns this ID value to each query, or you can also assign your own ID (see Inject a Custom ID Into a Neptune Gremlin or SPARQL Query).
- **queryString** – The submitted query. This is truncated to 1024 characters if it is longer than that.
- **queryEvalStats** – Statistics for this query:
  - **waited** – Indicates how long the query waited, in milliseconds.
  - **elapsed** – The number of milliseconds the query has been running so far.
  - **cancelled** – `True` indicates that the query was cancelled, or `False` that it has not been cancelled.

Examples of status requests and responses

- **Request for the status of all queries, including those waiting:**

  AWS CLI

  ```
  aws neptunedata get-open-cypher-query-status \
    --endpoint-url https://your-neptune-endpoint:port \
    --include-waiting
  ```

  For more information, see get-open-cypher-query-status in the AWS CLI Command Reference.

  SDK

  ```
  import boto3
  from botocore.config import Config

  client = boto3.client(
      'neptunedata',
      endpoint_url='https://your-neptune-endpoint:port',
      config=Config(read_timeout=None, retries={'total_max_attempts': 1})
  )

  response = client.get_open_cypher_query_status(
      includeWaiting=True
  )
  ```

```
print(response)
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher/status \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "includeWaiting=true"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher/status \
  --data-urlencode "includeWaiting=true"
```

Response:

```
{
  "acceptedQueryCount" : 0,
  "runningQueryCount" : 0,
  "queries" : [ ]
}
```

- **Request for the status of running queries, not including those waiting:**:

  AWS CLI

  ```
  aws neptunedata get-open-cypher-query-status \
    --endpoint-url https://your-neptune-endpoint:port
  ```

  For more information, see get-open-cypher-query-status in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_open_cypher_query_status()

print(response)
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher/status \
  --region us-east-1 \
  --service neptune-db
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher/status
```

Response:

```
{
  "acceptedQueryCount" : 0,
  "runningQueryCount" : 0,
  "queries" : [ ]
}
```

- **Request for the status of a single query:**

  AWS CLI

  ```
  aws neptunedata get-open-cypher-query-status \
    --endpoint-url https://your-neptune-endpoint:port \
    --query-id eadc6eea-698b-4a2f-8554-5270ab17ebee
  ```

  For more information, see get-open-cypher-query-status in the AWS CLI Command Reference.

  SDK

  ```
  import boto3
  from botocore.config import Config

  client = boto3.client(
      'neptunedata',
      endpoint_url='https://your-neptune-endpoint:port',
      config=Config(read_timeout=None, retries={'total_max_attempts': 1})
  )

  response = client.get_open_cypher_query_status(
      queryId='eadc6eea-698b-4a2f-8554-5270ab17ebee'
  )

  print(response)
  ```

  For AWS SDK examples in other languages, see AWS SDK.

  awscurl

  ```
  awscurl https://your-neptune-endpoint:port/openCypher/status \
    --region us-east-1 \
    --service neptune-db \
    -X POST \
    -d "queryId=eadc6eea-698b-4a2f-8554-5270ab17ebee"
  ```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher/status \
  --data-urlencode "queryId=eadc6eea-698b-4a2f-8554-5270ab17ebee"
```

Response:

```
{
  "queryId" : "eadc6eea-698b-4a2f-8554-5270ab17ebee",
  "queryString" : "MATCH (n1)-[:knows]->(n2), (n2)-[:knows]->(n3), (n3)-[:knows]-
>(n4), (n4)-[:knows]->(n5), (n5)-[:knows]->(n6), (n6)-[:knows]->(n7), (n7)-[:knows]-
>(n8), (n8)-[:knows]->(n9), (n9)-[:knows]->(n10) RETURN COUNT(n1);",
  "queryEvalStats" : {
    "waited" : 0,
    "elapsed" : 23463,
    "cancelled" : false
  }
}
```

- **Requests to cancel a query**

  AWS CLI

  ```
  aws neptunedata cancel-open-cypher-query \
    --endpoint-url https://your-neptune-endpoint:port \
    --query-id f43ce17b-db01-4d37-a074-c76d1c26d7a9
  ```

  For more information, see cancel-open-cypher-query in the AWS CLI Command Reference.

  SDK

  ```
  import boto3
  from botocore.config import Config

  client = boto3.client(
  ```

```
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.cancel_open_cypher_query(
    queryId='f43ce17b-db01-4d37-a074-c76d1c26d7a9'
)

print(response)
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher/status \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "cancelQuery" \
  -d "queryId=f43ce17b-db01-4d37-a074-c76d1c26d7a9"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

1. Using POST:

```
curl -X POST https://your-neptune-endpoint:port/openCypher/status \
  --data-urlencode "cancelQuery" \
  --data-urlencode "queryId=f43ce17b-db01-4d37-a074-c76d1c26d7a9"
```

2. Using GET:

```
curl -X GET https://your-neptune-endpoint:port/openCypher/status \
  --data-urlencode "cancelQuery" \
```

```
  --data-urlencode "queryId=588af350-cfde-4222-bee6-b9cedc87180d"
```

3. Using DELETE:

```
curl -X DELETE \
  "https://your-neptune-endpoint:port/openCypher/status?queryId=b9a516d1-
d25c-4301-bb80-10b2743ecf0e"
```

Response:

```
{
  "status" : "200 OK",
  "payload" : true
}
```

The Amazon Neptune OpenCypher HTTPS endpoint

Topics

- OpenCypher read and write queries on the HTTPS endpoint
- The default OpenCypher JSON results format
- Optional HTTP trailing headers for multi-part OpenCypher responses

> **Note**
>
> Neptune does not currently support HTTP/2 for REST API requests. Clients must use HTTP/1.1 when connecting to endpoints.

OpenCypher read and write queries on the HTTPS endpoint

The OpenCypher HTTPS endpoint supports read and update queries using both the GET and the POST method. The DELETE and PUT methods are not supported.

The following instructions walk you through connecting to the OpenCypher endpoint using the `curl` command and HTTPS. You must follow these instructions from an Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB instance.

The syntax is:

```
HTTPS://(the server):(the port number)/openCypher
```

Here is a sample read query:

AWS CLI

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "MATCH (n1) RETURN n1"
```

For more information, see execute-open-cypher-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_query(
    openCypherQuery='MATCH (n1) RETURN n1'
)

print(response['results'])
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=MATCH (n1) RETURN n1"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=MATCH (n1) RETURN n1"
```

Here is a sample write/update query:

AWS CLI

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "CREATE (n:Person { age: 25 })"
```

For more information, see execute-open-cypher-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_query(
    openCypherQuery='CREATE (n:Person { age: 25 })'
)

print(response['results'])
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=CREATE (n:Person { age: 25 })"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=CREATE (n:Person { age: 25 })"
```

The default OpenCypher JSON results format

The following JSON format is returned by default, or by setting the request header explicitly to `Accept: application/json`. This format is designed to be easily parsed into objects using native-language features of most libraries.

The JSON document that is returned contains one field, `results`, which contains the query return values. The examples below show the JSON formatting for common values.

Value response example:

```
{
  "results": [
    {
      "count(a)": 121
    }
  ]
}
```

Node response example:

```
{
  "results": [
    {
      "a": {
        "~id": "22",
        "~entityType": "node",
        "~labels": [
          "airport"
        ],
        "~properties": {
          "desc": "Seattle-Tacoma",
          "lon": -122.30899810791,
          "runways": 3,
          "type": "airport",
          "country": "US",
          "region": "US-WA",
          "lat": 47.4490013122559,
          "elev": 432,
          "city": "Seattle",
          "icao": "KSEA",
          "code": "SEA",
          "longest": 11901
        }
      }
    }
  ]
}
```

Relationship response example:

```
{
  "results": [
    {
      "r": {
        "~id": "7389",
        "~entityType": "relationship",
        "~start": "22",
        "~end": "151",
        "~type": "route",
        "~properties": {
          "dist": 956
        }
      }
```

```
    }
  ]
}
```

Path response example:

```
{
  "results": [
    {
      "p": [
        {
          "~id": "22",
          "~entityType": "node",
          "~labels": [
            "airport"
          ],
          "~properties": {
            "desc": "Seattle-Tacoma",
            "lon": -122.30899810791,
            "runways": 3,
            "type": "airport",
            "country": "US",
            "region": "US-WA",
            "lat": 47.4490013122559,
            "elev": 432,
            "city": "Seattle",
            "icao": "KSEA",
            "code": "SEA",
            "longest": 11901
          }
        },
        {
          "~id": "7389",
          "~entityType": "relationship",
          "~start": "22",
          "~end": "151",
          "~type": "route",
          "~properties": {
            "dist": 956
          }
        },
        {
          "~id": "151",
```

```
          "~entityType": "node",
          "~labels": [
            "airport"
          ],
          "~properties": {
            "desc": "Ontario International Airport",
            "lon": -117.600997924805,
            "runways": 2,
            "type": "airport",
            "country": "US",
            "region": "US-CA",
            "lat": 34.0559997558594,
            "elev": 944,
            "city": "Ontario",
            "icao": "KONT",
            "code": "ONT",
            "longest": 12198
          }
        }
      ]
    }
  ]
}
```

Optional HTTP trailing headers for multi-part OpenCypher responses

This feature is available starting with Neptune engine release 1.4.5.0.

The HTTP response to OpenCypher queries and updates is typically returned in multiple chunks. When failures occur after the initial response chunks have been sent (with an HTTP status code of 200), it can be challenging to diagnose the issue. By default, `Neptune reports such failures by appending an error message to the message body, which may be corrupted due to the streaming nature of the response.

Using trailing headers

To improve error detection and diagnosis, you can enable trailing headers by including a transfer-encoding (TE) trailers header (te: trailers)in your request. Doing this will cause Neptune to include two new header fields within the trailing headers of the response chunks:

- `X-Neptune-Status` – contains the response code followed by a short name. For instance, in case of success the trailing header would be: `X-Neptune-Status: 200 OK`. In the case of

failure, the response code would be a Neptune engine error code such as `X-Neptune-Status: 500 TimeLimitExceededException`.

- `X-Neptune-Detail` – is empty for successful requests. In the case of errors, it contains the JSON error message. Because only ASCII characters are allowed in HTTP header values, the JSON string is URL encoded. The error message is also still appended to the response message body.

For more information, see the MDN page about TE request headers.

OpenCypher trailing headers usage example

This example demonstrates how trailing headers help diagnose a query that exceeds its time limit:

```
curl --raw 'https://your-neptune-endpoint:port/openCypher' \
-H 'TE: trailers' \
-d 'query=MATCH(n) RETURN n.firstName'


Output:
< HTTP/1.1 200 OK
< transfer-encoding: chunked
< trailer: X-Neptune-Status, X-Neptune-Detail
< content-type: application/json;charset=UTF-8
<
<
{
  "results": [{
      "n.firstName": "Hossein"
    }, {
      "n.firstName": "Jan"
    }, {
      "n.firstName": "Miguel"
    }, {
      "n.firstName": "Eric"
    },
{"detailedMessage":"Operation terminated (deadline exceeded)",
"code":"TimeLimitExceededException",
"requestId":"a7e9d2aa-fbb7-486e-8447-2ef2a8544080",
"message":"Operation terminated (deadline exceeded)"}
0
X-Neptune-Status: 500 TimeLimitExceededException
X-Neptune-Detail: %7B%22detailedMessage%22%3A%22Operation+terminated+%28deadline
+exceeded%29%22%2C%22code%22%3A%22TimeLimitExceededException%22%2C%22requestId%22%3A
```

```
%22a7e9d2aa-fbb7-486e-8447-2ef2a8544080%22%2C%22message%22%3A%22Operation+terminated+
%28deadline+exceeded%29%22%7D
```

Response breakdown:

The previous example shows how an OpenCypher response with trailing headers can help diagnose query failures. Here we see four sequential parts: (1) initial headers with a 200 OK status indicating streaming begins, (2) partial (broken) JSON results successfully streamed before the failure, (3) the appended error message showing the timeout, and (4) trailing headers containing the final status (500 TimeLimitExceededException) and detailed error information.

Using the AWS SDK to run openCypher queries

With the AWS SDK, you can run openCypher queries against your Neptune graph using a programming language of your choice. The Neptune data API SDK (service name `neptunedata`) provides the ExecuteOpenCypherQuery action for submitting openCypher queries.

You must run these examples from an Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB cluster, or from a location that has network connectivity to your cluster endpoint.

Direct links to the API reference documentation for the `neptunedata` service in each SDK language can be found below:

Programming language	neptunedata API reference
C++	https://sdk.amazonaws.com/cpp/api/LATEST/aws-cpp-sdk-neptunedata/html/annotated.html
Go	https://docs.aws.amazon.com/sdk-for-go/api/service/neptunedata/
Java	https://sdk.amazonaws.com/java/api/latest/software/amazon/awssdk/services/neptunedata/package-summary.html
JavaScript	https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/Package/-aws-sdk-client-neptunedata/

Programming language	neptunedata API reference
Kotlin	https://sdk.amazonaws.com/kotlin/api/latest/neptunedata/index.html
.NET	https://docs.aws.amazon.com/sdkfornet/v3/apidocs/items/Neptunedata/NNeptunedata.html
PHP	https://docs.aws.amazon.com/aws-sdk-php/v3/api/namespace-Aws.Neptunedata.html
Python	https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/neptunedata.html
Ruby	https://docs.aws.amazon.com/sdk-for-ruby/v3/api/Aws/Neptunedata.html
Rust	https://crates.io/crates/aws-sdk-neptunedata
CLI	https://docs.aws.amazon.com/cli/latest/reference/neptunedata/

openCypher AWS SDK examples

The following examples show how to set up a `neptunedata` client, run an openCypher query, and print the results. Replace *`YOUR_NEPTUNE_HOST`* and *`YOUR_NEPTUNE_PORT`* with the endpoint and port of your Neptune DB cluster.

> **Client-side timeout and retry configuration**
>
> The SDK client timeout controls how long the *client* waits for a response. It does not control how long the query runs on the server. If the client times out before the server finishes, the query may continue running on Neptune while the client has no way to retrieve the results. We recommend setting the client-side read timeout to `0` (no timeout) or to a value that is at least a few seconds longer than the server-side neptune_query_timeout setting on your Neptune DB cluster. This lets Neptune control when queries time out.

We also recommend setting the maximum retry attempts to 1 (no retries). If the SDK retries a query that is still running on the server, it can result in duplicate operations. This is especially important for mutation queries, where a retry could cause unintended duplicate writes.

Python

1. Follow the installation instructions to install Boto3.
2. Create a file named `openCypherExample.py` and paste the following code:

```
import boto3
import json
from botocore.config import Config

# Disable the client-side read timeout and retries so that
# Neptune's server-side neptune_query_timeout controls query duration.
client = boto3.client(
    'neptunedata',
    endpoint_url=f'https://YOUR_NEPTUNE_HOST:YOUR_NEPTUNE_PORT',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_query(
    openCypherQuery='MATCH (n) RETURN n LIMIT 1'
)

print(json.dumps(response['results'], indent=2))
```

3. Run the example: `python openCypherExample.py`

Java

1. Follow the installation instructions to set up the AWS SDK for Java.
2. Use the following code to set up a `NeptunedataClient`, run an openCypher query, and print the result:

```
import java.net.URI;
import java.time.Duration;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
```

```
import software.amazon.awssdk.core.retry.RetryPolicy;
import software.amazon.awssdk.services.neptunedata.NeptunedataClient;
import
 software.amazon.awssdk.services.neptunedata.model.ExecuteOpenCypherQueryRequest;
import
 software.amazon.awssdk.services.neptunedata.model.ExecuteOpenCypherQueryResponse;

// Disable the client-side timeout and retries so that
// Neptune's server-side neptune_query_timeout controls query duration.
NeptunedataClient client = NeptunedataClient.builder()
    .endpointOverride(URI.create("https://YOUR_NEPTUNE_HOST:YOUR_NEPTUNE_PORT"))
    .overrideConfiguration(ClientOverrideConfiguration.builder()
        .apiCallTimeout(Duration.ZERO)
        .retryPolicy(RetryPolicy.none())
        .build())
    .build();

ExecuteOpenCypherQueryRequest request = ExecuteOpenCypherQueryRequest.builder()
    .openCypherQuery("MATCH (n) RETURN n LIMIT 1")
    .build();

ExecuteOpenCypherQueryResponse response = client.executeOpenCypherQuery(request);

System.out.println(response.results().toString());
```

JavaScript

1. Follow the installation instructions to set up the AWS SDK for JavaScript. Install the neptunedata client package: `npm install @aws-sdk/client-neptunedata`.
2. Create a file named `openCypherExample.js` and paste the following code:

```
import { NeptunedataClient, ExecuteOpenCypherQueryCommand } from "@aws-sdk/client-
neptunedata";
import { NodeHttpHandler } from "@smithy/node-http-handler";

const config = {
    endpoint: "https://YOUR_NEPTUNE_HOST:YOUR_NEPTUNE_PORT",
    // Disable the client-side request timeout so that
    // Neptune's server-side neptune_query_timeout controls query duration.
    requestHandler: new NodeHttpHandler({
        requestTimeout: 0
    }),
```

```
    maxAttempts: 1
};

const client = new NeptunedataClient(config);

const input = {
    openCypherQuery: "MATCH (n) RETURN n LIMIT 1"
};

const command = new ExecuteOpenCypherQueryCommand(input);
const response = await client.send(command);

console.log(JSON.stringify(response, null, 2));
```

3. Run the example: `node openCypherExample.js`

Using the Bolt protocol to make openCypher queries to Neptune

Bolt is a statement-oriented client/server protocol initially developed by Neo4j and licensed under the Creative Commons 3.0 Attribution-ShareAlike license. It is client-driven, meaning that the client always initiates message exchanges.

To connect to Neptune using Neo4j's Bolt drivers, simply replace the URL and Port number with your cluster endpoints using the `bolt` URI scheme. If you have a single Neptune instance running, use the read_write endpoint. If multiple instances are running, then two drivers are recommended, one for the writer and another for all the read replicas. If you have only the default two endpoints, a read_write and a read_only driver are sufficient, but if you have custom endpoints as well, consider creating a driver instance for each one.

> **Note**
>
> Althought the Bolt spec states that Bolt can connect using either TCP or WebSockets, Neptune only supports TCP connections for Bolt.

Neptune allows up to 1000 concurrent Bolt connections on all instance sizes except for t3.medium and t4g.medium. On t3.medium and t4g.medium instances only 512 connections are allowed.

For examples of openCypher queries in various languages that use the Bolt drivers, see the Neo4j Drivers & Language Guides documentation.

> **Important**
>
> The Neo4j Bolt drivers for Python, .NET, JavaScript, and Golang did not initially support the automatic renewal of AWS Signature v4 authentication tokens. This means that after the signature expired (often in 5 minutes), the driver failed to authenticate, and subsequent requests failed. The Python, .NET, JavaScript, and Go examples below were all affected by this issue.
> See Neo4j Python driver issue #834, Neo4j .NET issue #664, Neo4j JavaScript driver issue #993, and Neo4j goLang driver issue #429 for more information.
> As of driver version 5.8.0, a new preview re-authentication API was released for the Go driver (see v5.8.0 - Feedback wanted on re-authentication).

Using Bolt with Java to connect to Neptune

You can download a driver for whatever version you want to use from the Maven MVN repository, or can add this dependency to your project:

```
<dependency>
  <groupId>org.neo4j.driver</groupId>
  <artifactId>neo4j-java-driver</artifactId>
  <version>4.3.3</version>
</dependency>
```

Then, to connect to Neptune in Java using one of these Bolt drivers, create a driver instance for the primary/writer instance in your cluster using code like the following:

```
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;

final Driver driver =
  GraphDatabase.driver("bolt://(your cluster endpoint URL):(your cluster port)",
    AuthTokens.none(),
    Config.builder().withEncryption()
                    .withTrustStrategy(TrustStrategy.trustSystemCertificates())
                    .build());
```

If you have one or more reader replicas, you can similarly create a driver instance for them using code like this:

```
final Driver read_only_driver =              // (without connection timeout)
  GraphDatabase.driver("bolt://(your cluster endpoint URL):(your cluster port)",
      Config.builder().withEncryption()
                      .withTrustStrategy(TrustStrategy.trustSystemCertificates())
                      .build());
```

Or, with a timeout:

```
final Driver read_only_timeout_driver =      // (with connection timeout)
  GraphDatabase.driver("bolt://(your cluster endpoint URL):(your cluster port)",
    Config.builder().withConnectionTimeout(30, TimeUnit.SECONDS)
                    .withEncryption()
                    .withTrustStrategy(TrustStrategy.trustSystemCertificates())
                    .build());
```

If you have custom endpoints, it may also be worthwhile to create a driver instance for each one.

A Python openCypher query example using Bolt

Here is how to make an openCypher query in Python using Bolt:

```
python -m pip install neo4j
```

```
from neo4j import GraphDatabase
uri = "bolt://(your cluster endpoint URL):(your cluster port)"
driver = GraphDatabase.driver(uri, auth=("username", "password"), encrypted=True)
```

Note that the `auth` parameters are ignored.

A .NET openCypher query example using Bolt

To make an openCypher query in .NET using Bolt, the first step is to install the Neo4j driver using NuHet. To make synchronous calls, use the `.Simple` version, like this:

```
Install-Package Neo4j.Driver.Simple-4.3.0
```

```
using Neo4j.Driver;

namespace hello
{
  // This example creates a node and reads a node in a Neptune
```

```
 // Cluster where IAM Authentication is not enabled.
 public class HelloWorldExample : IDisposable
 {
   private bool _disposed = false;
   private readonly IDriver _driver;
   private static string url = "bolt://(your cluster endpoint URL):(your cluster
port)";
   private static string createNodeQuery = "CREATE (a:Greeting) SET a.message =
'HelloWorldExample'";
   private static string readNodeQuery = "MATCH(n:Greeting) RETURN n.message";

   ~HelloWorldExample() => Dispose(false);

   public HelloWorldExample(string uri)
   {
     _driver = GraphDatabase.Driver(uri, AuthTokens.None, o =>
o.WithEncryptionLevel(EncryptionLevel.Encrypted));
   }

   public void createNode()
   {
     // Open a session
     using (var session = _driver.Session())
     {
        // Run the query in a write transaction
       var greeting = session.WriteTransaction(tx =>
       {
         var result = tx.Run(createNodeQuery);
         // Consume the result
         return result.Consume();
       });

       // The output will look like this:
       //   ResultSummary{Query=`CREATE (a:Greeting) SET a.message =
'HelloWorldExample".....
       Console.WriteLine(greeting);
     }
   }

   public void retrieveNode()
   {
     // Open a session
     using (var session = _driver.Session())
     {
```

```
        // Run the query in a read transaction
        var greeting = session.ReadTransaction(tx =>
        {
          var result = tx.Run(readNodeQuery);
          // Consume the result. Read the single node
          // created in a previous step.
          return result.Single()[0].As<string>();
        });
        // The output will look like this:
        //   HelloWorldExample
        Console.WriteLine(greeting);
      }
    }

    public void Dispose()
    {
      Dispose(true);
      GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
      if (_disposed)
        return;
      if (disposing)
      {
        _driver?.Dispose();
      }
      _disposed = true;
    }

    public static void Main()
    {
      using (var apiCaller = new HelloWorldExample(url))
      {
        apiCaller.createNode();
        apiCaller.retrieveNode();
      }
    }
  }
}
```

A Java openCypher query example using Bolt with IAM authentication

The Java code below shows how to make openCypher queries in Java using Bolt with IAM authentication. The JavaDoc comment describes its usage. Once a driver instance is available, you can use it to make multiple authenticated requests.

```
package software.amazon.neptune.bolt;

import com.amazonaws.DefaultRequest;
import com.amazonaws.Request;
import com.amazonaws.auth.AWS4Signer;
import com.amazonaws.auth.AWSCredentialsProvider;
import com.amazonaws.http.HttpMethodName;
import com.google.gson.Gson;
import lombok.Builder;
import lombok.Getter;
import lombok.NonNull;
import org.neo4j.driver.Value;
import org.neo4j.driver.Values;
import org.neo4j.driver.internal.security.InternalAuthToken;
import org.neo4j.driver.internal.value.StringValue;

import java.net.URI;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

import static com.amazonaws.auth.internal.SignerConstants.AUTHORIZATION;
import static com.amazonaws.auth.internal.SignerConstants.HOST;
import static com.amazonaws.auth.internal.SignerConstants.X_AMZ_DATE;
import static com.amazonaws.auth.internal.SignerConstants.X_AMZ_SECURITY_TOKEN;

/**
 * Use this class instead of `AuthTokens.basic` when working with an IAM
 * auth-enabled server. It works the same as `AuthTokens.basic` when using
 * static credentials, and avoids making requests with an expired signature
 * when using temporary credentials. Internally, it generates a new signature
 * on every invocation (this may change in a future implementation).
 *
 * Note that authentication happens only the first time for a pooled connection.
 *
 * Typical usage:
 *
```

```
 * NeptuneAuthToken authToken = NeptuneAuthToken.builder()
 *     .credentialsProvider(credentialsProvider)
 *     .region("aws region")
 *     .url("cluster endpoint url")
 *     .build();
 *
 * Driver driver = GraphDatabase.driver(
 *     authToken.getUrl(),
 *     authToken,
 *     config
 * );
 */

public class NeptuneAuthToken extends InternalAuthToken {
  private static final String SCHEME = "basic";
  private static final String REALM = "realm";
  private static final String SERVICE_NAME = "neptune-db";
  private static final String HTTP_METHOD_HDR = "HttpMethod";
  private static final String DUMMY_USERNAME = "username";
  @NonNull
  private final String region;
  @NonNull
  @Getter
  private final String url;
  @NonNull
  private final AWSCredentialsProvider credentialsProvider;
  private final Gson gson = new Gson();

  @Builder
  private NeptuneAuthToken(
      @NonNull final String region,
      @NonNull final String url,
      @NonNull final AWSCredentialsProvider credentialsProvider
  ) {
      // The superclass caches the result of toMap(), which we don't want
      super(Collections.emptyMap());
      this.region = region;
      this.url = url;
      this.credentialsProvider = credentialsProvider;
  }

  @Override
  public Map<String, Value> toMap() {
    final Map<String, Value> map = new HashMap<>();
```

```
    map.put(SCHEME_KEY, Values.value(SCHEME));
    map.put(PRINCIPAL_KEY, Values.value(DUMMY_USERNAME));
    map.put(CREDENTIALS_KEY, new StringValue(getSignedHeader()));
    map.put(REALM_KEY, Values.value(REALM));

    return map;
  }

  private String getSignedHeader() {
    final Request<Void> request = new DefaultRequest<>(SERVICE_NAME);
    request.setHttpMethod(HttpMethodName.GET);
    request.setEndpoint(URI.create(url));
    // Comment out the following line if you're using an engine version older than
 1.2.0.0
    request.setResourcePath("/opencypher");

    final AWS4Signer signer = new AWS4Signer();
    signer.setRegionName(region);
    signer.setServiceName(request.getServiceName());
    signer.sign(request, credentialsProvider.getCredentials());

    return getAuthInfoJson(request);
  }

  private String getAuthInfoJson(final Request<Void> request) {
    final Map<String, Object> obj = new HashMap<>();
    obj.put(AUTHORIZATION, request.getHeaders().get(AUTHORIZATION));
    obj.put(HTTP_METHOD_HDR, request.getHttpMethod());
    obj.put(X_AMZ_DATE, request.getHeaders().get(X_AMZ_DATE));
    obj.put(HOST, request.getHeaders().get(HOST));
    obj.put(X_AMZ_SECURITY_TOKEN, request.getHeaders().get(X_AMZ_SECURITY_TOKEN));

    return gson.toJson(obj);
  }
}
```

A Python openCypher query example using Bolt with IAM authentication

The Python class below lets you make openCypher queries in Python using Bolt with IAM authentication:

```
import json
```

```
from neo4j import Auth
from botocore.awsrequest import AWSRequest
from botocore.credentials import Credentials
from botocore.auth import (
  SigV4Auth,
  _host_from_url,
)

SCHEME = "basic"
REALM = "realm"
SERVICE_NAME = "neptune-db"
DUMMY_USERNAME = "username"
HTTP_METHOD_HDR = "HttpMethod"
HTTP_METHOD = "GET"
AUTHORIZATION = "Authorization"
X_AMZ_DATE = "X-Amz-Date"
X_AMZ_SECURITY_TOKEN = "X-Amz-Security-Token"
HOST = "Host"


class NeptuneAuthToken(Auth):
  def __init__(
    self,
    credentials: Credentials,
    region: str,
    url: str,
    **parameters
  ):
    # Do NOT add "/opencypher" in the line below if you're using an engine version
 older than 1.2.0.0
    request = AWSRequest(method=HTTP_METHOD, url=url + "/opencypher")
    request.headers.add_header("Host", _host_from_url(request.url))
    sigv4 = SigV4Auth(credentials, SERVICE_NAME, region)
    sigv4.add_auth(request)

    auth_obj = {
      hdr: request.headers[hdr]
      for hdr in [AUTHORIZATION, X_AMZ_DATE, X_AMZ_SECURITY_TOKEN, HOST]
    }
    auth_obj[HTTP_METHOD_HDR] = request.method
    creds: str = json.dumps(auth_obj)
    super().__init__(SCHEME, DUMMY_USERNAME, creds, REALM, **parameters)
```

You use this class to create a driver as follows:

```
  authToken = NeptuneAuthToken(creds, REGION, URL)
  driver = GraphDatabase.driver(URL, auth=authToken, encrypted=True)
```

A Node.js example using IAM authentication and Bolt

The Node.js code below uses the AWS SDK for JavaScript version 3 and ES6 syntax to create a driver that authenticates requests:

```
import neo4j from "neo4j-driver";
import { HttpRequest }  from "@smithy/protocol-http";
import { defaultProvider } from "@aws-sdk/credential-provider-node";
import { SignatureV4 } from "@smithy/signature-v4";
import crypto from "@aws-crypto/sha256-js";
const { Sha256 } = crypto;
import assert from "node:assert";

const region = "us-west-2";
const serviceName = "neptune-db";
const host = "(your cluster endpoint URL)";
const port = 8182;
const protocol = "bolt";
const hostPort = host + ":" + port;
const url = protocol + "://" + hostPort;
const createQuery = "CREATE (n:Greeting {message: 'Hello'}) RETURN ID(n)";
const readQuery = "MATCH(n:Greeting) WHERE ID(n) = $id RETURN n.message";

async function signedHeader() {
  const req = new HttpRequest({
    method: "GET",
    protocol: protocol,
    hostname: host,
    port: port,
    // Comment out the following line if you're using an engine version older than
 1.2.0.0
    path: "/opencypher",
    headers: {
      host: hostPort
    }
  });

  const signer = new SignatureV4({
```

```
    credentials: defaultProvider(),
    region: region,
    service: serviceName,
    sha256: Sha256
  });

  return signer.sign(req, { unsignableHeaders: new Set(["x-amz-content-sha256"]) })
    .then((signedRequest) => {
      const authInfo = {
        "Authorization": signedRequest.headers["authorization"],
        "HttpMethod": signedRequest.method,
        "X-Amz-Date": signedRequest.headers["x-amz-date"],
        "Host": signedRequest.headers["host"],
        "X-Amz-Security-Token": signedRequest.headers["x-amz-security-token"]
      };
      return JSON.stringify(authInfo);
    });
}

async function createDriver() {
  let authToken = { scheme: "basic", realm: "realm", principal: "username",
 credentials: await signedHeader() };

  return neo4j.driver(url, authToken, {
      encrypted: "ENCRYPTION_ON",
      trust: "TRUST_SYSTEM_CA_SIGNED_CERTIFICATES",
      maxConnectionPoolSize: 1,
      // logging: neo4j.logging.console("debug")
    }
  );
}

async function unmanagedTxn(driver) {
  const session = driver.session();
  const tx = session.beginTransaction();
  try {
    const created = await tx.run(createQuery);
    const matched = await tx.run(readQuery, { id: created.records[0].get(0) });
    const msg = matched.records[0].get("n.message");
    assert.equal(msg, "Hello");
    await tx.commit();
  } catch (err) {
    // The transaction will be rolled back, now handle the error.
    console.log(err);
```

```
  } finally {
    await session.close();
  }
}

const driver = await createDriver();
try {
  await unmanagedTxn(driver);
} catch (err) {
  console.log(err);
} finally {
  await driver.close();
}
```

A .NET openCypher query example using Bolt with IAM authentication

To enable IAM authentication in .NET, you need to sign a request when establishing the connection. The example below shows how to create a `NeptuneAuthToken` helper to generate an authentication token:

```
using Amazon.Runtime;
using Amazon.Util;
using Neo4j.Driver;
using System.Security.Cryptography;
using System.Text;
using System.Text.Json;
using System.Web;

namespace Hello
{
  /*
   * Use this class instead of `AuthTokens.None` when working with an IAM-auth-enabled
 server.
   *
   * Note that authentication happens only the first time for a pooled connection.
   *
   * Typical usage:
   *
   * var authToken = new NeptuneAuthToken(AccessKey, SecretKey,
 Region).GetAuthToken(Host);
   * _driver = GraphDatabase.Driver(Url, authToken, o =>
 o.WithEncryptionLevel(EncryptionLevel.Encrypted));
   */
```

```
 public class NeptuneAuthToken
 {
   private const string ServiceName = "neptune-db";
   private const string Scheme = "basic";
   private const string Realm = "realm";
   private const string DummyUserName = "username";
   private const string Algorithm = "AWS4-HMAC-SHA256";
   private const string AWSRequest = "aws4_request";

   private readonly string _accessKey;
   private readonly string _secretKey;
   private readonly string _region;

   private readonly string _emptyPayloadHash;

   private readonly SHA256 _sha256;


   public NeptuneAuthToken(string awsKey = null, string secretKey = null, string
region = null)
   {
     var awsCredentials = awsKey == null || secretKey == null
       ? FallbackCredentialsFactory.GetCredentials().GetCredentials()
       : null;

     _accessKey = awsKey ?? awsCredentials.AccessKey;
     _secretKey = secretKey ?? awsCredentials.SecretKey;
     _region = region ?? FallbackRegionFactory.GetRegionEndpoint().SystemName; //ex:
us-east-1

     _sha256 = SHA256.Create();
     _emptyPayloadHash = Hash(Array.Empty<byte>());
   }

   public IAuthToken GetAuthToken(string url)
   {
     return AuthTokens.Custom(DummyUserName, GetCredentials(url), Realm, Scheme);
   }

   /******************* AWS SIGNING FUNCTIONS *********************/
   private string Hash(byte[] bytesToHash)
   {
     return ToHexString(_sha256.ComputeHash(bytesToHash));
```

```
    }

    private static byte[] HmacSHA256(byte[] key, string data)
    {
      return new HMACSHA256(key).ComputeHash(Encoding.UTF8.GetBytes(data));
    }

    private byte[] GetSignatureKey(string dateStamp)
    {
      var kSecret = Encoding.UTF8.GetBytes($"AWS4{_secretKey}");
      var kDate = HmacSHA256(kSecret, dateStamp);
      var kRegion = HmacSHA256(kDate, _region);
      var kService = HmacSHA256(kRegion, ServiceName);
      return HmacSHA256(kService, AWSRequest);
    }

    private static string ToHexString(byte[] array)
    {
      return Convert.ToHexString(array).ToLowerInvariant();
    }

    private string GetCredentials(string url)
    {
      var request = new HttpRequestMessage
      {
        Method = HttpMethod.Get,
        RequestUri = new Uri($"https://{url}/opencypher")
      };

      var signedrequest = Sign(request);

      var headers = new Dictionary<string, object>
      {
        [HeaderKeys.AuthorizationHeader] =
 signedrequest.Headers.GetValues(HeaderKeys.AuthorizationHeader).FirstOrDefault(),
        ["HttpMethod"] = HttpMethod.Get.ToString(),
        [HeaderKeys.XAmzDateHeader] =
 signedrequest.Headers.GetValues(HeaderKeys.XAmzDateHeader).FirstOrDefault(),
        // Host should be capitalized, not like in Amazon.Util.HeaderKeys.HostHeader
        ["Host"] =
 signedrequest.Headers.GetValues(HeaderKeys.HostHeader).FirstOrDefault(),
      };

      return JsonSerializer.Serialize(headers);
```

```
    }

    private HttpRequestMessage Sign(HttpRequestMessage request)
    {
      var now = DateTimeOffset.UtcNow;
      var amzdate = now.ToString("yyyyMMddTHHmmssZ");
      var datestamp = now.ToString("yyyyMMdd");

      if (request.Headers.Host == null)
      {
        request.Headers.Host = $"{request.RequestUri.Host}:{request.RequestUri.Port}";
      }

      request.Headers.Add(HeaderKeys.XAmzDateHeader, amzdate);

      var canonicalQueryParams = GetCanonicalQueryParams(request);

      var canonicalRequest = new StringBuilder();
      canonicalRequest.Append(request.Method + "\n");
      canonicalRequest.Append(request.RequestUri.AbsolutePath + "\n");
      canonicalRequest.Append(canonicalQueryParams + "\n");

      var signedHeadersList = new List<string>();
      foreach (var header in request.Headers.OrderBy(a => a.Key.ToLowerInvariant()))
      {
        canonicalRequest.Append(header.Key.ToLowerInvariant());
        canonicalRequest.Append(':');
        canonicalRequest.Append(string.Join(",", header.Value.Select(s => s.Trim())));
        canonicalRequest.Append('\n');
        signedHeadersList.Add(header.Key.ToLowerInvariant());
      }
      canonicalRequest.Append('\n');

      var signedHeaders = string.Join(";", signedHeadersList);
      canonicalRequest.Append(signedHeaders + "\n");
      canonicalRequest.Append(_emptyPayloadHash);

      var credentialScope = $"{datestamp}/{_region}/{ServiceName}/{AWSRequest}";
      var stringToSign = $"{Algorithm}\n{amzdate}\n{credentialScope}\n"
        + Hash(Encoding.UTF8.GetBytes(canonicalRequest.ToString()));

      var signing_key = GetSignatureKey(datestamp);
      var signature = ToHexString(HmacSHA256(signing_key, stringToSign));
```

```
      request.Headers.TryAddWithoutValidation(HeaderKeys.AuthorizationHeader,
        $"{Algorithm} Credential={_accessKey}/{credentialScope},
 SignedHeaders={signedHeaders}, Signature={signature}");

      return request;
    }

    private static string GetCanonicalQueryParams(HttpRequestMessage request)
    {
      var querystring = HttpUtility.ParseQueryString(request.RequestUri.Query);

      // Query params must be escaped in upper case (i.e. "%2C", not "%2c").
      var queryParams = querystring.AllKeys.OrderBy(a => a)
        .Select(key => $"{key}={Uri.EscapeDataString(querystring[key])}");
      return string.Join("&", queryParams);
    }
  }
}
```

Here is how to make an openCypher query in .NET using Bolt with IAM authentication. The example below uses the `NeptuneAuthToken` helper:

```
using Neo4j.Driver;

namespace Hello
{
  public class HelloWorldExample
  {
    private const string Host = "(your hostname):8182";
    private const string Url = $"bolt://{Host}";
    private const string CreateNodeQuery = "CREATE (a:Greeting) SET a.message =
 'HelloWorldExample'";
    private const string ReadNodeQuery = "MATCH(n:Greeting) RETURN n.message";

    private const string AccessKey = "(your access key)";
    private const string SecretKey = "(your secret key)";
    private const string Region = "(your AWS region)"; // e.g. "us-west-2"

    private readonly IDriver _driver;

    public HelloWorldExample()
    {
```

```
     var authToken = new NeptuneAuthToken(AccessKey, SecretKey,
Region).GetAuthToken(Host);

     // Note that when the connection is reinitialized after max connection lifetime
     // has been reached, the signature token could have already been expired (usually
5 min)
     // You can face exceptions like:
     //   `Unexpected server exception 'Signature expired: XXXX is now earlier than
YYYY (ZZZZ - 5 min.)`
     _driver = GraphDatabase.Driver(Url, authToken, o =>

o.WithMaxConnectionLifetime(TimeSpan.FromMinutes(60)).WithEncryptionLevel(EncryptionLevel.Encr
   }

   public async Task CreateNode()
   {
     // Open a session
     using (var session = _driver.AsyncSession())
     {
       // Run the query in a write transaction
       var greeting = await session.WriteTransactionAsync(async tx =>
       {
         var result = await tx.RunAsync(CreateNodeQuery);
         // Consume the result
         return await result.ConsumeAsync();
       });

       // The output will look like this:
       //   ResultSummary{Query=`CREATE (a:Greeting) SET a.message =
'HelloWorldExample".....
       Console.WriteLine(greeting.Query);
     }
   }

   public async Task RetrieveNode()
   {
     // Open a session
     using (var session = _driver.AsyncSession())
     {
       // Run the query in a read transaction
       var greeting = await session.ReadTransactionAsync(async tx =>
       {
         var result = await tx.RunAsync(ReadNodeQuery);
         var records = await result.ToListAsync();
```

```
          // Consume the result. Read the single node
          // created in a previous step.
          return records[0].Values.First().Value;
        });
        // The output will look like this:
        //   HelloWorldExample
        Console.WriteLine(greeting);
      }
    }
  }
}
```

This example can be launched by running the code below on `.NET 6` or `.NET 7` with the following packages:

- **Neo4j**`.Driver=4.3.0`
- **AWSSDK**`.Core=3.7.102.1`

```
namespace Hello
{
  class Program
  {
    static async Task Main()
    {
      var apiCaller = new HelloWorldExample();

      await apiCaller.CreateNode();
      await apiCaller.RetrieveNode();
    }
  }
}
```

A Golang openCypher query example using Bolt with IAM authentication

The following example shows how to make openCypher queries in Go using the Bolt protocol with IAM authentication. It uses the AWS SDK for Go v2 for SigV4 signing and the Neo4j Go driver v5 with an `AuthTokenManager` struct that implements the Neo4j Go driver's token manager interface (`github.com/neo4j/neo4j-go-driver/v5/neo4j/auth.TokenManager`) to automatically refresh credentials before they expire.

First, create an `AuthTokenManager` that generates SigV4-signed tokens. Save this as `auth_token_manager.go`:

```
// AuthTokenManager for Amazon Neptune IAM authentication via the Bolt protocol.
// Provides SigV4-signed credentials to the Neo4j driver's auth interface.
package main

import (
 "context"
 "encoding/json"
 "fmt"
 "net/http"
 "reflect"
 "sync"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/aws/signer/v4"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/credentials/stscreds"
 "github.com/aws/aws-sdk-go-v2/service/sts"
 "github.com/neo4j/neo4j-go-driver/v5/neo4j"
 "github.com/neo4j/neo4j-go-driver/v5/neo4j/db"
)

const (
 serviceName      = "neptune-db"
 emptyPayloadHash = "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
)

// AuthTokenManager manages SigV4-signed authentication tokens for Neptune with
 automatic refresh.
type AuthTokenManager struct {
 region          string
 endpoint        string
 refreshInterval time.Duration
 credentials     aws.CredentialsProvider
 mutex           sync.Mutex
 cachedToken     neo4j.AuthToken
 tokenTime       time.Time
}

// NewAuthTokenManager creates a new AuthTokenManager.
```

```
//
// Parameters:
//   - region: AWS region (e.g., "us-east-1")
//   - endpoint: Neptune endpoint with port (e.g.,
 "cluster.region.neptune.amazonaws.com:8182")
//   - profile: AWS profile name (optional, pass "" to use default)
//   - roleArn: AWS role ARN to assume (optional, pass "" to skip)
//   - refreshInterval: Token refresh interval
func NewAuthTokenManager(region, endpoint, profile, roleArn string, refreshInterval
 time.Duration) (*AuthTokenManager, error) {
 credentials, err := createCredentialsProvider(region, profile, roleArn)
 if err != nil {
  return nil, err
 }
 return &AuthTokenManager{
  region:          region,
  endpoint:        endpoint,
  refreshInterval: refreshInterval,
  credentials:     credentials,
 }, nil
}

// createCredentialsProvider builds an AWS credentials provider, optionally using
// a named profile and/or assuming a role.
func createCredentialsProvider(region, profile, roleArn string)
 (aws.CredentialsProvider, error) {
 ctx := context.Background()
 var opts []func(*config.LoadOptions) error
 opts = append(opts, config.WithRegion(region))
 if profile != "" {
  opts = append(opts, config.WithSharedConfigProfile(profile))
 }
 cfg, err := config.LoadDefaultConfig(ctx, opts...)
 if err != nil {
  return nil, fmt.Errorf("failed to load AWS config: %w", err)
 }
 var credentials aws.CredentialsProvider = cfg.Credentials
 if roleArn != "" {
  stsClient := sts.NewFromConfig(cfg)
  credentials = stscreds.NewAssumeRoleProvider(stsClient, roleArn, func(o
 *stscreds.AssumeRoleOptions) {
   o.RoleSessionName = "NeptuneAuthSession"
   o.Duration = 900 * time.Second
  })
```

```
 }
 return credentials, nil
}

// GetAuthToken returns a valid authentication token, using cached token if still
 valid.
func (m *AuthTokenManager) GetAuthToken(ctx context.Context) (neo4j.AuthToken, error) {
 m.mutex.Lock()
 defer m.mutex.Unlock()

 if time.Since(m.tokenTime) < m.refreshInterval && m.cachedToken.Tokens != nil {
  return m.cachedToken, nil
 }

 token, err := m.generateToken(ctx)
 if err != nil {
  return neo4j.AuthToken{}, err
 }

 m.cachedToken = token
 m.tokenTime = time.Now()
 return token, nil
}

// HandleSecurityException handles security exceptions by invalidating the cached
// token (if it matches the token that caused the error) and returning true so
// the driver retries with a fresh token. The comparison prevents a concurrent
// retry from unnecessarily invalidating a freshly generated token.
func (m *AuthTokenManager) HandleSecurityException(ctx context.Context, token
 neo4j.AuthToken, err *db.Neo4jError) (bool, error) {
 m.mutex.Lock()
 defer m.mutex.Unlock()
 if reflect.DeepEqual(m.cachedToken.Tokens, token.Tokens) {
  m.tokenTime = time.Time{}
  m.cachedToken = neo4j.AuthToken{}
 }
 return true, nil
}

// generateToken generates a new SigV4-signed authentication token for Neptune.
func (m *AuthTokenManager) generateToken(ctx context.Context) (neo4j.AuthToken, error)
 {
 req, err := http.NewRequest(http.MethodGet, "https://"+m.endpoint+"/opencypher", nil)
 if err != nil {
```

```
  return neo4j.AuthToken{}, err
 }
 req.Host = m.endpoint

 creds, err := m.credentials.Retrieve(ctx)
 if err != nil {
  return neo4j.AuthToken{}, err
 }

 signer := v4.NewSigner()
 err = signer.SignHTTP(ctx, creds, req, emptyPayloadHash, serviceName, m.region,
 time.Now())
 if err != nil {
  return neo4j.AuthToken{}, err
 }

 authData := map[string]string{
  "Authorization": req.Header.Get("Authorization"),
  "X-Amz-Date":    req.Header.Get("X-Amz-Date"),
  "Host":          m.endpoint,
  "HttpMethod":    req.Method,
 }

 if st := req.Header.Get("X-Amz-Security-Token"); st != "" {
  authData["X-Amz-Security-Token"] = st
 }

 authJSON, err := json.Marshal(authData)
 if err != nil {
  return neo4j.AuthToken{}, err
 }

 return neo4j.BasicAuth("username", string(authJSON), ""), nil
}
```

Then use the token manager to create a driver and find a node by ID. Note the use of a parameterized query ($nodeId) instead of string interpolation:

```
package main

import (
 "context"
 "fmt"
```

```
 "log"
 "time"

 "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func findNode(ctx context.Context, driver neo4j.DriverWithContext, nodeId string)
 (string, error) {
 session := driver.NewSession(ctx, neo4j.SessionConfig{
  AccessMode: neo4j.AccessModeRead,
 })
 defer session.Close(ctx)

 // Use parameterized queries to prevent injection and enable query plan caching.
 result, err := session.Run(ctx,
  "MATCH (n) WHERE ID(n) = $nodeId RETURN n",
  map[string]any{"nodeId": nodeId},
 )
 if err != nil {
  return "", fmt.Errorf("error running query: %v", err)
 }

 if !result.Next(ctx) {
  if err = result.Err(); err != nil {
   return "", fmt.Errorf("error fetching result: %v", err)
  }
  return "", fmt.Errorf("node not found")
 }

 n, found := result.Record().Get("n")
 if !found {
  return "", fmt.Errorf("node not found")
 }

 return fmt.Sprintf("%+v", n), nil
}

func main() {
 region := "(your AWS region)"                  // e.g. "us-east-1"
 endpoint := "(your Neptune endpoint):8182"     // e.g. "cluster.xxx.us-
east-1.neptune.amazonaws.com:8182"

 ctx := context.Background()
```

```
 // Pass a profile name for local development, or a role ARN for cross-account access
 authManager, err := NewAuthTokenManager(region, endpoint, "", "", 4*time.Minute) //
 Refresh before the 5-minute SigV4 signature expiry
 if err != nil {
  log.Fatalf("auth manager error: %v", err)
 }

 // bolt+s:// enables TLS with full certificate verification.
 // If you're developing on macOS, use bolt+ssc:// instead. Go on macOS uses the
 // system TLS verifier, which requires Certificate Transparency compliance that
 // Neptune endpoints don't support. In production (typically Linux), bolt+s://
 // works with system CA certificates.
 driver, err := neo4j.NewDriverWithContext("bolt+s://"+endpoint, authManager)
 if err != nil {
  log.Fatalf("driver error: %v", err)
 }
 defer driver.Close(ctx)

 if err = driver.VerifyConnectivity(ctx); err != nil {
  log.Fatalf("connectivity error: %v", err)
 }

 // Neptune assigns UUID-format node IDs if a user does not supply their own node IDs
 res, err := findNode(ctx, driver, "72c2e8c1-7d5f-5f30-10ca-9d2bb8c4afbc")
 if err != nil {
  log.Fatal(err)
 }
 fmt.Println(res)
}
```

Note

The `auth.TokenManager` interface (`github.com/neo4j/neo4j-go-driver/v5/neo4j/auth`) used in this example became generally available in Neo4j Go driver **v5.14.0**. This interface enables automatic credential refresh, which is necessary for Neptune IAM authentication since SigV4 signatures are only valid for a short period and must be regenerated when the driver establishes new connections.

This example was validated using the following Go modules:

```
require (
```

```
    github.com/aws/aws-sdk-go-v2         v1.30.3
    github.com/aws/aws-sdk-go-v2/config   v1.27.27
    github.com/aws/aws-sdk-go-v2/credentials v1.17.27
    github.com/aws/aws-sdk-go-v2/service/sts v1.30.3
    github.com/neo4j/neo4j-go-driver/v5   v5.22.0
  )
```

Bolt connection behavior in Neptune

Here are some things to keep in mind about Neptune Bolt connections:

- Because Bolt connections are created at the TCP layer, you can't use an Application Load Balancer in front of them, as you can with an HTTP endpoint.
- The port that Neptune uses for Bolt connections is your DB cluster's port.
- Based on the Bolt preamble passed to it, the Neptune server selects the highest appropriate Bolt version (1, 2, 3, or 4.0).
- The maximum number of connections to the Neptune server that a client can have open at any point in time is 1,000.
- If the client doesn't close a connection after a query, that connection can be used to execute the next query.
- However, if a connection is idle for 20 minutes, the server closes it automatically.
- If IAM authentication is not enabled, you can use `AuthTokens.none()` rather than supplying a dummy user name and password. For example, in Java:

```
GraphDatabase.driver("bolt://(your cluster endpoint URL):(your cluster port)",
 AuthTokens.none(),

 Config.builder().withEncryption().withTrustStrategy(TrustStrategy.trustSystemCertificates())
```

- When IAM authentication is enabled, a Bolt connection is always disconnected a few minutes more than 10 days after it was established if it hasn't already closed for some other reason.
- If the client sends a query for execution over a connection without having consumed the results of a previous query, the new query is discarded. To discard the previous results instead, the client must send a reset message over the connection.
- Only one transaction at a time can be created on a given connection.
- If an exception occurs during a transaction, the Neptune server rolls back the transaction and closes the connection. In this case, the driver creates a new connection for the next query.

- Be aware that sessions are not thread-safe. Multiple parallel operations must use multiple separate sessions.

Examples of openCypher parameterized queries

Neptune supports parameterized openCypher queries. This lets you use the same query structure multiple times with different arguments. Since the query structure doesn't change, Neptune can cache its abstract syntax tree (AST) rather than having to parse it multiple times.

Example of an openCypher parameterized query using the HTTPS endpoint

Below is an example of using a parameterized query with the Neptune openCypher HTTPS endpoint. The query is:

```
MATCH (n {name: $name, age: $age})
RETURN n
```

The parameters are defined as follows:

```
parameters={"name": "john", "age": 20}
```

You can submit the parameterized query like this:

AWS CLI

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "MATCH (n {name: \$name, age: \$age}) RETURN n" \
  --parameters '{"name": "john", "age": 20}'
```

For more information, see execute-open-cypher-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
```

```
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_query(
    openCypherQuery='MATCH (n {name: $name, age: $age}) RETURN n',
    parameters='{"name": "john", "age": 20}'
)

print(response['results'])
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=MATCH (n {name: \$name, age: \$age}) RETURN n" \
  -d 'parameters={"name": "john", "age": 20}'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

Using POST:

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=MATCH (n {name: \$name, age: \$age}) RETURN n" \
  -d "parameters={\"name\": \"john\", \"age\": 20}"
```

Using GET (URL-encoded):

```
curl -X GET \
  "https://your-neptune-endpoint:port/openCypher?query=MATCH%20%28n%20%7Bname:
\$name,age:\$age%7D%29%20RETURN%20n&parameters=%7B%22name%22:%22john%22,%22age
%22:20%7D"
```

Using `DIRECT POST`:

```
curl -H "Content-Type: application/opencypher" \
  "https://your-neptune-endpoint:port/openCypher?parameters=%7B%22name%22:%22john
%22,%22age%22:20%7D" \
  -d "MATCH (n {name: \$name, age: \$age}) RETURN n"
```

Examples of openCypher parameterized queries using Bolt

Here is a Python example of an openCypher parameterized query using the Bolt protocol:

```
from neo4j import GraphDatabase
uri = "bolt://[neptune-endpoint-url]:8182"
driver = GraphDatabase.driver(uri, auth=("", ""))

def match_name_and_age(tx, name, age):
  # Parameterized Query
  tx.run("MATCH (n {name: $name, age: $age}) RETURN n", name=name, age=age)

with driver.session() as session:
  # Parameters
  session.read_transaction(match_name_and_age, "john", 20)

driver.close()
```

Here is a Java example of an openCypher parameterized query using the Bolt protocol:

```
Driver driver = GraphDatabase.driver("bolt+s://(your cluster endpoint URL):8182");
HashMap<String, Object> parameters = new HashMap<>();
parameters.put("name", "john");
parameters.put("age", 20);
String queryString = "MATCH (n {name: $name, age: $age}) RETURN n";
Result result = driver.session().run(queryString, parameters);
```

openCypher data model

The Neptune openCypher engine builds on the same property-graph model as Gremlin. In particular:

- Every node has one or more labels. If you insert a node without labels, a default label named `vertex` is attached. If you try to delete all of a node's labels, an error is thrown.

- A relationship is an entity that has exactly one relationship type and that forms a unidirectional connection between two nodes (that is, *from* one of the nodes *to* the other).
- Both nodes and relationships can have properties, but don't have to. Neptune supports nodes and relationships with zero properties.
- Neptune does not support metaproperties, which are not included in the openCypher specification either.
- Properties in your graph can be multi-valued if they were created using Gremlin. That is a node or relationship property can have a set of different values rather than only one. Neptune has extended openCypher semantics to handle multi-valued properties gracefully.

Supported data types are documented in openCypher data format. However, we do not recommend inserting `Array` property values into an openCypher graph at present. Although it is possible to insert an array property value using the bulk loader, the current Neptune openCypher release treats it as a set of multi-valued properties instead of as a single list value.

Below is the list of data types supported in this release:

- `Bool`
- `Byte`
- `Short`
- `Int`
- `Long`
- `Float` (Includes plus and minus Infinity and NaN, but not INF)
- `Double` (Includes plus and minus Infinity and NaN, but not INF)
- `DateTime`
- `String`

The openCypher `explain` feature

The openCypher `explain` feature is a self-service tool in Amazon Neptune that helps you understand the execution approach taken by the Neptune engine. To invoke explain, you pass a parameter to an openCypher HTTPS request with `explain=`*`mode`*, where the *mode* value can be one of the following:

- **`static`** – In `static` mode, `explain` prints only the static structure of the query plan. It doesn't actually run the query.
- **`dynamic`** – In `dynamic` mode, `explain` also runs the query, and includes dynamic aspects of the query plan. These may include the number of intermediate bindings flowing through the operators, the ratio of incoming bindings to outgoing bindings, and the total time taken by each operator.
- **`details`** – In `details` mode, `explain` prints the information shown in dynamic mode plus additional details, such as the actual openCypher query string and the estimated range count for the pattern underlying a join operator.

For example, using `POST` with `dynamic` mode:

AWS CLI

```
aws neptunedata execute-open-cypher-explain-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "MATCH (n) RETURN n LIMIT 1" \
  --explain-mode dynamic
```

For more information, see execute-open-cypher-explain-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_explain_query(
    openCypherQuery='MATCH (n) RETURN n LIMIT 1',
    explainMode='dynamic'
)
```

```
print(response['results'].read().decode('utf-8'))
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=MATCH (n) RETURN n LIMIT 1" \
  -d "explain=dynamic"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=MATCH (n) RETURN n LIMIT 1" \
  -d "explain=dynamic"
```

Limitations for openCypher `explain` in Neptune

The current release of openCypher explain has the following limitations:

- Explain plans are currently only available for queries that perform read-only operations. Queries that perform any sort of mutation, such as `CREATE`, `DELETE`, `MERGE`, `SET` and so on, are not supported.
- Operators and output for a specific plan may change in future releases.

DFE operators in openCypher `explain` output

To use the information that the openCypher `explain` feature provides, you need to understand some details about how the DFE query engine works (DFE being the engine that Neptune uses to process openCypher queries).

The DFE engine translates every query into a pipeline of operators. Starting from the first operator, intermediate solutions flow from one operator to the next through this operator pipeline. Each row in the explain table represents a result, up to the point of evaluation.

The operators that can appear in a DFE query plan are as follows:

DFEApply – Executes the function specified in the arguments section, on the value stored in the specified variable

DFEBindRelation – Binds together variables with the specified names

DFEChunkLocalSubQuery – This is a non-blocking operation that acts as a wrapper around subqueries being performed.

DFEDistinctColumn – Returns the distinct subset of the input values based on the variable specified.

DFEDistinctRelation – Returns the distinct subset of the input solutions based on the variable specified.

DFEDrain – Appears at the end of a subquery to act as a termination step for that subquery. The number of solutions is recorded as `Units In`. `Units Out` is always zero.

DFEForwardValue – Copies all input chunks directly as output chunks to be passed to its downstream operator.

DFEGroupByHashIndex – Performs a group-by operation over the input solutions based on a previously computed hash index (using the `DFEHashIndexBuild` operation). As an output, the given input is extended by a column containing a group key for every input solution.

DFEHashIndexBuild – Builds a hash index over a set of variables as a side-effect. This hash index is typically reused in later operations. See `DFEHashIndexJoin` or `DFEGroupByHashIndex` for where this hash index might be used.

DFEHashIndexJoin – Performs a join over the incoming solutions against a previously built hash index. See `DFEHashIndexBuild` for where this hash index might be built.

DFEJoinExists – Takes a left and right hand input relation, and retains values from the left relation that have a corresponding value in the right relation as defined by the given join variables.

– This is a non-blocking operation that acts as a wrapper for a subquery, allowing it to be run repeatedly for use in loops.

DFEMergeChunks – This is a blocking operation that combines chunks from its upstream operator into a single chunk of solutions to pass to its downstream operator (inverse of `DFESplitChunks`).

DFEMinus – Takes a left and right hand input relation, and retains values from the left relation that do not have a corresponding value in the right relation as defined by the given join variables. If there is no overlap in variables across both relations, then this operator simply returns the left hand input relation.

DFENotExists – Takes a left and right hand input relation, and retains values from the left relation that do not have a corresponding value in the right relation as defined by the given join variables. If there is no overlap in variables across both relations, then this operator returns an empty relation.

DFEOptionalJoin – Performs a left outer join (also called OPTIONAL join): solutions from the left hand side that have at least one join partner in the right-hand side are joined, and solutions from the left-hand side without join partner in the right-hand side are forwarded as is. This is a blocking operation.

DFEPipelineJoin – Joins the input against the tuple pattern defined by the `pattern` argument.

DFEPipelineRangeCount – Counts the number of solutions matching a given pattern, and returns a single one-ary solution containing the count value.

DFEPipelineScan – Scans the database for the given `pattern` argument, with or without a given filter on column(s).

DFEProject – Takes multiple input columns and projects only the desired columns.

DFEReduce – Performs the specified aggregation function on specified variables.

DFERelationalJoin – Joins the input of the previous operator based on the specified pattern keys using a merge join. This is a blocking operation.

DFERouteChunks – Takes input chunks from its singular incoming edge and routes those chunks along its multiple outgoing edges.

DFESelectRows – This operator selectively takes rows from its left input relation solutions to forward to its downstream operator. The rows selected based on the row identifiers supplied in the operator's right input relation.

DFESerialize – Serializes a query's final results into a JSON string serialization, mapping each input solution to the appropriate variable name. For node and edge results, these results are serialized into a map of entity properties and metadata.

DFESort – Takes an input relation and produces a sorted relation based on the provided sort key.

DFESplitByGroup – Splits each single input chunk from one incoming edge into smaller output chunks corresponding to row groups identified by row IDs from the corresponding input chunk from the other incoming edge.

DFESplitChunks – Splits each single input chunk into smaller output chunks (inverse of `DFEMergeChunks`).

DFEStreamingHashIndexBuild – Streaming version of `DFEHashIndexBuild`.

DFEStreamingGroupByHashIndex – Streaming version of `DFEGroupByHashIndex`.

DFESubquery – This operator appears at the beginning of all plans and encapsulates the portions of the plan that are run on the DFE engine, which is the entire plan for openCypher.

DFESymmetricHashJoin – Joins the input of the previous operator based on the specified pattern keys using a hash join. This is a non-blocking operation.

DFESync – This operator is a synchronization operator supporting non-blocking plans. It takes solutions from two incoming edges and forwards these solutions to the appropriate downstream edges. For synchronization purposes, the inputs along one of these edges may be buffered internally.

DFETee – This is a branching operator that sends the same set of solutions to multiple operators.

DFETermResolution – Performs a localize or globalize operation on its inputs, resulting in columns of either localized or globalized identifiers respectively.

- Unfolds lists of values from an input column into the output column as individual elements.

DFEUnion – Takes two or more input relations and produces a union of those relations using the desired output schema.

SolutionInjection – Appears before everything else in the explain output, with a value of 1 in the Units Out column. However, it serves as a no-op, and doesn't actually inject any solutions into the DFE engine.

TermResolution – Appears at the end of plans and translates objects from the Neptune engine into openCypher objects.

Columns in openCypher `explain` output

The query plan information that Neptune generates as openCypher explain output contains tables with one operator per row. The table has the following columns:

ID – The numeric ID of this operator in the plan.

Out #1 (and **Out #2**) – The ID(s) of operator(s) that are downstream from this operator. There can be at most two downstream operators.

Name – The name of this operator.

Arguments – Any relevant details for the operator. This includes things like input schema, output schema, pattern (for `PipelineScan` and `PipelineJoin`), and so on.

Mode – A label describing fundamental operator behavior. This column is mostly blank (`-`). One exception is `TermResolution`, where mode can be `id2value_opencypher`, indicating a resolution from ID to openCypher value.

Units In – The number of solutions passed as input to this operator. Operators without upstream operators, such as `DFEPipelineScan`, `SolutionInjections`, and a `DFESubquery` with no static value injected, would have zero value.

Units Out – The number of solutions produced as output of this operator. `DFEDrain` is a special case, where the number of solutions being drained is recorded in `Units In` and `Units Out` is always zero.

Ratio – The ratio of `Units Out` to `Units In`.

Time (ms) – The CPU time consumed by this operator, in milliseconds.

A basic example of openCypher explain output

The following is a basic example of openCypher `explain` output. The query is a single-node lookup in the air routes dataset for a node with the airport code `ATL` that invokes `explain` using the `details` mode in default ASCII output format.

To invoke `explain` for this query:

AWS CLI

```
aws neptunedata execute-open-cypher-explain-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "MATCH (n {code: 'ATL'}) RETURN n" \
  --explain-mode details
```

For more information, see execute-open-cypher-explain-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_explain_query(
    openCypherQuery="MATCH (n {code: 'ATL'}) RETURN n",
    explainMode='details'
)

print(response['results'].read().decode('utf-8'))
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=MATCH (n {code: 'ATL'}) RETURN n" \
  -d "explain=details"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=MATCH (n {code: 'ATL'}) RETURN n" \
  -d "explain=details"
```

The `explain` output:

```
Query:
MATCH (n {code: 'ATL'}) RETURN n

#######################################################################################
# ID # Out #1 # Out #2 # Name              # Arguments          # Mode               #
 Units In # Units Out # Ratio # Time (ms) #
#######################################################################################
# 0  # 1      # -      # SolutionInjection # solutions=[{}]     # -                  #
 0        # 1         # 0.00  # 0         #
#######################################################################################
# 1  # 2      # -      # DFESubquery       # subQuery=subQuery1 # -                  #
 0        # 1         # 0.00  # 4.00      #
#######################################################################################
# 2  # -      # -      # TermResolution    # vars=[?n]          # id2value_opencypher #
 1        # 1         # 1.00  # 2.00      #
#######################################################################################



subQuery1
#######################################################################################
# ID # Out #1 # Out #2 # Name                  # Arguments
                                                                    # Mode # Units
 In # Units Out # Ratio # Time (ms) #
#######################################################################################
# 0  # 1      # -      # DFEPipelineScan       # pattern=Node(?n) with property 'code'
 as ?n_code2 and label 'ALL'                                        # -    # 0
   # 1         # 0.00  # 0.21      #
```

```
#    #        #        #                       # inlineFilters=[(?n_code2 IN
 ["ATL"^^xsd:string])]                                                      #
  #          #           #       #           #
#    #        #        #                       # patternEstimate=1
                                                                    #      #
  #           #       #           #
#################################################################################
# 1  # 2      # -      # DFEChunkLocalSubQuery # subQuery=http://aws.amazon.com/
neptune/vocab/v01/dfe/past/graph#9d84f97c-c3b0-459a-98d5-955a8726b159/graph_1 # -    #
 1        # 1         # 1.00  # 0.04      #
#################################################################################
# 2  # 3      # -      # DFEProject            # columns=[?n]
                                                                    # -    # 1
  # 1         # 1.00  # 0.04      #
#################################################################################
# 3  # -      # -      # DFEDrain              # -
                                                                    # -    # 1
  # 0         # 0.00  # 0.03      #
#################################################################################


subQuery=http://aws.amazon.com/neptune/vocab/v01/dfe/past/graph#9d84f97c-
c3b0-459a-98d5-955a8726b159/graph_1
#################################################################################
# ID # Out #1 # Out #2 # Name                 # Arguments
                    # Mode # Units In # Units Out # Ratio # Time (ms) #
#################################################################################
# 0  # 1      # -      # DFESolutionInjection # outSchema=[?n, ?n_code2]
                    # -    # 0        # 1         # 0.00  # 0.02      #
#################################################################################
# 1  # 2      # 3      # DFETee               # -
                    # -    # 1        # 2         # 2.00  # 0.02      #
#################################################################################
# 2  # 4      # -      # DFEDistinctColumn    # column=?n
                    # -    # 1        # 1         # 1.00  # 0.20      #
#    #        #        #                     # ordered=false
                    #      #          #           #       #           #
#################################################################################
# 3  # 5      # -      # DFEHashIndexBuild    # vars=[?n]
                    # -    # 1        # 1         # 1.00  # 0.04      #
#################################################################################
# 4  # 5      # -      # DFEPipelineJoin      # pattern=Node(?n) with property 'ALL'
 and label '?n_label1' # -    # 1        # 1         # 1.00  # 0.25      #
```

```
#    #        #        #                      # patternEstimate=3506
                     #      #          #           #       #           #
##########################################################################################
# 5  # 6      # 7      # DFESync               # -
                     # -    # 2        # 2         # 1.00  # 0.02      #
##########################################################################################
# 6  # 8      # -      # DFEForwardValue       # -
                     # -    # 1        # 1         # 1.00  # 0.01      #
##########################################################################################
# 7  # 8      # -      # DFEForwardValue       # -
                     # -    # 1        # 1         # 1.00  # 0.01      #
##########################################################################################
# 8  # 9      # -      # DFEHashIndexJoin      # -
                     # -    # 2        # 1         # 0.50  # 0.35      #
##########################################################################################
# 9  # -      # -      # DFEDrain              # -
                     # -    # 1        # 0         # 0.00  # 0.02      #
##########################################################################################
```

At the top-level, `SolutionInjection` appears before everything else, with 1 unit out. Note that it doesn't actually inject any solutions. You can see that the next operator, `DFESubquery`, has 0 units in.

After `SolutionInjection` at the top-level are `DFESubquery` and `TermResolution` operators. `DFESubquery` encapsulates the parts of the query execution plan that is being pushed to the DFE engine (for openCypher queries, the entire query plan is executed by the DFE). All the operators in the query plan are nested inside `subQuery1` that is referenced by `DFESubquery`. The only exception is `TermResolution`, which materializes internal IDs into fully serialized openCypher objects.

All the operators that are pushed down to the DFE engine have names that start with a `DFE` prefix. As mentioned above, the whole openCypher query plan is executed by the DFE, so as a result, all the operators except the final `TermResolution` operator start with `DFE`.

Inside `subQuery1`, there can be zero or more `DFEChunkLocalSubQuery` or `DFELoopSubQuery` operators that encapsulate a part of the pushed execution plan that is executed in a memory-bounded mechanism. `DFEChunkLocalSubQuery` here contains one `SolutionInjection` that is used as an input to the subquery. To find the table for that subquery in the output, search for the `subQuery=`*`graph URI`* specified in the `Arguments` column for the `DFEChunkLocalSubQuery` or `DFELoopSubQuery` operator.

In `subQuery1`, `DFEPipelineScan` with `ID 0` scans the database for a specified `pattern`. The pattern scans for an entity with property `code` saved as a variable `?n_code2` over all labels (you could filter on a specific label by appending `airport` to `n:airport`). The `inlineFilters` argument shows the filtering for the `code` property equalling `ATL`.

Next, the `DFEChunkLocalSubQuery` operator joins the intermediate results of a subquery that contains `DFEPipelineJoin`. This ensures that `?n` is actually a node, since the previous `DFEPipelineScan` scans for any entity with the `code` property.

Example of `explain` output for a relationship lookup with a limit

This query looks for relationships between two anonymous nodes with type `route`, and returns at most 10. Again, the `explain` mode is `details` and the output format is the default ASCII format.

Here, `DFEPipelineScan` scans for edges that start from anonymous node `?anon_node7` and end at another anonymous node `?anon_node21`, with a relationship type saved as `?p_type1`. There is a filter for `?p_type1` being `el://route` (where `el` stands for edge label), which corresponds to `[p:route]` in the query string.

`DFEDrain` collects the output solution with a limit of 10, as shown in its `Arguments` column. `DFEDrain` terminates once the limit is reached or the all solutions are produced, whichever happens first.

To invoke `explain` for this query:

AWS CLI

```
aws neptunedata execute-open-cypher-explain-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "MATCH ()-[p:route]->() RETURN p LIMIT 10" \
  --explain-mode details
```

For more information, see execute-open-cypher-explain-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config
```

```
client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_explain_query(
    openCypherQuery='MATCH ()-[p:route]->() RETURN p LIMIT 10',
    explainMode='details'
)

print(response['results'].read().decode('utf-8'))
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=MATCH ()-[p:route]->() RETURN p LIMIT 10" \
  -d "explain=details"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=MATCH ()-[p:route]->() RETURN p LIMIT 10" \
  -d "explain=details"
```

The `explain` output:

```
Query:
MATCH ()-[p:route]->() RETURN p LIMIT 10
```

```
##########################################################################################
# ID # Out #1 # Out #2 # Name              # Arguments          # Mode                #
 Units In # Units Out # Ratio # Time (ms) #
##########################################################################################
# 0  # 1      # -      # SolutionInjection # solutions=[{}]     # -                   #
 0        # 1         # 0.00  # 0         #
##########################################################################################
# 1  # 2      # -      # DFESubquery       # subQuery=subQuery1 # -                   #
 0        # 10        # 0.00  # 5.00      #
##########################################################################################
# 2  # -      # -      # TermResolution    # vars=[?p]          # id2value_opencypher #
 10       # 10        # 1.00  # 1.00      #
##########################################################################################



subQuery1
##########################################################################################
# ID # Out #1 # Out #2 # Name            # Arguments
              # Mode # Units In # Units Out # Ratio # Time (ms) #
##########################################################################################
# 0  # 1      # -      # DFEPipelineScan # pattern=Edge((?anon_node7)-[?p:?p_type1]->(?
anon_node21)) # -    # 0        # 1000      # 0.00  # 0.66      #
#    #        #        #                 # inlineFilters=[(?p_type1 IN [<el://route>])]
              #      #          #           #       #           #
#    #        #        #                 # patternEstimate=26219
              #      #          #           #       #           #
##########################################################################################
# 1  # 2      # -      # DFEProject      # columns=[?p]
              # -    # 1000     # 1000      # 1.00  # 0.14      #
##########################################################################################
# 2  # -      # -      # DFEDrain        # limit=10
              # -    # 1000     # 0         # 0.00  # 0.11      #
##########################################################################################
```

Example of `explain` output for a value expression function

The function is:

```
MATCH (a) RETURN DISTINCT labels(a)
```

In the `explain` output below, `DFEPipelineScan` (ID 0) scans for all the node labels. This corresponds to `MATCH (a)`.

`DFEChunkLocalSubquery` (ID 1) aggregates the label of `?a` for each `?a`. This corresponds to `labels(a)`. You can see that through `DFEApply` and `DFEReduce`.

`BindRelation` (ID 2) is used to rename the column generic `?__gen_labels0fa2` into `?labels(a)`.

`DFEDistinctRelation` (ID 4) retrieves only the distinct labels (multiple :airport nodes would give duplicate labels(a): ["airport"]). This corresponds to `DISTINCT labels(a)`.

To invoke `explain` for this query:

AWS CLI

```
aws neptunedata execute-open-cypher-explain-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "MATCH (a) RETURN DISTINCT labels(a)" \
  --explain-mode details
```

For more information, see execute-open-cypher-explain-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_explain_query(
    openCypherQuery='MATCH (a) RETURN DISTINCT labels(a)',
    explainMode='details'
)

print(response['results'].read().decode('utf-8'))
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=MATCH (a) RETURN DISTINCT labels(a)" \
  -d "explain=details"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=MATCH (a) RETURN DISTINCT labels(a)" \
  -d "explain=details"
```

The `explain` output:

```
Query:
MATCH (a) RETURN DISTINCT labels(a)

#######################################################################################
# ID # Out #1 # Out #2 # Name              # Arguments          # Mode               #
 Units In # Units Out # Ratio # Time (ms) #
#######################################################################################
# 0  # 1      # -      # SolutionInjection # solutions=[{}]     # -                  #
 0        # 1         # 0.00  # 0         #
#######################################################################################
# 1  # 2      # -      # DFESubquery       # subQuery=subQuery1 # -                  #
 0        # 5         # 0.00  # 81.00     #
#######################################################################################
# 2  # -      # -      # TermResolution    # vars=[?labels(a)]  # id2value_opencypher #
 5        # 5         # 1.00  # 1.00      #
#######################################################################################
```

```
subQuery1
#######################################################################################
# ID # Out #1 # Out #2 # Name                  # Arguments
                                                                          # Mode # Units
 In # Units Out # Ratio # Time (ms) #
#######################################################################################
# 0  # 1      # -      # DFEPipelineScan       # pattern=Node(?a) with property 'ALL'
 and label '?a_label1'                                                     # -    # 0
    # 3750      # 0.00  # 26.77     #
#    #        #        #                       # patternEstimate=3506
                                                                        #      #
  #           #       #           #
#######################################################################################
# 1  # 2      # -      # DFEChunkLocalSubQuery # subQuery=http://aws.amazon.com/
neptune/vocab/v01/dfe/past/graph#8b314f55-2cc7-456a-a48a-c76a0465cfab/graph_1 # -    #
 3750     # 3750      # 1.00  # 0.04      #
#######################################################################################
# 2  # 3      # -      # DFEBindRelation       # inputVars=[?a, ?__gen_labelsOfa2, ?
__gen_labelsOfa2]                                                          # -    # 3750
     # 3750      # 1.00  # 0.08      #
#    #        #        #                       # outputVars=[?a, ?__gen_labelsOfa2, ?
labels(a)]                                                                 #      #
    #           #       #           #
#######################################################################################
# 3  # 4      # -      # DFEProject            # columns=[?labels(a)]
                                                                        # -    # 3750
  # 3750      # 1.00  # 0.05      #
#######################################################################################
# 4  # 5      # -      # DFEDistinctRelation   # -
                                                                        # -    # 3750
  # 5         # 0.00  # 2.78      #
#######################################################################################
# 5  # -      # -      # DFEDrain              # -
                                                                        # -    # 5
  # 0         # 0.00  # 0.03      #
#######################################################################################



subQuery=http://aws.amazon.com/neptune/vocab/v01/dfe/past/graph#8b314f55-2cc7-456a-
a48a-c76a0465cfab/graph_1
#######################################################################################
# ID # Out #1 # Out #2 # Name                  # Arguments
                     # Mode     # Units In # Units Out # Ratio # Time (ms) #
#######################################################################################
```

```
# 0  # 1      # -      # DFESolutionInjection # outSchema=[?a]
                     # -        # 0        # 3750      # 0.00  # 0.02      #
##########################################################################################
# 1  # 2      # 3      # DFETee               # -
                     # -        # 3750     # 7500      # 2.00  # 0.02      #
##########################################################################################
# 2  # 4      # -      # DFEProject           # columns=[?a]
                     # -        # 3750     # 3750      # 1.00  # 0.04      #
##########################################################################################
# 3  # 17     # -      # DFEOptionalJoin      # -
                     # -        # 7500     # 3750      # 0.50  # 0.44      #
##########################################################################################
# 4  # 5      # -      # DFEDistinctRelation  # -
                     # -        # 3750     # 3750      # 1.00  # 2.23      #
##########################################################################################
# 5  # 6      # -      # DFEDistinctColumn    # column=?a
                     # -        # 3750     # 3750      # 1.00  # 1.50      #
#    #        #        #                      # ordered=false
                     #          #          #           #       #           #
##########################################################################################
# 6  # 7      # -      # DFEPipelineJoin      # pattern=Node(?a) with property 'ALL'
 and label '?a_label3' # -        # 3750     # 3750      # 1.00  # 10.58     #
#    #        #        #                      # patternEstimate=3506
                     #          #          #           #       #           #
##########################################################################################
# 7  # 8      # 9      # DFETee               # -
                     # -        # 3750     # 7500      # 2.00  # 0.02      #
##########################################################################################
# 8  # 10     # -      # DFEBindRelation      # inputVars=[?a_label3]
                     # -        # 3750     # 3750      # 1.00  # 0.04      #
#    #        #        #                      # outputVars=[?100]
                     #          #          #           #       #           #
##########################################################################################
# 9  # 11     # -      # DFEBindRelation      # inputVars=[?a, ?a_label3, ?100]
                     # -        # 7500     # 3750      # 0.50  # 0.07      #
#    #        #        #                      # outputVars=[?a, ?a_label3, ?100]
                     #          #          #           #       #           #
##########################################################################################
# 10 # 9      # -      # DFETermResolution    # column=?100
                     # id2value # 3750     # 3750      # 1.00  # 7.60      #
##########################################################################################
# 11 # 12     # -      # DFEBindRelation      # inputVars=[?a, ?a_label3, ?100]
                     # -        # 3750     # 3750      # 1.00  # 0.06      #
```

```
#    #        #        #                      # outputVars=[?a, ?100, ?a_label3]
                     #          #          #           #       #           #
##########################################################################################
# 12 # 13     # -      # DFEApply             # functor=nodeLabel(?a_label3)
                     # -        # 3750     # 3750      # 1.00  # 0.55      #
##########################################################################################
# 13 # 14     # -      # DFEProject           # columns=[?a, ?a_label3_alias4]
                     # -        # 3750     # 3750      # 1.00  # 0.05      #
##########################################################################################
# 14 # 15     # -      # DFEMergeChunks       # -
                     # -        # 3750     # 3750      # 1.00  # 0.02      #
##########################################################################################
# 15 # 16     # -      # DFEReduce            # functor=collect(?a_label3_alias4)
                     # -        # 3750     # 3750      # 1.00  # 6.37      #
#    #        #        #                      # segmentationKey=[?a]
                     #          #          #           #       #           #
##########################################################################################
# 16 # 3      # -      # DFEMergeChunks       # -
                     # -        # 3750     # 3750      # 1.00  # 0.03      #
##########################################################################################
# 17 # -      # -      # DFEDrain             # -
                     # -        # 3750     # 0         # 0.00  # 0.02      #
##########################################################################################
```

Example of `explain` output for a mathematical value expression function

In this example, `RETURN abs(-10)` performs a simple evaluation, taking the absolute value of a constant, `-10`.

`DFEChunkLocalSubQuery` (ID 1) performs a solution injection for the static value `-10`, which is stored in the variable, `?100`.

`DFEApply` (ID 2) is the operator that executes the absolute value function `abs()` on the static value stored in `?100` variable.

Here is the query and resulting `explain` output.

To invoke `explain` for this query:

AWS CLI

```
aws neptunedata execute-open-cypher-explain-query \
```

```
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "RETURN abs(-10)" \
  --explain-mode details
```

For more information, see execute-open-cypher-explain-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_explain_query(
    openCypherQuery='RETURN abs(-10)',
    explainMode='details'
)

print(response['results'].read().decode('utf-8'))
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=RETURN abs(-10)" \
  -d "explain=details"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=RETURN abs(-10)" \
  -d "explain=details"
```

The `explain` output:

```
Query:
RETURN abs(-10)

#######################################################################################
# ID # Out #1 # Out #2 # Name              # Arguments             # Mode
  # Units In # Units Out # Ratio # Time (ms) #
#######################################################################################
# 0  # 1      # -      # SolutionInjection # solutions=[{}]        # -
  # 0        # 1         # 0.00  # 0         #
#######################################################################################
# 1  # 2      # -      # DFESubquery       # subQuery=subQuery1    # -
  # 0        # 1         # 0.00  # 4.00      #
#######################################################################################
# 2  # -      # -      # TermResolution    # vars=[?_internalVar1] #
 id2value_opencypher # 1        # 1         # 1.00  # 1.00      #
#######################################################################################


subQuery1
#######################################################################################
# ID # Out #1 # Out #2 # Name                  # Arguments
                                                                    # Mode # Units
 In # Units Out # Ratio # Time (ms) #
#######################################################################################
# 0  # 1      # -      # DFESolutionInjection  # outSchema=[]
                                                                   # -    # 0
  # 1         # 0.00  # 0.01      #
#######################################################################################
# 1  # 2      # -      # DFEChunkLocalSubQuery # subQuery=http://aws.amazon.com/
neptune/vocab/v01/dfe/past/graph#c4cc6148-cce3-4561-93c0-deb91f257356/graph_1 # -    #
 1        # 1         # 1.00  # 0.03      #
#######################################################################################
```

```
# 2  # 3      # -      # DFEApply              # functor=abs(?100)
                                                                    # -    # 1
  # 1         # 1.00  # 0.26      #
##################################################################################
# 3  # 4      # -      # DFEBindRelation       # inputVars=[?_internalVar2, ?
_internalVar2]                                                        # -
 # 1        # 1         # 1.00  # 0.04      #
#    #        #        #                      # outputVars=[?_internalVar2, ?
_internalVar1]                                                       #
 #          #           #       #           #
##################################################################################
# 4  # 5      # -      # DFEProject            # columns=[?_internalVar1]
                                                                    # -    # 1
  # 1         # 1.00  # 0.06      #
##################################################################################
# 5  # -      # -      # DFEDrain              # -
                                                                    # -    # 1
  # 0         # 0.00  # 0.05      #
##################################################################################

subQuery=http://aws.amazon.com/neptune/vocab/v01/dfe/past/graph#c4cc6148-
cce3-4561-93c0-deb91f257356/graph_1
##################################################################################
# ID # Out #1 # Out #2 # Name                 # Arguments                          #
 Mode # Units In # Units Out # Ratio # Time (ms) #
##################################################################################
# 0  # 1      # -      # DFESolutionInjection # solutions=[?100 -> [-10^^<LONG>]]   # -
    # 0        # 1         # 0.00  # 0.01      #
#    #        #        #                      # outSchema=[?100]                   #
    #          #           #       #           #
##################################################################################
# 1  # 3      # -      # DFERelationalJoin    # joinVars=[]                        # -
    # 2        # 1         # 0.50  # 0.18      #
##################################################################################
# 2  # 1      # -      # DFESolutionInjection # outSchema=[]                      # -
    # 0        # 1         # 0.00  # 0.01      #
##################################################################################
# 3  # -      # -      # DFEDrain              # -                                  # -
    # 1        # 0         # 0.00  # 0.02      #
##################################################################################
```

Example of `explain` output for a variable-length path (VLP) query

This is an example of a more complex query plan for handling a variable-length path query. This example only shows part of the `explain` output, for clarity.

In `subQuery1`, `DFEPipelineScan` (ID 0) and `DFEChunkLocalSubQuery` (ID 1), which injects the `...graph_1` subquery, are responsible for scanning for a node with the `YPO` code.

In `subQuery1`, `DFEChunkLocalSubQuery` (ID 2), which injects the `...graph_2` subquery, is responsible for scanning for a node with the `LAX` code.

In `subQuery1`, `DFEChunkLocalSubQuery` (ID 3) injects the `...graph3` subquery, which contains `DFELoopSubQuery` (ID 17), which in turn injects the `...graph5` subquery. This operation is responsible for resolving the `-[*2]->` variable-length pattern in the query string between two nodes.

To invoke `explain` for this query:

AWS CLI

```
aws neptunedata execute-open-cypher-explain-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "MATCH p=(a {code: 'YPO'})-[*2]->(b{code: 'LAX'}) return p" \
  --explain-mode details
```

For more information, see execute-open-cypher-explain-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_explain_query(
    openCypherQuery="MATCH p=(a {code: 'YPO'})-[*2]->(b{code: 'LAX'}) return p",
    explainMode='details'
)
```

```
print(response['results'].read().decode('utf-8'))
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=MATCH p=(a {code: 'YPO'})-[*2]->(b{code: 'LAX'}) return p" \
  -d "explain=details"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=MATCH p=(a {code: 'YPO'})-[*2]->(b{code: 'LAX'}) return p" \
  -d "explain=details"
```

The `explain` output:

```
Query:
MATCH p=(a {code: 'YPO'})-[*2]->(b{code: 'LAX'}) return p

#######################################################################################
# ID # Out #1 # Out #2 # Name              # Arguments          # Mode               #
 Units In # Units Out # Ratio # Time (ms) #
#######################################################################################
# 0  # 1      # -      # SolutionInjection # solutions=[{}]     # -                  #
 0        # 1         # 0.00  # 0         #
#######################################################################################
# 1  # 2      # -      # DFESubquery       # subQuery=subQuery1 # -                  #
 0        # 0         # 0.00  # 84.00     #
#######################################################################################
```

```
# 2  # -      # -      # TermResolution    # vars=[?p]          # id2value_opencypher #
 0        # 0         # 0.00  # 0         #
##################################################################################################



subQuery1
##################################################################################################
# ID # Out #1 # Out #2 # Name                  # Arguments
                                                                                  # Mode # Units
 In # Units Out # Ratio # Time (ms) #
##################################################################################################
# 0  # 1      # -      # DFEPipelineScan       # pattern=Node(?a) with property 'code'
 as ?a_code7 and label 'ALL'                                                      # -    # 0
   # 1         # 0.00  # 0.68      #
#    #        #        #                       # inlineFilters=[(?a_code7 IN
 ["YPO"^^xsd:string])]                                                                  #
  #          #           #       #           #
#    #        #        #                       # patternEstimate=1
                                                                                 #      #
  #           #       #           #
##################################################################################################
# 1  # 2      # -      # DFEChunkLocalSubQuery # subQuery=http://aws.amazon.com/
neptune/vocab/v01/dfe/past/graph#cc05129f-d07e-4622-bbe3-9e99558eca46/graph_1 # -    #
 1        # 1         # 1.00  # 0.03      #
##################################################################################################
# 2  # 3      # -      # DFEChunkLocalSubQuery # subQuery=http://aws.amazon.com/
neptune/vocab/v01/dfe/past/graph#cc05129f-d07e-4622-bbe3-9e99558eca46/graph_2 # -    #
 1        # 1         # 1.00  # 0.02      #
##################################################################################################
# 3  # 4      # -      # DFEChunkLocalSubQuery # subQuery=http://aws.amazon.com/
neptune/vocab/v01/dfe/past/graph#cc05129f-d07e-4622-bbe3-9e99558eca46/graph_3 # -    #
 1        # 0         # 0.00  # 0.04      #
##################################################################################################
# 4  # 5      # -      # DFEBindRelation       # inputVars=[?__gen_path6, ?
anon_rel26, ?b_code8, ?b, ?a_code7, ?a, ?__gen_path6]                              # -
   # 0        # 0         # 0.00  # 0.10      #
#    #        #        #                       # outputVars=[?__gen_path6, ?
anon_rel26, ?b_code8, ?b, ?a_code7, ?a, ?p]                                            #
  #          #           #       #           #
##################################################################################################
# 5  # 6      # -      # DFEProject            # columns=[?p]
                                                                                 # -    # 0
  # 0         # 0.00  # 0.05      #
##################################################################################################
```

```
# 6  # -      # -      # DFEDrain              # -
                                                                    # -    # 0
  # 0         # 0.00  # 0.02      #
##################################################################################


subQuery=http://aws.amazon.com/neptune/vocab/v01/dfe/past/graph#cc05129f-d07e-4622-
bbe3-9e99558eca46/graph_1
##################################################################################
# ID # Out #1 # Out #2 # Name                 # Arguments
                    # Mode # Units In # Units Out # Ratio # Time (ms) #
##################################################################################
# 0  # 1      # -      # DFESolutionInjection # outSchema=[?a, ?a_code7]
                    # -    # 0        # 1         # 0.00  # 0.01      #
##################################################################################
# 1  # 2      # 3      # DFETee               # -
                    # -    # 1        # 2         # 2.00  # 0.01      #
##################################################################################
# 2  # 4      # -      # DFEDistinctColumn    # column=?a
                    # -    # 1        # 1         # 1.00  # 0.25      #
#    #        #        #                      # ordered=false
                    #      #          #           #       #           #
##################################################################################
# 3  # 5      # -      # DFEHashIndexBuild    # vars=[?a]
                    # -    # 1        # 1         # 1.00  # 0.05      #
##################################################################################
# 4  # 5      # -      # DFEPipelineJoin      # pattern=Node(?a) with property 'ALL'
 and label '?a_label1' # -    # 1        # 1         # 1.00  # 0.47      #
#    #        #        #                      # patternEstimate=3506
                    #      #          #           #       #           #
##################################################################################
# 5  # 6      # 7      # DFESync              # -
                    # -    # 2        # 2         # 1.00  # 0.04      #
##################################################################################
# 6  # 8      # -      # DFEForwardValue      # -
                    # -    # 1        # 1         # 1.00  # 0.01      #
##################################################################################
# 7  # 8      # -      # DFEForwardValue      # -
                    # -    # 1        # 1         # 1.00  # 0.01      #
##################################################################################
# 8  # 9      # -      # DFEHashIndexJoin     # -
                    # -    # 2        # 1         # 0.50  # 0.26      #
##################################################################################
```

```
# 9  # -      # -      # DFEDrain             # -
                    # -    # 1        # 0         # 0.00  # 0.02      #
##################################################################################



subQuery=http://aws.amazon.com/neptune/vocab/v01/dfe/past/graph#cc05129f-d07e-4622-
bbe3-9e99558eca46/graph_2
##################################################################################
# ID # Out #1 # Out #2 # Name                 # Arguments
                           # Mode # Units In # Units Out # Ratio # Time (ms) #
##################################################################################
# 0  # 1      # -      # DFEPipelineScan      # pattern=Node(?b) with property 'code'
 as ?b_code8 and label 'ALL' # -    # 0        # 1         # 0.00  # 0.38      #
#    #        #        #                      # inlineFilters=[(?b_code8 IN
 ["LAX"^^xsd:string])]                 #      #          #           #       #
  #
#    #        #        #                      # patternEstimate=1
                           #      #          #           #       #           #
##################################################################################
# 1  # 2      # -      # DFEMergeChunks       # -
                           # -    # 1        # 1         # 1.00  # 0.02      #
##################################################################################
# 2  # 4      # -      # DFERelationalJoin    # joinVars=[]
                           # -    # 2        # 1         # 0.50  # 0.19      #
##################################################################################
# 3  # 2      # -      # DFESolutionInjection # outSchema=[?a, ?a_code7]
                           # -    # 0        # 1         # 0.00  # 0         #
##################################################################################
# 4  # -      # -      # DFEDrain             # -
                           # -    # 1        # 0         # 0.00  # 0.01      #
##################################################################################



subQuery=http://aws.amazon.com/neptune/vocab/v01/dfe/past/graph#cc05129f-d07e-4622-
bbe3-9e99558eca46/graph_3
##################################################################################
# ID # Out #1 # Out #2 # Name                   # Arguments
                                                                     # Mode     #
 Units In # Units Out # Ratio # Time (ms) #
##################################################################################
...
# 17 # 18     # -      # DFELoopSubQuery       # subQuery=http://aws.amazon.com/
neptune/vocab/v01/dfe/past/graph#cc05129f-d07e-4622-bbe3-9e99558eca46/graph_5 # -
  # 1        # 2         # 2.00  # 0.31      #
```

```
...
```

Transactions in Neptune openCypher

The openCypher implementation in Amazon Neptune uses the transaction semantics defined by Neptune However, isolation levels provided by the Bolt driver have some specific implications for Bolt transaction semantics, as described in the sections below.

Read-only Bolt transaction queries

There are various ways that read-only queries can be processed, with different transaction models and isolation levels, as follows:

Implicit read-only transaction queries

Here is an example of a read-only implicit transaction:

```
public void executeReadImplicitTransaction()
{
  // end point
  final String END_POINT = "(End Point URL)";

  // read query
  final String READ_QUERY = "MATCH (n) RETURN n limit 10";

  // create the driver
  final Driver driver = GraphDatabase.driver(END_POINT, AuthTokens.none(),
          Config.builder().withEncryption()
                          .withTrustStrategy(TrustStrategy.trustSystemCertificates())
                          .build());

  // create the session config
  SessionConfig sessionConfig = SessionConfig.builder()
                                            .withFetchSize(1000)
                                            .withDefaultAccessMode(AccessMode.READ)
                                            .build();

  // run the query as access mode read
  driver.session(sessionConfig).readTransaction(new TransactionWork<String>()
    {
      final StringBuilder resultCollector = new StringBuilder();

      @Override
```

```
      public String execute(final Transaction tx)
      {
        // execute the query
        Result queryResult = tx.run(READ_QUERY);

        // Read the result
        for (Record record : queryResult.list())
        {
          for (String key : record.keys())
          {
            resultCollector.append(key)
                           .append(":")
                           .append(record.get(key).asNode().toString());
          }
        }
        return resultCollector.toString();
      }

    }
  );

  // close the driver.
  driver.close();
}
```

Because read-replicas only accept read-only queries, all queries against read-replicas execute as read-implicit transactions regardless of the access mode set in the session configuration. Neptune evaluates read-implicit transactions as read-only queries under SNAPSHOT isolation semantics.

In case of failure, read-implicit transactions are retried by default.

Autocommit read-only transaction queries

Here is an example of a read-only autocommit transaction:

```
public void executeAutoCommitTransaction()
{
  // end point
  final String END_POINT = "(End Point URL)";

  // read query
  final String READ_QUERY = "MATCH (n) RETURN n limit 10";
```

```
  // Create the session config.
  final SessionConfig sessionConfig = SessionConfig
    .builder()
    .withFetchSize(1000)
    .withDefaultAccessMode(AccessMode.READ)
    .build();

  // create the driver
  final Driver driver = GraphDatabase.driver(END_POINT, AuthTokens.none(),
    Config.builder()
          .withEncryption()
          .withTrustStrategy(TrustStrategy.trustSystemCertificates())
          .build());

  // result collector
  final StringBuilder resultCollector = new StringBuilder();

  // create a session
  final Session session = driver.session(sessionConfig);

  // run the query
  final Result queryResult = session.run(READ_QUERY);
  for (final Record record : queryResult.list())
  {
    for (String key : record.keys())
    {
      resultCollector.append(key)
                     .append(":")
                     .append(record.get(key).asNode().toString());
    }
  }

  // close the session
  session.close();

  // close the driver
  driver.close();
}
```

If the access mode is set to READ in the session configuration, Neptune evaluates autocommit transaction queries as read-only queries under SNAPSHOT isolation semantics. Note that read-replicas only accept read-only queries.

If you don't pass in a session configuration, autocommit queries are processed by default with mutation query isolation, so it is important to pass in a session configuration that explicitly sets the access mode to READ.

In case of failure, read-only autocommit queries are not re-tried.

Explicit read-only transaction queries

Here is an example of an explicit read-only transaction:

```
public void executeReadExplicitTransaction()
{
  // end point
  final String END_POINT = "(End Point URL)";

  // read query
  final String READ_QUERY = "MATCH (n) RETURN n limit 10";

  // Create the session config.
  final SessionConfig sessionConfig = SessionConfig
    .builder()
    .withFetchSize(1000)
    .withDefaultAccessMode(AccessMode.READ)
    .build();

  // create the driver
  final Driver driver = GraphDatabase.driver(END_POINT, AuthTokens.none(),
    Config.builder()
          .withEncryption()
          .withTrustStrategy(TrustStrategy.trustSystemCertificates())
          .build());

  // result collector
  final StringBuilder resultCollector = new StringBuilder();

  // create a session
  final Session session = driver.session(sessionConfig);

  // begin transaction
  final Transaction tx = session.beginTransaction();

  // run the query on transaction
  final List<Record> list = tx.run(READ_QUERY).list();
```

```
  // read the result
  for (final Record record : list)
  {
    for (String key : record.keys())
    {
      resultCollector
        .append(key)
        .append(":")
        .append(record.get(key).asNode().toString());
    }
  }

  // commit the transaction and for rollback we can use beginTransaction.rollback();
  tx.commit();

  // close the driver
  driver.close();
}
```

If the access mode is set to READ in the session configuration, Neptune evaluates explicit read-only transactions as read-only queries under SNAPSHOT isolation semantics. Note that read-replicas only accept read-only queries.

If you don't pass in a session configuration, explicit read-only transactions are processed by default with mutation query isolation, so it is important to pass in a session configuration that explicitly sets the access mode to READ.

In case of failure, read-only explicit queries are retried by default.

Mutation Bolt transaction queries

As with read-only queries, there are various ways that mutation queries can be processed, with different transaction models and isolation levels, as follows:

Implicit mutation transaction queries

Here is an example of an implicit mutation transaction:

```
public void executeWriteImplicitTransaction()
{
  // end point
  final String END_POINT = "(End Point URL)";
```

```
// create node with label as label and properties.
final String WRITE_QUERY = "CREATE (n:label {name : 'foo'})";

// Read the vertex created with label as label.
final String READ_QUERY = "MATCH (n:label) RETURN n";

// create the driver
final Driver driver = GraphDatabase.driver(END_POINT, AuthTokens.none(),
  Config.builder()
        .withEncryption()
        .withTrustStrategy(TrustStrategy.trustSystemCertificates())
        .build());

// create the session config
SessionConfig sessionConfig = SessionConfig
  .builder()
  .withFetchSize(1000)
  .withDefaultAccessMode(AccessMode.WRITE)
  .build();

final StringBuilder resultCollector = new StringBuilder();

// run the query as access mode write
driver.session(sessionConfig).writeTransaction(new TransactionWork<String>()
{
  @Override
  public String execute(final Transaction tx)
  {
    // execute the write query and consume the result.
    tx.run(WRITE_QUERY).consume();

    // read the vertex written in the same transaction
    final List<Record> list = tx.run(READ_QUERY).list();

    // read the result
    for (final Record record : list)
    {
      for (String key : record.keys())
      {
        resultCollector
          .append(key)
          .append(":")
          .append(record.get(key).asNode().toString());
      }
```

```
      }
      return resultCollector.toString();
    }
  }); // at the end, the transaction is automatically committed.

  // close the driver.
  driver.close();
}
```

Reads made as part of mutation queries are executed under `READ COMMITTED` isolation with the usual guarantees for Neptune mutation transactions.

Whether or not you specifically pass in a session configuration, the transaction is always treated as a write transaction.

For conflicts, see Conflict Resolution Using Lock-Wait Timeouts.

Autocommit mutation transaction queries

Mutation autocommit queries inherit the same behavior as mutation implicit transactions.

If you do not pass in a session configuration, the transaction is treated as a write transaction by default.

In case of failure, mutation autocommit queries are not automatically retried.

Explicit mutation transaction queries

Here is an example of an explicit mutation transaction:

```
public void executeWriteExplicitTransaction()
{
  // end point
  final String END_POINT = "(End Point URL)";

  // create node with label as label and properties.
  final String WRITE_QUERY = "CREATE (n:label {name : 'foo'})";

  // Read the vertex created with label as label.
  final String READ_QUERY = "MATCH (n:label) RETURN n";

  // create the driver
  final Driver driver = GraphDatabase.driver(END_POINT, AuthTokens.none(),
    Config.builder()
```

```
        .withEncryption()
        .withTrustStrategy(TrustStrategy.trustSystemCertificates())
        .build());

// create the session config
SessionConfig sessionConfig = SessionConfig
  .builder()
  .withFetchSize(1000)
  .withDefaultAccessMode(AccessMode.WRITE)
  .build();

final StringBuilder resultCollector = new StringBuilder();

final Session session = driver.session(sessionConfig);

// run the query as access mode write
final Transaction tx = driver.session(sessionConfig).beginTransaction();

// execute the write query and consume the result.
tx.run(WRITE_QUERY).consume();

// read the result from the previous write query in a same transaction.
final List<Record> list = tx.run(READ_QUERY).list();

// read the result
for (final Record record : list)
{
  for (String key : record.keys())
  {
    resultCollector
      .append(key)
      .append(":")
      .append(record.get(key).asNode().toString());
  }
}

// commit the transaction and for rollback we can use tx.rollback();
tx.commit();

// close the session
session.close();

// close the driver.
driver.close();
```

```
}
```

Explicit mutation queries inherit the same behavior as implicit mutation transactions.

If you do not pass in a session configuration, the transaction is treated as a write transaction by default.

For conflicts, see Conflict Resolution Using Lock-Wait Timeouts.

Streaming query results with the Bolt protocol

When you run an openCypher query that returns a large number of records, you don't have to load the entire result set into client memory at once. Neptune supports result streaming over the Bolt protocol, which lets the driver fetch records in batches and process them incrementally as they arrive. This can be advantageous if you want to process results as they are being returned from the server, but requires configuring `FetchSize` and using lazy iteration to avoid loading the full result set into memory.

This works through the Bolt v4.0 PULL mechanism. Instead of requesting all records in a single message, the driver sends a PULL request with a batch size. For example, "give me the next 50 records." Neptune sends back that number of records along with a flag indicating whether more are available. The driver then requests the next batch when the client is ready, and this continues until the result set is fully consumed.

You control the batch size using the `FetchSize` setting in the driver's session configuration. Combined with lazy iteration (processing records one at a time as they're returned), this bounds client-side memory usage regardless of how large the total result set is.

> **Important**
>
> `FetchSize` controls the flow of data between server and client. It does not affect how Neptune executes the query. Neptune begins executing the query and producing results as soon as it receives the RUN message. Results are buffered server-side until the client requests them with a PULL message. If the client does not consume results within the configured query timeout, Neptune terminates the query, discards buffered results, and releases server-side resources. There is no separate server-side buffer limit; buffering continues until the client pulls the results or the query timeout is reached.
> Streaming is most effective with non-aggregating queries that can produce results incrementally. Queries that include `ORDER BY`, aggregation functions (such as `count()`,

`collect()`, or `sum()`), or `DISTINCT` require Neptune to compute the full result set before returning any records. In these cases, `FetchSize` still limits client-side memory usage per batch, but the server must hold the entire result set in memory before streaming begins.

Note

The openCypher HTTPS endpoint delivers results using HTTP chunked transfer encoding, which streams data to the client as it is produced. However, the client cannot control the pace of delivery. There is no equivalent of `FetchSize` for the HTTPS endpoint. To control client-side memory consumption with large result sets, use a Bolt driver connection with `FetchSize` configured.

Configuring FetchSize and processing results in batches

Set `FetchSize` on the session or driver configuration to control how many records the driver requests from Neptune per PULL message. Note that `FetchSize` controls the driver-to-server flow, not application-level batching. To process results in application-level batches (for example, to perform intermediate work between groups of records), use a helper method that accumulates records from the driver's lazy iterator.

The following examples show both: configuring `FetchSize` on the session, and using a helper method to group records into application-level batches for processing.

Example Java

```
static void processBatches(Result result, int size,
        Consumer<List<Record>> handler) {
    var batch = new ArrayList<Record>();
    while (result.hasNext()) {
        batch.add(result.next());
        if (batch.size() >= size) {
            handler.accept(batch);
            batch = new ArrayList<>();
        }
    }
    if (!batch.isEmpty()) handler.accept(batch);
}
```

```
// Usage:
try (Session session = driver.session(SessionConfig.builder()
        .withFetchSize(50)
        .withDefaultAccessMode(AccessMode.READ)
        .build())) {

    Result result = session.run("MATCH (m:movie) RETURN m.title AS title");
    processBatches(result, 10, batch -> {
        for (var record : batch) {
            System.out.println(record.get("title").asString());
        }

        // Do other intermediary work here between batch calls
    });
    result.consume();
}
```

Example Python

```
async def batched(result, size):
    batch = []
    async for record in result:
        batch.append(record)
        if len(batch) >= size:
            yield batch
            batch = []
    if batch:
        yield batch

# Usage:
async with driver.session(fetch_size=50) as session:
    result = await session.run("MATCH (m:movie) RETURN m.title AS title")
    async for batch in batched(result, 10):
        for record in batch:
            print(record["title"])

        # Do other intermediary work here between batch calls
```

Example Go

```
func processBatches(ctx context.Context, result neo4j.ResultWithContext,
        size int, handler func([]*neo4j.Record)) error {
    var batch []*neo4j.Record
```

```
    for result.Next(ctx) {
        batch = append(batch, result.Record())
        if len(batch) >= size {
            handler(batch)
            batch = nil
        }
    }
    if len(batch) > 0 {
        handler(batch)
    }
    return result.Err()
}

// Usage:
session := driver.NewSession(ctx, neo4j.SessionConfig{
    AccessMode: neo4j.AccessModeRead,
    FetchSize:  50,
})
defer session.Close(ctx)

result, err := session.Run(ctx,
    "MATCH (m:movie) RETURN m.title AS title", nil)
if err != nil {
    log.Fatal(err)
}

if err := processBatches(ctx, result, 10, func(batch []*neo4j.Record) {
    for _, record := range batch {
        title, _ := record.Get("title")
        fmt.Println(title)
    }

    // Do other intermediary work here between batch calls
}); err != nil {
    log.Fatal(err)
}
```

Example.NET

```
static async IAsyncEnumerable<List<IRecord>> Batched(
        IResultCursor result, int size)
{
    var batch = new List<IRecord>();
```

```
    while (await result.FetchAsync())
    {
        batch.Add(result.Current);
        if (batch.Count >= size)
        {
            yield return batch;
            batch = new List<IRecord>();
        }
    }
    if (batch.Count > 0) yield return batch;
}

// Usage:
await using var session = driver.AsyncSession(o => o
    .WithFetchSize(50)
    .WithDefaultAccessMode(AccessMode.Read));

var result = await session.RunAsync("MATCH (m:movie) RETURN m.title AS title");
await foreach (var batch in Batched(result, 10))
{
    foreach (var record in batch)
    {
        Console.WriteLine(record["title"].As<string>());
    }

    // Do other intermediary work here between batch calls
}
```

Example Node.js

```
// Uses for-await-of on the Result's async iterator (Neo4j driver v5+).
// This is pull-based: the driver pauses fetching while the loop body
// executes, providing natural backpressure. This is preferred over
// result.subscribe(), which is push-based and continues delivering
// records during async processing.
async function* batched(result, size) {
    let batch = [];
    for await (const record of result) {
        batch.push(record);
        if (batch.length >= size) {
            yield batch;
            batch = [];
        }
```

```
    }
    if (batch.length > 0) yield batch;
}

// Usage:
const session = driver.session({
    defaultAccessMode: neo4j.session.READ,
    fetchSize: 50
});

try {
    const result = session.run('MATCH (m:movie) RETURN m.title AS title');
    for await (const batch of batched(result, 10)) {
        for (const record of batch) {
            console.log(record.get('title'));
        }

        // Do other intermediary work here between batch calls
    }
} finally {
    await session.close();
}
```

Eager vs. incremental consumption

The following methods block until the entire result set is collected into memory, preventing your application from acting on results as they arrive:

- **Java:** `result.list()`
- **Python:** `result.data()`, `list(result)`
- **Go:** `result.Collect(ctx)`, `neo4j.EagerResultTransformer`
- **.NET:** `result.ToListAsync()`
- **Node.js:** `const { records } = await result`

To process results incrementally, use the lazy iteration and batch helper patterns shown in the examples above.

openCypher query hints

> **Important**
>
> openCypher query hint is only available from engine release 1.3.2.0 and later.

In Amazon Neptune, you can use the `USING` clause to specify query hints for openCypher queries. These hints allow you to control optimization and evaluation strategies.

The syntax for query hints is:

```
USING {scope}:{hint} {value}
```

1. `{scope}` defines the scope in which the hint applies to: `Query` or `Clause`.

   A scope value of `Query` means that the query hint applies to the whole query (query-level).

   A scope value of `Clause` means that the query hint applies to the clause the hint precedes (clause-level).
2. `{hint}` is the name of the query hint being applied.
3. `{value}` is the argument for the `{hint}`.

The values can be case-insensitive.

For example, to enable the query plan cache for a query:

```
Using QUERY:PLANCACHE "enabled"
MATCH (a:Person {firstName: "Erin", lastName: $lastName})
 RETURN a
```

> **Note**
>
> Currently, the **Query** scope query hints **PLANCACHE**, **TIMEOUTMILLISECONDS**, and **assumeConsistentDataTypes** are supported. Supported query hints are listed below.

Topics

- openCypher query plan cache hint
- AssumeConsistentDataTypes hint
- openCypher query timeout hint

openCypher query plan cache hint

Query plan cache behavior can be overridden on a per-query (parameterized or not) basis by query-level query hint `QUERY:PLANCACHE`. It needs to be used with the `USING` clause. The query hint accepts `enabled` or `disabled` as a value. For more information on query plan cache, see Query plan cache in Amazon Neptune.

AWS CLI

Forcing plan to be cached or reused:

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "Using QUERY:PLANCACHE \"enabled\" MATCH(n) RETURN n LIMIT 1"
```

With parameters:

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "Using QUERY:PLANCACHE \"enabled\" RETURN \$arg" \
  --parameters '{"arg": 123}'
```

Forcing plan to be neither cached nor reused:

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "Using QUERY:PLANCACHE \"disabled\" MATCH(n) RETURN n LIMIT 1"
```

For more information, see execute-open-cypher-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
```

```
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

# Forcing plan to be cached or reused
response = client.execute_open_cypher_query(
    openCypherQuery='Using QUERY:PLANCACHE "enabled" MATCH(n) RETURN n LIMIT 1'
)

print(response['results'])
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

Forcing plan to be cached or reused:

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=Using QUERY:PLANCACHE \"enabled\" MATCH(n) RETURN n LIMIT 1"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

Forcing plan to be cached or reused:

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=Using QUERY:PLANCACHE \"enabled\" MATCH(n) RETURN n LIMIT 1"
```

With parameters:

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=Using QUERY:PLANCACHE \"enabled\" RETURN \$arg" \
  -d "parameters={\"arg\": 123}"
```

Forcing plan to be neither cached nor reused:

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=Using QUERY:PLANCACHE \"disabled\" MATCH(n) RETURN n LIMIT 1"
```

AssumeConsistentDataTypes hint

openCypher follows a paradigm where matches of numerical datatypes (e.g., int, byte, short, long, etc.) are carried out under type promotion semantics. For instance, when looking up all properties with an input value 10 with short type, under type promotion semantics, it would also match properties that have 10 as a long value. In some cases, type casting can induce overhead and result in query plans that are less efficient than they could be if no type casting was performed. In particular in cases where datatypes are used consistently in the data (e.g., if all person's ages are stored as long value) performing type promotions causes overhead without impacting the query result.

To allow optimization for cases when it is known that numeric property data values stored in the database are of consistent type, a query hint called `assumeConsistentDataTypes` (with value `true/false`, with the default being `false`) can be used. When this query hint is supplied with a value of `true`, the engine assumes the only property values are always long or double and will skip the type promotion semantics. Numerical values specified in the query are considered to be either long values (for non-floating point values) and double (for floating point values).

If the data is consistently using a single datatype (e.g. all ages are stored as `long`), then using the `assumeConsistentDataTypes` hint can optimize the query by skipping unnecessary equality checks for different numeric types. However, if the data has inconsistent datatypes for the same property, then using the hint may cause some results to be missed, as the query will only match the single datatype that the hint assumes.

```
# Database loaded with following openCypher CSV's

# File 1
:ID,age:Int
n1,20
n2,25

# File 2
:ID,age:Long
n3,25
```

```
# Example (no hint)
MATCH (n:Person)
WHERE n.age >= 25
RETURN n

# Result
n2
n3

Returns all person whose age is >= 25 and the values >= 25 can be with any of these
 datatypes
i.e. byte, short, int, long, double or float

--------------------------------------------------------------------------------

# Example (with hint present)
USING QUERY:assumeConsistentDataTypes "true"
MATCH (n:Person)
WHERE n.age >= 25
RETURN n

# Result
n3

Returns only "n3" and not "n2". The reason is that even though the numerical value
matches (25), the datatype is "int" and is considered a non-match.
```

The difference can also be validated via the explain.

Without the explain:

```
# Query
MATCH (n)
WHERE n.age = 20
RETURN n

# Explain Snippet
##########################################################################################
# ID # Out #1 # Out #2 # Name                    # Arguments

         # Mode # Units In # Units Out # Ratio # Time (ms) #
##########################################################################################
```

```
# 0  # 1      # -      # DFEPipelineScan (DFX)  # pattern=Node(?n) with property 'age'
 as ?n_age2 and label 'ALL'
          # -    # 0        # 1         # 0.00  # 0.10      #
#    #        #        #                        # inlineFilters=[(?n_age2 IN
 ["20"^^xsd:byte, "20"^^xsd:int, "20"^^xsd:long, "20"^^xsd:short, "20.0"^^xsd:double,
 "20.0"^^xsd:float])] #      #          #           #       #           #
#    #        #        #                        # patternEstimate=1

         #      #          #           #       #           #
##########################################################################################

# The inFilters field contains all numeric types
```

With the hint:

```
# Query
MATCH (n)
WHERE n.age = 20
RETURN n

# Explain Snippet
##########################################################################################
# ID # Out #1 # Out #2 # Name                    # Arguments
                           # Mode # Units In # Units Out # Ratio # Time (ms) #
##########################################################################################
# 0  # 1      # -      # DFEPipelineScan (DFX)  # pattern=Node(?n) with property 'age'
 as ?n_age2 and label 'ALL' # -    # 0        # 1         # 0.00  # 0.07      #
#    #        #        #                        # inlineFilters=[(?n_age2 IN
 ["20"^^xsd:long])]                   #      #          #           #       #
 #
#    #        #        #                        # patternEstimate=1
                           #      #          #           #       #           #
##########################################################################################

# The inFilters field only contains long datatype
```

openCypher query timeout hint

Query timeout behavior can be configured on a per-query basis by query-level query hint `QUERY:TIMEOUTMILLISECONDS`. It must be used with the `USING` clause. The query hint accepts non-negative long as a value.

AWS CLI

```
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-neptune-endpoint:port \
  --open-cypher-query "USING QUERY:TIMEOUTMILLISECONDS 100 MATCH(n) RETURN n LIMIT
 1"
```

For more information, see execute-open-cypher-query in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.execute_open_cypher_query(
    openCypherQuery='USING QUERY:TIMEOUTMILLISECONDS 100 MATCH(n) RETURN n LIMIT 1'
)

print(response['results'])
```

For AWS SDK examples in other languages, see AWS SDK.

awscurl

```
awscurl https://your-neptune-endpoint:port/openCypher \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -d "query=USING QUERY:TIMEOUTMILLISECONDS 100 MATCH(n) RETURN n LIMIT 1"
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl https://your-neptune-endpoint:port/openCypher \
  -d "query=USING QUERY:TIMEOUTMILLISECONDS 100 MATCH(n) RETURN n LIMIT 1"
```

Query timeout behavior will consider the minimum of cluster-level timeout and query-level timeout. Please see below examples to understand query timeout behavior. For more information on cluster-level query timeout, see neptune_query_timeout.

```
# Suppose `neptune_query_timeout` is 10000 ms and query-level timeout is set to 100 ms
# It will consider 100 ms as the final timeout

curl https://your-neptune-endpoint:port/openCypher \
  -d "query=USING QUERY:TIMEOUTMILLISECONDS 100 MATCH(n) RETURN n LIMIT 1"

# Suppose `neptune_query_timeout` is 100 ms and query-level timeout is set to 10000 ms
# It will still consider 100 ms as the final timeout

curl https://your-neptune-endpoint:port/openCypher \
  -d "query=USING QUERY:TIMEOUTMILLISECONDS 10000 MATCH(n) RETURN n LIMIT 1"
```

If a query exceeds the timeout, Neptune terminates it and returns a time-out error. Whether to retry a timed-out query depends on the nature of the failure and your workload. For guidance, see the section called “Exceptions and Retries”.

Neptune openCypher restrictions

The Amazon Neptune release of openCypher still does not support everything that is specified in the Cypher Query Language Reference, Version 9, as is detailed in openCypher specification compliance. Future releases are expected to address many of those limitations.

Neptune openCypher exceptions

When working with openCypher on Amazon Neptune, a variety of exceptions may occur. Below are common exceptions you may receive, either from the HTTPS endpoint or from the Bolt driver (all exceptions from the Bolt driver are reported as Server State Exceptions). Where the remedy is to retry the request, see the section called “Exceptions and Retries” for guidance on developing an effective retry strategy.

HTTP code	Error message	Retriable?	Remedy
400	*(syntax error, propagated directly from the openCypher parser)*	No	Correct query syntax, then retry.
500	`Operation terminated (out of memory)`	Yes	Rework the query to add additional filtering criteria to reduce required memory
500	Operation terminated (deadline exceeded)	Yes	Increase the query timeout in the DB cluster parameter group, or retry the request.
500	Operation terminated (cancelled by user)	Yes	Retry the request.
500	Database reset is in progress. Please retry the query after the cluster is available.	Yes	Retry when the reset is completed.
500	Operation failed due to conflicting concurren	Yes	Retry the request.

HTTP code	Error message	Retriable?	Remedy
	t operations (please retry). Transactions are currently rolling back.		
400	*`(operation name)`* operation /feature unsupported Exception	No	The specified operation is not supported.
400	openCypher update attempted on a read-only replica	No	Change the target end point to the writer end point.
400	Malformed QueryException (Neptune does not show the internal parser state)	No	Correct query syntax and retry.
400	Cannot delete node, because it still has relationships. To delete this node, you must first delete its relationships.	No	Instead of using `MATCH (n) DELETE n` use `MATCH(n) DETACH DELETE(n)`

HTTP code	Error message	Retriable?	Remedy	
400	Invalid operation : attempting to remove the last label of a node. A node must have at least one label.	No	Neptune requires all nodes to have at least one label, and if nodes are created without an explicit label, a default label `vertex` is assigned. Change the query and/ or application logic so as not to delete the last label. A singleton label of a node can be updated by setting a new label and then removing the old label.	
500	Max number of request have breached, Configure dQueueCap acity={} for connId = {}	Yes	Currently only 8,192 concurren t requests can be processed , regardless of the stack and protocol.	

HTTP code	Error message	Retriable?	Remedy
500	Max connection limit breached.	Yes	Only 1000 concurrent Bolt connections per instance are allowed (for HTTP there is no limit).
400	Expected a [one of: Node, Relationship or Path] and got a Literal	No	Check that you are passing the correct argument(s), correct query syntax, and retry.
400	Property value must be a simple literal. Or: Expected Map for Set propertie s but didn't find one.	No	A SET clause only accepts simple literals, not composite types.
400	Entity found passed for deletion is not found	No	Check that the entity you are trying to delete exists in the database.
400	User does not have access to the database.	No	Check the policy on the IAM role being used.

HTTP code	Error message	Retriable?	Remedy
400	There is no token passed as part of the request	No	A properly signed token must be passed as part of the query request on an IAM enabled cluster.
400	Error message is propagated.	No	Contact AWS Support with the Request Id.
500	Operation terminated (internal error)	Yes	Contact AWS Support with the Request Id.

openCypher extensions in Amazon Neptune

Amazon Neptune supports the openCypher specification reference version 9. Please refer to openCypher specification compliance in Amazon Neptune in Amazon Neptune for details. Additionally, Amazon Neptune supports the features listed here. Unless specific versions are mentioned, the features are available in Neptune Database and Neptune Analytics.

Query-time S3 data access

Available in Neptune Database 1.4.7.0 and up.

Neptune supports the `neptune.read()` function to read CSV or Parquet data from Amazon S3 directly within openCypher queries. Unlike the bulk loader which imports data before querying, `neptune.read()` accesses Amazon S3 data at query execution time.

For complete documentation, see neptune.read().

The Neptune-specific `join()` function

Available in Neptune Database and Neptune Analytics.

Neptune implements a `join()` function that is not present in the openCypher specification. It creates a string literal from a list of string literals and a string delimiter. It takes two arguments:

- The first argument is a list of string literals.
- The second argument is the delimiter string, which can consist of zero, one, or more than one characters.

Example:

```
join(["abc", "def", "ghi"], ", ")    // Returns "abc, def, ghi"
```

The Neptune-specific `removeKeyFromMap()` function

Available in Neptune Database and Neptune Analytics.

Neptune implements a `removeKeyFromMap()` function that is not present in the openCypher specification. It removes a specified key from a map and returns the resulting new map.

The function takes two arguments:

- The first argument is the map from which to remove the key.
- The second argument is the key to remove from the map.

The `removeKeyFromMap()` function is particularly useful in situations where you want to set values for a node or relationship by unwinding a list of maps. For example:

```
UNWIND [{`~id`: 'id1', name: 'john'}, {`~id`: 'id2', name: 'jim'}] as val
CREATE (n {`~id`: val.`~id`})
SET n = removeKeyFromMap(val, '~id')
```

Custom ID values for node and relationship properties

Available in Neptune Database 1.2.0.2 and up, and Neptune Analytics.

Starting in engine release 1.2.0.2, Neptune has extended the openCypher specification so that you can now specify the `id` values for nodes and relationships in `CREATE`, `MERGE`, and `MATCH` clauses. This lets you assign user-friendly strings instead of system-generated UUIDs to identify nodes and relationships.

In Neptune Analytics, custom id values are not available for edges.

> **Warning**
>
> This extension to the openCypher specification is backward incompatible, because `~id` is now considered a reserved property name. If you are already using `~id` as a property in your data and queries, you will need to migrate the existing property to a new property key and remove the old one. See What to do if you're currently using `~id` as a property.

Here is an example showing how to create nodes and relationships that have custom IDS:

```
CREATE (n {`~id`: 'fromNode', name: 'john'})
  -[:knows {`~id`: 'john-knows->jim', since: 2020}]
  ->(m {`~id`: 'toNode', name: 'jim'})
```

If you try to create a custom ID that is already in use, Neptune throws a `DuplicateDataException` error.

Here is an example of using a custom ID in a `MATCH` clause:

```
MATCH (n {`~id`: 'id1'})
RETURN n
```

Here is an example of using custom IDs in a `MERGE` clause:

```
MATCH (n {name: 'john'}), (m {name: 'jim'})
MERGE (n)-[r {`~id`: 'john->jim'}]->(m)
RETURN r
```

What to do if you're currently using `~id` as a property

With engine release 1.2.0.2, the `~id` key in openCypher clauses is now treated as `id` instead of as a property. This means that if you have a property named `~id`, accessing it becomes impossible.

If you're using an `~id` property, what you have to do before upgrading to engine release `1.2.0.2` or above is first to migrate the existing `~id` property to a new property key, and then remove the `~id` property. For example, the query below:

- Creates a new property named 'newId' for all nodes,
- copies over the value of the '~id' property into the 'newId' property,
- and removes the '~id' property from the data

```
MATCH (n)
WHERE exists(n.`~id`)
SET n.newId = n.`~id`
REMOVE n.`~id`
```

The same thing needs to be done for any relationships in the data that have an `~id` property.

You will also have to change any queries you're using that reference an `~id` property. For example, this query:

```
MATCH (n)
WHERE n.`~id` = 'some-value'
RETURN n
```

...would change to this:

```
MATCH (n)
WHERE n.newId = 'some-value'
RETURN n
```

CALL subquery support in Neptune

Available in Neptune Database 1.4.1.0 and up, and Neptune Analytics.

Amazon Neptune supports `CALL` subqueries. A `CALL` subquery is a part of the main query that runs in an isolated scope for each input to the `CALL` subquery.

For example, suppose a graph contains data about persons, their friends, and cities they lived in. We can retrieve the two largest cities where each friend of someone lived in by using a `CALL` subquery:

```
MATCH (person:Person)-[:knows]->(friend)
CALL {
```

```
  WITH friend
  MATCH (friend)-[:lived_in]->(city)
  RETURN city
  ORDER BY city.population DESC
  LIMIT 2
}
RETURN person, friend, city
```

In this example, the query part inside `CALL { ... }` is executed for each `friend` that is matched by the preceding MATCH clause. When the inner query is executed the `ORDER` and `LIMIT` clauses are local to the cities where a specific friend lived in, so we obtain (at most) two cities per friend.

All query clauses are available inside `CALL` subqueries. This includes nested `CALL` subqueries as well. Some restrictions for the first `WITH` clause and the emitted variables exist and are explained below.

Scope of variables inside CALL subquery

The variables from the clauses before the `CALL` subquery that are used inside it must be imported by the initial `WITH` clause. Unlike regular `WITH` clauses it can only contain a list of variables but doesn't allow aliasing and can't be used together with `DISTINCT`, `ORDER BY`, `WHERE`, `SKIP`, or `LIMIT`.

Variables returned from CALL subquery

The variables that are emitted from the `CALL` subquery are specified with the final `RETURN` clause. Note that the emitted variables cannot overlap with variables before the `CALL` subquery.

Limitations

As of now, updates inside of a `CALL` subquery are not supported.

Neptune openCypher functions

Available in Neptune Database 1.4.1.0 and up, and Neptune Analytics.

textIndexOf

```
textIndexOf(text :: STRING, lookup :: STRING, from = 0 :: INTEGER?, to =
-1 :: INTEGER?) :: (INTEGER?)
```

Returns the index of the first occurrence of `lookup` in the range of `text` starting at offset `from` (inclusive), through offset `to` (exclusive). If `to` is -1, the range continues to the end of `text`. Indexing is zero-based, and is expressed in Unicode scalar values (non-surrogate code points).

```
RETURN textIndexOf('Amazon Neptune', 'e')
{
  "results": [{
      "textIndexOf('Amazon Neptune', 'e')": 8
    }]
}
```

collToSet

`collToSet(values :: LIST OF ANY?) :: (LIST? OF ANY?)`

Returns a new list containing only the unique elements from the original list. The order of the original list is **maintained** (e.g `[1, 6, 5, 1, 5]` returns `[1, 6, 5]`).

```
RETURN collToSet([1, 6, 5, 1, 1, 5])
{
  "results": [{
      "collToSet([1, 6, 5, 1, 1, 5])": [1, 6, 5]
    }]
}
```

collSubtract

`collSubtract(first :: LIST OF ANY?, second :: LIST OF ANY?) :: (LIST? OF ANY?)`

Returns a new list containing all the unique elements of `first` excluding elements from `second`.

```
RETURN collSubtract([2, 5, 1, 0], [1, 5])
{
  "results": [{
      "collSubtract([2, 5, 1, 0], [1, 5])": [0, 2]
    }]
}
```

collIntersection

`collIntersection(first :: LIST? OF ANY?, second :: LIST? OF ANY?) :: (LIST? OF ANY?)`

Returns a new list containing all the unique elements of the intersection of `first` and `second`.

```
RETURN collIntersection([2, 5, 1, 0], [1, 5])
{
  "results": [{
      "collIntersection([2, 5, 1, 0], [1, 5])": [1, 5]
    }]
}
```

Sorting functions

The following sections define functions to sort collections. These functions take (in some cases optional) `config` map arguments, or a list of multiple such maps, that define the sort key and/or the sort direction:

```
{ key: STRING, order: STRING }
```

Here `key` is either a map or node property whose value is to be used for sorting. `order` is either "`asc`" or "`desc`" (case insensitive) to specify an ascending or descending sort, respectively. By default, sorting will be performed in ascending order.

collSort

`collSort(coll :: LIST OF ANY, config :: MAP?) :: (LIST? OF ANY?)`

Returns a new sorted list containing the elements from the `coll` input list.

```
RETURN collSort([5, 3, 1], {order: 'asc'})
{
  "results": [{
      "collSort([5, 3, 1])": [1, 3, 5]
    }]
}
```

collSortMaps

`collSortMaps(coll :: LIST OF MAP, config :: MAP) :: (LIST? OF ANY?)`

Returns a list of maps sorted by the value of the specified `key` property.

```
RETURN collSortMaps([{name: 'Alice', age: 25}, {name: 'Bob', age: 35}, {name:
 'Charlie', age: 18}], {key: 'age', order: 'desc'})
{
  "results": [{
      "x": [{
          "age": 35,
          "name": "Bob"
        }, {
          "age": 25,
          "name": "Alice"
        }, {
          "age": 18,
          "name": "Charlie"
        }]
    }]
}
```

collSortMulti

```
collSortMulti(coll :: LIST OF MAP?,
configs = [] :: LIST OF MAP,
limit = -1 :: INTEGER?,
skip = 0 :: INTEGER?) :: (LIST? OF ANY?)
```

Returns a list of maps sorted by the value of the specified `key` properties, optionally applying limit and skip.

```
RETURN collSortMulti([{name: 'Alice', age: 25}, {name: 'Bob', age: 35}, {name:
 'Charlie', age: 18}], [{key: 'age', order: 'desc'}, {key:'name'}]) as x
{
  "results": [{
      "x": [{
          "age": 35,
          "name": "Bob"
        }, {
          "age": 25,
          "name": "Alice"
        }, {
          "age": 18,
          "name": "Charlie"
        }]
    }]
```

```
}
```

collSortNodes

`collSortNodes(coll :: LIST OF NODE, config :: MAP) :: (LIST? OF NODE?)`

Returns a sorted version of the `coll` input list, sorting the node elements by the values of their respective `key` properties.

```
create (n:person {name: 'Alice', age: 23}), (m:person {name: 'Eve', age: 21}),
 (o:person {name:'Bob', age:25})
{"results":[]}

match (n:person) with collect(n) as people return collSortNodes(people, {key: 'name',
 order: 'desc'})
{
  "results": [{
      "collSortNodes(people, 'name')": [{
          "~id": "e599240a-8c23-4337-8aa8-f603c8fb5488",
          "~entityType": "node",
          "~labels": ["person"],
          "~properties": {
            "age": 21,
            "name": "Eve"
          }
        }, {
          "~id": "8a6ef785-59e3-4a0b-a0ff-389655a9c4e6",
          "~entityType": "node",
          "~labels": ["person"],
          "~properties": {
            "age": 25,
            "name": "Bob"
          }
        }, {
          "~id": "466bc826-f47f-452c-8a27-6b7bdf7ae9b4",
          "~entityType": "node",
          "~labels": ["person"],
          "~properties": {
            "age": 23,
            "name": "Alice"
          }
        }]
    }]
```

```
}

match (n:person) with collect(n) as people return collSortNodes(people, {key: 'age'})
{
  "results": [{
      "collSortNodes(people, '^age')": [{
          "~id": "e599240a-8c23-4337-8aa8-f603c8fb5488",
          "~entityType": "node",
          "~labels": ["person"],
          "~properties": {
            "age": 21,
            "name": "Eve"
          }
        }, {
          "~id": "466bc826-f47f-452c-8a27-6b7bdf7ae9b4",
          "~entityType": "node",
          "~labels": ["person"],
          "~properties": {
            "age": 23,
            "name": "Alice"
          }
        }, {
          "~id": "8a6ef785-59e3-4a0b-a0ff-389655a9c4e6",
          "~entityType": "node",
          "~labels": ["person"],
          "~properties": {
            "age": 25,
            "name": "Bob"
          }
        }]
    }]
}
```

Temporal functions

Temporal functions are available from Neptune version 1.4.5.0 and up.

day

```
day(temporal :: (datetime | date)) :: (LONG)
```

Returns the `day` of the month from a `datetime` or `date` value. For `datetime`: values are normalized to UTC based on input before extracting the day. For `date`: day is extracted based on the timezone.

The `datetime` input is available in both Neptune Database and Neptune Analytics:

```
RETURN day(datetime('2021-06-03T01:48:14Z'))
{
  "results": [{
      "day(datetime('2021-06-03T01:48:14Z'))": 3
    }]
}
```

Here, the `datetime` is normalized to UTC, so +08:00 shifts back to June 2.

```
RETURN day(datetime('2021-06-03T00:00:00+08:00'))
{
  "results": [{
      "day(datetime('2021-06-03T00:00:00+08:00'))": 2
    }]
}
```

The `date` input is available only in Neptune Analytics:

```
RETURN day(date('2021-06-03Z'))
{
  "results": [{
      "day(date('2021-06-03Z'))": 3
    }]
}
```

The `date` preserves timezone, keeping June 3.

```
RETURN day(date('2021-06-03+08:00'))
{
  "results": [{
      "day(date('2021-06-03+08:00'))": 3
    }]
}
```

month

`month(temporal :: (datetime | date)) :: (LONG)`

Returns the month from a `datetime` or `date` value (1-12). For `datetime`: values are normalized to UTC based on input before extracting the month. For `date`: month is extracted based on the timezone.

The `datetime` input is available in both Neptune Database and Neptune Analytics:

```
RETURN month(datetime('2021-06-03T01:48:14Z'))
{
  "results": [{
      "month(datetime('2021-06-03T01:48:14Z'))": 6
    }]
}
```

Here, the `datetime` is normalized to UTC, so +08:00 shifts back to May 31.

```
RETURN month(datetime('2021-06-01T00:00:00+08:00'))
{
  "results": [{
      "month(datetime('2021-06-01T00:00:00+08:00'))": 5
    }]
}
```

The `date` input is available only in Neptune Analytics:

```
RETURN month(date('2021-06-03Z'))
{
  "results": [{
      "month(date('2021-06-03Z'))": 6
    }]
}
```

The `date` preserves timezone, keeping June 1.

```
RETURN month(date('2021-06-01+08:00'))
{
  "results": [{
      "month(date('2021-06-01+08:00'))": 6
```

```
    }]
}
```

year

`year(temporal :: (datetime | date)) :: (LONG)`

Returns the year from a `datetime` or `date` value. For `datetime`: the values are normalized to UTC based on input before extracting the year. For `date`: the year is extracted based on the timezone.

The `datetime` input is available in both Neptune Database and Neptune Analytics:

```
RETURN year(datetime('2021-06-03T01:48:14Z'))
{
  "results": [{
      "year(datetime('2021-06-03T01:48:14Z'))": 2021
    }]
}
```

Here, the `datetime` is normalized to UTC, so +08:00 shifts back to December 31, 2020.

```
RETURN year(datetime('2021-01-01T00:00:00+08:00'))
{
  "results": [{
      "year(datetime('2021-01-01T00:00:00+08:00'))": 2020
    }]
}
```

The `date` input is available only in Neptune Analytics:

```
RETURN year(date('2021-06-03Z'))
{
  "results": [{
      "year(date('2021-06-03Z'))": 2021
    }]
}
```

The `date` preserves timezone, keeping June 2021.

```
RETURN year(date('2021-01-01+08:00'))
```

```
{
  "results": [{
      "year(date('2021-01-01+08:00'))": 2021
    }]
}
```

Neptune openCypher functions

Available in Neptune Database 1.4.6.0 and up, and Neptune Analytics.

reduce()

Reduce sequentially processes each list element by combining it with a running total or 'accumulator.' Starting with an initial value, it updates the accumulator after each operation and uses that updated value in the next iteration.

`for i in (0, ..., n) acc = acc X list[I], where X denotes any binary operator`

Once all elements have been processed, it returns the final accumulated result.

A typical reduce() structure would be - `reduce(accumulator = initial , variable IN list | expression)`

Type specifications:

`- initial: starting value for the accumulator :: (Long | FLOAT | STRING | LIST? OF (STRING, LONG, FLOAT)) - list: the input list :: LIST OF T where T matches initial type - variable :: represents each element in the input list - expression :: Only supports '+' and '*' operator - return :: Same type as initial`

Restrictions:

Currently, the `reduce()` expression only supports :

- Numeric Multiplication
- Numeric Addition
- String Concatenation
- List Concatenation

They are represented by the + or * operator. The expression should be a binary expression as specified below - `expression pattern: accumulator + any variable or accumulator * any variable`

Overflow handling:

Neptune detects numeric overflow during the `reduce()` evaluation and responds differently based on the data type:

```
LONG (signed#64-bit)
--------------------
• Valid range: -9#223#372#036#854#775#808#…#9#223#372#036#854#775#807
• If any intermediate or final value falls outside this range,
  Neptune aborts the query with long overflow error message.

FLOAT (IEEE-754 double)
-----------------------
• Largest finite value ≈#1.79#×#10^308
• Larger results overflow to INF
  Once `INF` is produced, it propagates through the remainder
  of the reduction.
```

Examples:

See the following examples for the reduce() function.

```
1. Long Addition:
RETURN reduce(sum = 0, n IN [1, 2, 3] | sum + n)
{
  "results": [{
      "reduce(sum = 0, n IN [1, 2, 3] | sum + n)": 6
    }]
}

2. String Concatenation:
RETURN reduce(str = "", x IN ["A", "B", "C"] | str + x)
{
  "results": [{
      "reduce(str = "", x IN ["A", "B", "C"] | str + x)": "ABC"
    }]
}
```

```
3. List Combination:
RETURN reduce(lst = [], x IN [1, 2, 3] | lst + x)
{
  "results": [{
      "reduce(lst = [], x IN [1, 2, 3] | lst + x)": [1, 2, 3]
    }]
}

4. Float Addition:
RETURN reduce(total = 0.0, x IN [1.5, 2.5, 3.5] | total + x)
{
  "results": [{
      "reduce(total = 0.0, x IN [1.5, 2.5, 3.5] | total + x)": 7.5
    }]
}

5. Long Multiplication:
RETURN reduce(product = 1, n IN [1, 2, 3] | product * n)
{
  "results": [{
      "reduce(product = 0, n IN [1, 2, 3] | product * n)": 6
    }]
}

6. Float Multiplication:
RETURN reduce(product = 1.0, n IN [1.5, 2.5, 3.5] | product * n)
{
  "results": [{
      "reduce(product = 1.0, n IN [1.5, 2.5, 3.5] | product * n)": 13.125
    }]
}

7. Long Overflow (Exception):
RETURN reduce(s = 9223372036854775807, x IN [2, 3] | s * x) AS result
{
"results": [{
    "reduce(s = 9223372036854775807, x IN [2, 3] | s * x) AS result": long overflow
    }]
}

8. Float Overflow:
RETURN reduce(s = 9.0e307, x IN [8.0e307, 1.0e307] | s + x) AS result
{
"results": [{
```

```
    "reduce(s = 9.0e307, x IN [8.0e307, 1.0e307] | s + x) AS result": INF
    }]
}
```

neptune.read()

Neptune supports a `CALL` procedure `neptune.read` to read data from Amazon S3 and then run an openCypher query (read, insert, update) using the data. The procedure yields each row in the file as a declared result variable row. It uses the IAM credentials of the caller to access the data in Amazon S3. See Managing permissions for neptune.read() to set up the permissions. The AWS region of the Amazon S3 bucket must be in the same region where instance is located. Currently, cross-region reads are not supported.

Syntax

```
CALL neptune.read(
  {
    source: "string",
    format: "parquet/csv",
    concurrency: 10
  }
)
YIELD row
...
```

Inputs

- **source** (required) - Amazon S3 URI to a **single** object. Amazon S3 prefix to multiple objects is not supported.
- **format** (required) - `parquet` and `csv` are supported.
  - More details on the supported Parquet format can be found in Supported Parquet column types.
  - For more information on the supported csv format, see Gremlin load data format.
- **concurrency** (optional) - Type: 0 or greater integer. Default: 0. Specifies the number of threads to be used for reading the file. If the value is 0, the maximum number of threads allowed by the resource will be used. For Parquet, it is recommended to be set to a number of row groups.

Outputs

The neptune.read returns:

- **row** - type:Map
  - Each row in the file, where the keys are the columns and the values are the data found in each column.
  - You can access each column's data like a property access (`row.col`).

Best practices for neptune.read()

Neptune S3 read operations can be memory-intensive. Please use instance types well-suited for production workloads as outlined in Choosing instance types for Amazon Neptune.

Memory usage and performance of `neptune.read()` requests are affected by a variety of factors like file size, number of columns, number of rows, and file format. Depending on structure, small files (e.g., CSV files 100MB or under, Parquet files 20MB or under) may work reliably on most production-suited instance types, whereas larger files may require substantial memory that smaller instance types cannot provide.

When testing this feature, it is recommended to start with small files and scale gradually to ensure your read workload can be accommodated by your instance size. If you notice `neptune.read()` requests leading to out-of-memory exceptions or instance restarts, consider splitting your files into smaller chunks, reducing file complexity, or upgrading to larger instance types.

Query examples using parquet

The following example query returns the number of rows in a given Parquet file:

```
CALL neptune.read(
  {
    source: "<s3 path>",
    format: "parquet"
  }
)
YIELD row
RETURN count(row)
```

You can run the query example using the `execute-open-cypher-query` operation in the AWS CLI by executing the following code:

```
aws neptunedata execute-open-cypher-query \
--open-cypher-query "CALL neptune.read({source: '<s3 path>', format: 'parquet'}) YIELD
 row RETURN count(row)" \
--endpoint-url https://my-cluster-name.cluster-abcdefgh1234.us-
east-1.neptune.amazonaws.com:8182
```

A query can be flexible in what it does with rows read from a Parquet file. For example, the following query creates a node with a field being set to data found in the Parquet file:

```
CALL neptune.read(
  {
    source: "<s3 path>",
    format: "parquet"
  }
)
YIELD row
CREATE (n {someField: row.someCol})
RETURN n
```

> **Warning**
>
> It is not considered good practice to use a large results-producing clause like `MATCH(n)` prior to a `CALL` clause. This would lead to a long-running query, due to cross product between incoming solutions from prior clauses and the rows read by neptune.read. It's recommended to start the query with `CALL` neptune.read.

Supported Parquet column types

Parquet Data Types:

- NULL
- BOOLEAN
- FLOAT
- DOUBLE
- STRING
- SIGNED INTEGER: UINT8, UINT16, UINT32, UINT64
- MAP: Only supports one-level. Does not support nested.

- LIST: Only supports one-level. Does not support nested.

Neptune-specific data types:

Unlike the property column headers of the CSV format, the property column headers of the Parquet format only need to have the property names, so there is no need to have the type names nor the cardinality.

There are however, some special column types in the Parquet format that require annotation in the metadata, including the Any type, Date type, dateTime type, and Geometry type. The following object is an example of the required metadata annotation for files containing columns of these special types:

```
"metadata": {
    "anyTypeColumns": ["UserCol1"],
    "dateTypeColumns": ["UserCol2"],
    "dateTimeTypeColumns": ["UserCol3"],
    "geometryTypeColumns": ["UserCol4"]
}
```

Below are details on the expected payload associated with these types:

- A column type Any is supported in the user columns. An Any type is a type "syntactic sugar" for all of the other types we support. It is extremely useful if a user column has multiple types in it. The payload of an Any type value is a list of json strings as follows: `{"value": "10", "type": "Int"};{"value": "1.0", "type": "Float"}`, which has a value field and a type field in each individual json string. The cardinality value of an Any column is set, meaning that the column can accept multiple values.
  - Neptune supports the following types in an Any type: Bool (or Boolean), Byte, Short, Int, Long, UnsignedByte, UnsignedShort, UnsignedInt, UnsignedLong, Float, Double, Date, dateTime, String, and Geometry.
  - Vector type is not supported in Any type.
  - Nested Any type is not supported. For example, `{"value": {"value": "10", "type": "Int"}, "type": "Any"}`.
- Columns of type Date and Datetime are supported in the user columns. The payload of these columns must be provided as strings following the XSD format or one of the formats below:
  - yyyy-MM-dd

- yyyy-MM-ddTHH:mm
- yyyy-MM-ddTHH:mm:ss
- yyyy-MM-ddTHH:mm:ssZ
- yyyy-MM-ddTHH:mm:ss.SSSZ
- yyyy-MM-ddTHH:mm:ss[+|-]hhmm
- yyyy-MM-ddTHH:mm:ss.SSS[+|-]hhmm

- A Geometry column type is supported in the user columns. The payload of these columns must only contain Geometry primitives of type Point, provided as strings in Well-known text (WKT) format. For example, POINT (30 10) would be a valid Geometry value.

Sample parquet output

Given a Parquet file like this:

```
<s3 path>

Parquet Type:
    int8     int16       int32             int64              float      double
 string
+--------+---------+-------------+----------------------+------------+------------
+----------+
|   Byte |   Short |       Int   |                Long  |     Float  |    Double  |
 String   |
|--------+---------+-------------+----------------------+------------+------------
+----------|
|   -128 |  -32768 | -2147483648 | -9223372036854775808 |    1.23456 |    1.23457 |
 first    |
|    127 |   32767 |  2147483647 |  9223372036854775807 |  nan       |  nan       |
 second   |
|      0 |       0 |           0 |                    0 | -inf       | -inf       |
 third    |
|      0 |       0 |           0 |                    0 |  inf       |  inf       |
 fourth   |
+--------+---------+-------------+----------------------+------------+------------
+----------+
```

Here is an example of the output returned by neptune.read using the following query:

```
aws neptunedata execute-open-cypher-query \
```

```
--open-cypher-query "CALL neptune.read({source: '<s3 path>', format: 'parquet'}) YIELD
 row RETURN row" \
--endpoint-url https://my-cluster-name.cluster-abcdefgh1234.us-
east-1.neptune.amazonaws.com:8182
```


```
{
 "results": [{
 "row": {
 "Float": 1.23456,
 "Byte": -128,
 "Int": -2147483648,
 "Long": -9223372036854775808,
 "String": "first",
 "Short": -32768,
 "Double": 1.2345678899999999
 }
 }, {
 "row": {
 "Float": "NaN",
 "Byte": 127,
 "Int": 2147483647,
 "Long": 9223372036854775807,
 "String": "second",
 "Short": 32767,
 "Double": "NaN"
 }
 }, {
 "row": {
 "Float": "-INF",
 "Byte": 0,
 "Int": 0,
 "Long": 0,
 "String": "third",
 "Short": 0,
 "Double": "-INF"
 }
 }, {
 "row": {
 "Float": "INF",
 "Byte": 0,
 "Int": 0,
 "Long": 0,
 "String": "fourth",
```

```
 "Short": 0,
 "Double": "INF"
 }
 }]
}
```

Currently, there is no way to set a node or edge label to a data field coming from a Parquet file. It is recommended that you partition the queries into multiple queries, one for each label/Type.

```
CALL neptune.read({source: '<s3 path>', format: 'parquet'})
 YIELD row
WHERE row.`~label` = 'airport'
CREATE (n:airport)

CALL neptune.read({source: '<s3 path>', format: 'parquet'})
YIELD row
WHERE row.`~label` = 'country'
CREATE (n:country)
```

Query examples using CSV

In this example, the query returns the number of rows in a given CSV file:

```
CALL neptune.read(
  {
    source: "<s3 path>",
    format: "csv"
  }
)
YIELD row
RETURN count(row)
```

You can run the query example using the execute-open-cypher-query operation in the AWS CLI by executing the following code:

```
aws neptunedata execute-open-cypher-query \
--open-cypher-query "CALL neptune.read({source: '<s3 path>', format: 'csv'}) YIELD row
 RETURN count(row)" \
--endpoint-url https://my-cluster-name.cluster-abcdefgh1234.us-
east-1.neptune.amazonaws.com:8182
```

A query can be flexible in what it does with rows read from a CSV file. For instance, the following query creates a node with a field set to data from a CSV file:

```
CALL neptune.read(
  {
    source: "<s3 path>",
    format: "csv"
  }
)
YIELD row
CREATE (n {someField: row.someCol})
RETURN n
```

Warning

It is not considered good practice use a large results-producing clause like MATCH(n) prior to a CALL clause. This would lead to a long-running query due to cross product between incoming solutions from prior clauses and the rows read by neptune.read. It is recommended to start the query with CALL neptune.read.

Property column headers

You can specify a column (`:`) for a property by using the following syntax. The type names are not case sensitive. If a colon appears within a property name, it must be escaped by preceding it with a backslash: `\:`.

```
propertyname:type
```

Note

- Space, comma, carriage return and newline characters are not allowed in the column headers, so property names cannot include these characters.
- You can specify a column for an array type by adding `[]` to the type:

```
propertyname:type[]
```

- Edge properties can only have a single value and will cause an error if an array type is specified or a second value is specified. The following example shows the column header for a property named age of type Int:

```
age:Int
```

Every row in the file would be required to have an integer in that position or be left empty. Arrays of strings are allowed, but strings in an array cannot include the semicolon (`;`) character unless it is escaped using a backslash (`\;`).

Supported CSV column types

- **BOOL (or BOOLEAN)** - Allowed values: true, false. Indicates a Boolean field. Any value other than true will be treated as false.
- **FLOAT** - Range: 32-bit IEEE 754 floating point including Infinity, INF, -Infinity, -INF and NaN (not-a-number).
- **DOUBLE** - Range: 64-bit IEEE 754 floating point including Infinity, INF, -Infinity, -INF and NaN (not-a-number).
- **STRING** -
  - Quotation marks are optional. Commas, newline, and carriage return characters are automatically escaped if they are included in a string surrounded by double quotation marks ("). Example: "Hello, World".
  - To include quotation marks in a quoted string, you can escape the quotation mark by using two in a row: Example: "Hello ""World""".
  - Arrays of strings are allowed, but strings in an array cannot include the semicolon (;) character unless it is escaped using a backslash (\;).
  - If you want to surround strings in an array with quotation marks, you must surround the whole array with one set of quotation marks. Example: "String one; String 2; String 3".
- **DATE, DATETIME** - The datetime values can be provided in either the XSD format, or one of the following formats:
  - yyyy-MM-dd
  - yyyy-MM-ddTHH:mm
  - yyyy-MM-ddTHH:mm:ss

  - yyyy-MM-ddTHH:mm:ssZ
  - yyyy-MM-ddTHH:mm:ss.SSSZ
  - yyyy-MM-ddTHH:mm:ss[+|-]hhmm
  - yyyy-MM-ddTHH:mm:ss.SSS[+|-]hhmm
- **SIGNED INTEGER** -
  - Byte: -128 to 127
  - Short: -32768 to 32767
  - Int: -2^31 to 2^31-1
  - Long: -2^63 to 2^63-1

Neptune-specific column types:

- A column type Any is supported in the user columns. An Any type is a type "syntactic sugar" for all of the other types we support. It is extremely useful if a user column has multiple types in it. The payload of an Any type value is a list of json strings as follows: `{"value": "10", "type": "Int"};{"value": "1.0", "type": "Float"}`, which has a value field and a type field in each individual json string. The column header of an Any type is propertyname:Any. The cardinality value of an Any column is set, meaning that the column can accept multiple values.
  - Neptune supports the following types in an Any type: Bool (or Boolean), Byte, Short, Int, Long, UnsignedByte, UnsignedShort, UnsignedInt, UnsignedLong, Float, Double, Date, dateTime, String, and Geometry.
  - Vector type is not supported in Any type.
  - Nested Any type is not supported. For example, `{"value": {"value": "10", "type": "Int"}, "type": "Any"}`.
- A Geometry column type is supported in the user columns. The payload of these columns must only contain Geometry primitives of type Point, provided as strings in Well-known text (WKT) format. For example, POINT (30 10) would be a valid Geometry value.

Sample CSV output

Given the following CSV file:

```
<s3 path>
```

```
colA:byte,colB:short,colC:int,colD:long,colE:float,colF:double,colG:string
-128,-32768,-2147483648,-9223372036854775808,1.23456,1.23457,first
127,32767,2147483647,9223372036854775807,nan,nan,second
0,0,0,0,-inf,-inf,third
0,0,0,0,inf,inf,fourth
```

This example shows the output returned by neptune.read using the following query:

```
aws neptunedata execute-open-cypher-query \
--open-cypher-query "CALL neptune.read({source: '<s3 path>', format: 'csv'}) YIELD row
 RETURN row" \
--endpoint-url https://my-cluster-name.cluster-abcdefgh1234.us-
east-1.neptune.amazonaws.com:8182
```

```
{
  "results": [{
      "row": {
        "colD": -9223372036854775808,
        "colC": -2147483648,
        "colE": 1.23456,
        "colB": -32768,
        "colF": 1.2345699999999999,
        "colG": "first",
        "colA": -128
      }
    }, {
      "row": {
        "colD": 9223372036854775807,
        "colC": 2147483647,
        "colE": "NaN",
        "colB": 32767,
        "colF": "NaN",
        "colG": "second",
        "colA": 127
      }
    }, {
      "row": {
        "colD": 0,
        "colC": 0,
        "colE": "-INF",
        "colB": 0,
        "colF": "-INF",
        "colG": "third",
```

```
        "colA": 0
      }
    }, {
      "row": {
        "colD": 0,
        "colC": 0,
        "colE": "INF",
        "colB": 0,
        "colF": "INF",
        "colG": "fourth",
        "colA": 0
      }
    }]
}
```

Currently, there is no way to set a node or edge label to a data field coming from a CSV file. It is recommended that you partition the queries into multiple queries, one for each label/type.

```
CALL neptune.read({source: '<s3 path>', format: 'csv'})
 YIELD row
WHERE row.`~label` = 'airport'
CREATE (n:airport)

CALL neptune.read({source: '<s3 path>', format: 'csv'})
YIELD row
WHERE row.`~label` = 'country'
CREATE (n:country)
```

Managing permissions for neptune.read()

Required IAM Policies

To execute openCypher queries that use `neptune.read()`, you must have the appropriate permissions to access data in your Neptune database. Read-only queries require the `ReadDataViaQuery` action. Queries that modify data require `WriteDataViaQuery` for insertions or `DeleteDataViaQuery` for deletions. The example below grants all three actions on the specified cluster.

Additionally, you need permissions to access the S3 bucket containing your data files. The NeptuneS3Access policy statement grants the required S3 permissions:

- **`s3:ListBucket`**: Required to verify bucket existence and list contents.

- **`s3:GetObject`**: Required to access the specified object so its content can be read for integration into openCypher queries.

If your S3 bucket uses server-side encryption with AWS KMS, you must also grant KMS permissions. The NeptuneS3KMSAccess policy statement allows Neptune to decrypt data and generate data keys when accessing encrypted S3 objects. The condition restricts KMS operations to requests originating from S3 and RDS services in your region.

- **`kms:Decrypt`**: Required to perform decryption of the encrypted object so its data can be read by Neptune.
- **`kms:GenerateDataKey`**: Also required by the S3 API used to retrieve objects to be read.

```
{
  "Sid": "NeptuneQueryAccess",
  "Effect": "Allow",
  "Action": [
      "neptune-db:ReadDataViaQuery",
      "neptune-db:WriteDataViaQuery",
      "neptune-db:DeleteDataViaQuery"
  ],
  "Resource": "arn:aws:neptune-db:<REGION>:<AWS_ACCOUNT_ID>:<CLUSTER_RESOURCE_ID>/*"
},
{
  "Sid": "NeptuneS3Access",
  "Effect": "Allow",
  "Action": [
      "s3:ListBucket",
      "s3:GetObject"
  ],
  "Resource": [
      "arn:aws:s3:::neptune-read-bucket",
      "arn:aws:s3:::neptune-read-bucket/*"
  ]
},
{
  "Sid": "NeptuneS3KMSAccess",
  "Effect": "Allow",
  "Action": [
      "kms:Decrypt",
      "kms:GenerateDataKey"
```

```
  ],
  "Resource": "arn:aws:kms:<REGION>:<AWS_ACCOUNT_ID>:key/<KEY_ID>",
  "Condition": {
      "StringEquals": {
        "kms:ViaService": [
            "s3.<REGION>.amazonaws.com",
            "rds.<REGION>.amazonaws.com"
        ]
      }
  }
}
```

Important prerequisites

These permissions and prerequisites ensure secure and reliable integration of S3 data into openCypher queries, while maintaining proper access controls and data protection measures.

- **IAM authentication**: This feature is only supported for Neptune clusters with IAM authentication enabled. See Securing your Amazon Neptune database for detailed instructions on how to create and connect to IAM authentication-enabled clusters.
- **VPC endpoint**:
  - A Gateway-type VPC endpoint for Amazon S3 is required to allow Neptune to communicate with Amazon S3.
  - To use custom AWS KMS encryption in the query, an Interface-type VPC endpoint for AWS KMS is required to allow Neptune to communicate with AWS KMS.
  - For detailed instructions for how to configure this endpoint, see Creating the Amazon S3 VPC Endpoint.

Spatial Data

Amazon Neptune now supports spatial queries, allowing you to store and analyze geometric data in your graph. While commonly used for geographic locations (like coordinates on a map), spatial features work with any two-dimensional data where position and proximity matter. Use this feature to answer questions like "Which stores are within 5 miles of this customer?", "Find all delivery routes that intersect with this service area," or "Which components in this floor plan overlap with the HVAC zone?" Neptune implements spatial support using industry-standard Spatial Types functions that work with points, polygons, and other geometric shapes. You can store spatial data

as properties on nodes and edges, then use spatial functions to calculate distances, check if points fall within boundaries, or find overlapping regions, all within your openCypher queries.

Common use cases:

- **Geographic applications**: Location-based recommendations, geofencing, route planning, and territory analysis
- **Facility and space management**: Floor plan layouts, equipment placement, and zone coverage
- **Network topology**: Physical infrastructure mapping, coverage areas, and service boundaries
- **Design and CAD**: Component positioning, collision detection, and spatial relationships in 2D designs
- **Game development**: Character positioning, collision detection, and area-of-effect calculations

The Spatial Types implementation in Amazon Neptune follows the ISO/IEC 13249-3:2016 directives, like other databases. The Spatial Functions are available in the openCypher query language.

Coordinate system

Neptune has one Spatial Reference Identifier (SRID) for an entire database. Homogeneity of the coordinate system reduce user errors in querying and improves the database performance. The first release (1.4.7.0) supports the Cartesian coordinate system, also referred as SRID 0.

The Neptune implementation of SRID 0 is compatible with longitude and latitude values. Use `ST_DistanceSpheroid` to calculate distances based on WGS84/SRID 4326.

The current implementation supports storing 3-dimensional coordinates. The Spatial Functions currently only support using the x- and y-axis (2-dimensional) coordinates. The z-axis coordinates are currently not supported by the available Spatial Functions.

Storing location data

Store location data on nodes and edges using the Geometry property type. Create Geometry values from Well-Known Text (WKT) format, a standard way to represent geographic shapes as text. For example, to store a point location:

```
CREATE (n:airport {code: 'ATL', location: ST_GeomFromText('POINT (-84.4281 33.6367)')})
```

When working with geographic coordinates, the first argument (x) represents longitude and the second argument (y) represents latitude. This follows the standard coordinate order used in spatial databases and the ISO 19125 standard.

> **Note**
>
> Neptune now supports a new data type called "Geometry". The Geometry property of a node or an edge can be created from a WKT string using the `ST_GeomFromText` function. Neptune will automatically store Points data in a specialized spatial index to improve the performance of the Spatial Types functions. For instance, `ST_Contains` used to find the points within a polygon is accelerated by the specialized spatial index.
> Wikipedia page for Well-Known Text representation of geometry

Loading spatial data in bulk

When bulk loading data, specify the Geometry type in your CSV header. Neptune will parse WKT strings and create the appropriate Geometry properties:

```
:ID,:LABEL,code:String,city:String,location:Geometry
21,airport,ATL,Atlanta,POINT (-84.42810059 33.63669968)
32,airport,ANC,Anchorage,POINT (-149.9960022 61.17440033)
43,airport,AUS,Austin,POINT (-97.66989899 30.19449997)
```

For complete CSV format details, see openCypher bulk load format.

Querying spatial data

The following query examples use the air-routes dataset to demonstrate how to use Spatial Functions in Neptune.

If your data has separate latitude and longitude properties instead of a Geometry property, you can convert them to points at query time. Find the 10 nearest airports to a given location:

```
MATCH (a:airport)
WITH a, ST_GeomFromText('POINT (' + a.lon + ' ' + a.lat + ')') AS airportLocation
WITH a, airportLocation, ST_Distance(ST_GeomFromText('POINT (-84.4281 33.6367)'),
 airportLocation) AS distance
WHERE distance IS NOT NULL
RETURN a.code, a.city, distance
ORDER BY distance ASC
```

```
LIMIT 10
```

If you already have locations stored as `ST_Point` then you can use those location values directly:

1. Set the property

```
MATCH (a:airport)
SET a.location = ST_GeomFromText('POINT (' + a.lon + ' ' + a.lat + ')')
```

2. Query using ST_Distance:

```
MATCH (a:airport)
WHERE a.location IS NOT NULL
WITH a, ST_Distance(ST_GeomFromText('POINT (-84.4281 33.6367)'), a.location) AS
 distance
RETURN a.code, a.city, distance
ORDER BY distance ASC
LIMIT 10
```

Using the Bolt driver

Most query methods return Geometry values as WKT strings, which are human-readable. If you're using the Bolt driver, Geometry values are returned in WKB (Well-Known Binary) format for efficiency. Convert WKB to a Geometry object in your application:

```
try (Session session = driver.session()) {
    Result result = session.run("MATCH (n:airport {code: 'ATL'}) RETURN n.location as
 geom");

    Record record = result.single();
    byte[] wkbBytes = record.get("geom").asByteArray();

    // Convert WKB to Geometry object using JTS library
    WKBReader wkbReader = new WKBReader();
    Geometry geom = wkbReader.read(wkbBytes);
}
```

Spatial Functions

The following spatial functions are available in Neptune openCypher for working with geometry data types:

- ST_Point
- ST_GeomFromText
- ST_AsText
- ST_GeometryType
- ST_Equals
- ST_Contains
- ST_Intersects
- ST_Distance
- ST_DistanceSpheroid
- ST_Envelope
- ST_Buffer

ST_Point

ST_Point returns a point from the input coordinate values.

Syntax

```
ST_Point(x, y, z)
```

Arguments

- x - A value of data type DOUBLE PRECISION that represents a first coordinate.
- y - A value of data type DOUBLE PRECISION that represents a second coordinate.
- z - (optional)

Coordinate order

When working with geographic coordinates, the first argument (x) represents **longitude** and the second argument (y) represents **latitude**. This follows the standard coordinate order used in spatial databases and the ISO 19125 standard.

```
// Correct: longitude first, latitude second
ST_Point(-84.4281, 33.6367)  // Atlanta airport

// Incorrect: latitude first, longitude second
```

```
ST_Point(33.6367, -84.4281)  // This will return NaN in distance calculations
```

Valid coordinate ranges

For geographic data, ensure coordinates fall within valid ranges:

- Longitude (x): -180 to 180
- Latitude (y): -90 to 90

Coordinates outside these ranges will return `NaN` (Not a Number) when used with distance calculation functions like `ST_DistanceSpheroid`.

Return type

GEOMETRY of subtype POINT

If x or y is null, then null is returned.

Examples

The following constructs a point geometry from the input coordinates.

```
RETURN ST_Point(5.0, 7.0);
POINT(5 7)
```

ST_GeomFromText

ST_GeomFromText constructs a geometry object from a well-known text (WKT) representation of an input geometry.

Syntax

```
ST_GeomFromText(wkt_string)
```

Arguments

- `wkt_string` - A value of data type STRING that is a WKT representation of a geometry.

Return type

GEOMETRY

If wkt_string is null, then null is returned.

If wkt_string is not valid, then a BadRequestException is returned.

Examples

```
RETURN ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))')
POLYGON((0 0,0 1,1 1,1 0,0 0))
```

ST_AsText

ST_AsText returns the well-known text (WKT) representation of an input geometry.

Syntax

```
ST_AsText(geo)
```

Arguments

- geo - A value of data type GEOMETRY, or an expression that evaluates to a GEOMETRY.

Return type

STRING

If geo is null, then null is returned.

If the input parameter is not a Geometry, then a BadRequestException is returned.

If the result is larger than a 64-KB STRING, then an error is returned.

Examples

```
RETURN ST_AsText(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))'))
POLYGON((0 0,0 1,1 1,1 0,0 0))
```

ST_GeometryType

ST_GeometryType returns the type of the geometry as a string.

Syntax

```
ST_GeometryType(geom)
```

Arguments

- geom - A value of data type GEOMETRY or an expression that evaluates to a GEOMETRY type.

Return type

STRING

If geom is null, then null is returned.

If the input parameter is not a Geometry, then a BadRequestException is returned.

Examples

```
RETURN ST_GeometryType(ST_GeomFromText('LINESTRING(77.29 29.07,77.42 29.26,77.27
 29.31,77.29 29.07)'));
ST_LineString
```

ST_Equals

ST_Equals returns true if the 2D projections of the input geometries are topologically equal. Geometries are considered topologically equal if they have equal point sets. In topologically equal geometries, the order of vertices may differ while maintaining this equality.

Syntax

```
ST_Equals(geom1, geom2)
```

Arguments

- geom1 - A value of data type GEOMETRY or an expression that evaluates to a GEOMETRY type.
- geom2 - A value of data type GEOMETRY or an expression that evaluates to a GEOMETRY type. This value is compared with geom1 to determine if it is equal to geom1.

Return type

BOOLEAN

If geom1 or geom2 is null, then null is returned.

If geom1 or geom2 are not Geometries, then a BadRequestException is returned.

Examples

```
RETURN ST_Equals(
    ST_GeomFromText('POLYGON ((0 2,1 1,0 -1,0 2))'),
    ST_GeomFromText('POLYGON((-1 3,2 1,0 -3,-1 3))'));
false
```

The following checks if the two linestrings are geometrically equal.

```
RETURN ST_Equals(
    ST_GeomFromText('LINESTRING (1 0, 10 0)'),
    ST_GeomFromText('LINESTRING(1 0,5 0,10 0)'));
true
```

ST_Contains

ST_Contains returns true if the 2D projection of the first input geometry contains the 2D projection of the second input geometry. Geometry A contains geometry B if every point in B is a point in A, and their interiors have nonempty intersection. ST_Contains(A, B) is equivalent to ST_Within(B, A).

Syntax

```
ST_Contains(geom1, geom2)
```

Arguments

- geom1 - A value of type GEOMETRY or an expression that evaluates to a GEOMETRY type.
- geom2 - A value of type GEOMETRY or an expression that evaluates to a GEOMETRY type. This value is compared with geom1 to determine if it is contained within geom1.

Return type

BOOLEAN

If geom1 or geom2 is null, then null is returned.

If the input parameter is not a Geometry, then a BadRequestException is returned.

Examples

```
RETURN ST_Contains(
    ST_GeomFromText('POLYGON((0 2,1 1,0 -1,0 2))'),
    ST_GeomFromText('POLYGON((-1 3,2 1,0 -3,-1 3))'));
false
```

ST_Intersects

ST_Intersects returns true if the 2D projections of the two input geometries have at least one point in common.

Syntax

```
ST_Intersects(geom1, geom2)
```

Arguments

- `geom1` - A value of data type GEOMETRY or an expression that evaluates to a GEOMETRY type.
- `geom2` - A value of data type GEOMETRY or an expression that evaluates to a GEOMETRY type.

Return type

BOOLEAN

If geom1 or geom2 is null, then null is returned.

If the input parameter is not a Geometry, then a BadRequestException is returned.

Examples

```
RETURN ST_Intersects(
    ST_GeomFromText('POLYGON((0 0,10 0,10 10,0 10,0 0),(2 2,2 5,5 5,5 2,2 2))'),
    ST_GeomFromText('MULTIPOINT((4 4),(6 6))'));
true
```

ST_Distance

For input geometries, ST_Distance returns the minimum Euclidean distance between the 2D projections of the two input geometry values.

Syntax

```
ST_Distance(geo1, geo2)
```

Arguments

- geo1 - A value of data type GEOMETRY, or an expression that evaluates to a GEOMETRY type.
- geo2 - A value of data type GEOMETRY, or an expression that evaluates to a GEOMETRY.

Return type

DOUBLE PRECISION in the same units as the input geometries.

If geo1 or geo2 is null, then null is returned.

If the input parameter is not a Geometry, then a BadRequestException is returned.

Examples

```
RETURN ST_Distance(
    ST_GeomFromText('POLYGON((0 2,1 1,0 -1,0 2))'),
    ST_GeomFromText('POLYGON((-1 -3,-2 -1,0 -3,-1 -3))'));
1.4142135623731
```

ST_DistanceSpheroid

Returns the minimum distance in meters between two lon/lat geometries. The spheroid is WGS84/SRID 4326.

Syntax

```
ST_DistanceSpheroid(geom1, geom2);
```

Arguments

- geom1 - A value of data type GEOMETRY or an expression that evaluates to a GEOMETRY type.
- geom2 - A value of data type GEOMETRY or an expression that evaluates to a GEOMETRY type.

Return type

FLOAT

If geom is null, then null is returned.

Examples

```
RETURN ST_DistanceSpheroid(
    ST_GeomFromText('POINT(-110 42)'),
    ST_GeomFromText('POINT(-118 38)'))
814278.77
```

ST_Envelope

ST_Envelope returns the minimum bounding box of the input geometry, as follows:

- If the input geometry is empty, the returned geometry will be POINT EMPTY.
- If the minimum bounding box of the input geometry degenerates to a point, the returned geometry is a point.
- If none of the preceding is true, the function returns a counter-clockwise-oriented polygon whose vertices are the corners of the minimum bounding box.

For all nonempty input, the function operates on the 2D projection of the input geometry.

Syntax

```
ST_Envelope(geom)
```

Arguments

- `geom` - A value of data type GEOMETRY or an expression that evaluates to a GEOMETRY type.

Return type

GEOMETRY

If geom is null, then null is returned.

Examples

```
RETURN ST_Envelope(ST_GeomFromText("POLYGON ((2 1, 4 3, 6 1, 5 5, 3 4, 2 1))"))
```

```
POLYGON ((2 1, 6 1, 6 5, 2 5, 2 1))
```

ST_Buffer

ST_Buffer returns 2D geometry that represents all points whose distance from the input geometry projected on the xy-Cartesian plane is less than or equal to the input distance.

Syntax

```
ST_Buffer(geom, distance, number_of_segments_per_quarter_circle)
```

Arguments

- `geom` - A value of data type GEOMETRY or an expression that evaluates to a GEOMETRY type.
- `distance` - A value of data type DOUBLE PRECISION that represents distance (or radius) of the buffer.
- `number_of_segments_per_quarter_circle` - A value of data type INTEGER (should be larger or equal to 0). This value determines the number of points to approximate a quarter circle around each vertex of the input geometry. Negative values default to zero. The default is 8.

Return type

GEOMETRY

The ST_Buffer function returns two-dimensional (2D) geometry in the xy-Cartesian plane.

Examples

```
RETURN ST_Buffer(ST_GeomFromText('LINESTRING (1 2,5 2,5 8)'), 2, 4);
POLYGON ((3 4, 3 8, 3.1522409349774265 8.76536686473018,
         3.585786437626905 9.414213562373096, 4.234633135269821 9.847759065022574,
         5 10, 5.765366864730179 9.847759065022574,
         6.414213562373095 9.414213562373096, 6.847759065022574 8.76536686473018,
         7 8, 7 2, 6.847759065022574 1.2346331352698203,
         6.414213562373095 0.5857864376269051, 5.765366864730179 0.1522409349774265,
         5 0, 1 0, 0.2346331352698193 0.152240934977427,
         -0.4142135623730954 0.5857864376269051,
         -0.8477590650225737 1.2346331352698208, -1 2.0000000000000004,
         -0.8477590650225735 2.7653668647301797,
         -0.4142135623730949 3.414213562373095,
```

```
          0.2346331352698206 3.8477590650225735, 1 4, 3 4))
```

The following returns the buffer of the input point geometry which approximates a circle. Because the command specifies 3 as the number of segments per quarter circle, the function uses three segments to approximate the quarter circle.

```
RETURN ST_Buffer(ST_GeomFromText('POINT (1 1)'), 1.0, 8));
POLYGON ((2 1, 1.9807852804032304 0.8049096779838718,
     1.9238795325112867 0.6173165676349102, 1.8314696123025453 0.4444297669803978,
     1.7071067811865475 0.2928932188134525, 1.5555702330196022 0.1685303876974548,
     1.3826834323650898 0.0761204674887133, 1.1950903220161284 0.0192147195967696,
     1 0, 0.8049096779838718 0.0192147195967696, 0.6173165676349103 0.0761204674887133,
    0.444429766980398 0.1685303876974545, 0.2928932188134525 0.2928932188134524,
     0.1685303876974546 0.4444297669803978, 0.0761204674887133 0.6173165676349102,
     0.0192147195967696 0.8049096779838714, 0 0.9999999999999999,
     0.0192147195967696 1.1950903220161284, 0.0761204674887132 1.3826834323650896,
     0.1685303876974545 1.555570233019602, 0.2928932188134523 1.7071067811865475,
     0.4444297669803978 1.8314696123025453, 0.6173165676349097 1.9238795325112865,
     0.8049096779838714 1.9807852804032304, 0.9999999999999998 2,
     1.1950903220161284 1.9807852804032304, 1.38268343236509 1.9238795325112865,
     1.5555702330196017 1.8314696123025453, 1.7071067811865475 1.7071067811865477,
     1.8314696123025453 1.5555702330196022, 1.9238795325112865 1.3826834323650905,
     1.9807852804032304 1.1950903220161286, 2 1))
```

Accessing the Neptune graph with SPARQL

SPARQL is a query language for the Resource Description Framework (RDF), which is a graph data format designed for the web. Amazon Neptune is compatible with SPARQL 1.1. This means that you can connect to a Neptune DB instance and query the graph using the query language described in the SPARQL 1.1 Query Language specification.

A query in SPARQL consists of a `SELECT` clause to specify the variables to return and a `WHERE` clause to specify which data to match in the graph. If you are unfamiliar with SPARQL queries, see Writing Simple Queries in the SPARQL 1.1 Query Language.

> **Important**
>
> To load data, `SPARQL UPDATE INSERT` may work well for a small dataset, but if you need to load a substantial amount of data from a file, see Using the Amazon Neptune bulk loader to ingest data.

For more information about the specifics of Neptune's SPARQL implementation, see SPARQL standards compliance.

Before you begin, you must have the following:

- A Neptune DB instance. For information about creating a Neptune DB instance, see Creating an Amazon Neptune cluster.
- An Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB instance.

Topics

- Using the RDF4J console to connect to a Neptune DB instance
- Using RDF4J Workbench to connect to a Neptune DB instance
- Using Java to connect to a Neptune DB instance
- SPARQL HTTP API
- SPARQL query hints
- SPARQL DESCRIBE behavior with respect to the default graph
- SPARQL query status API
- SPARQL query cancellation
- Using the SPARQL 1.1 Graph Store HTTP Protocol (GSP) in Amazon Neptune
- Analyzing Neptune query execution using SPARQL explain
- SPARQL federated queries in Neptune using the SERVICE extension

Using the RDF4J console to connect to a Neptune DB instance

The RDF4J Console allows you to experiment with Resource Description Framework (RDF) graphs and queries in a REPL (read-eval-print loop) environment.

You can add a remote graph database as a repository and query it from the RDF4J Console. This section walks you through the configuration of the RDF4J Console to connect remotely to a Neptune DB instance.

To connect to Neptune using the RDF4J Console

1. Download the RDF4J SDK from the Download page on the RDF4J website.

2. Unzip the RDF4J SDK zip file.
3. In a terminal, navigate to the RDF4J SDK directory, and then enter the following command to run the RDF4J Console:

```
bin/console.sh
```

You should see output similar to the following:

```
14:11:51.126 [main] DEBUG o.e.r.c.platform.PlatformFactory - os.name = linux
14:11:51.130 [main] DEBUG o.e.r.c.platform.PlatformFactory - Detected Posix
 platform
Connected to default data directory
RDF4J Console 3.6.1

3.6.1
Type 'help' for help.
>
```

You are now at the > prompt. This is the general prompt for the RDF4J Console. You use this prompt for setting up repositories and other operations. A repository has its own prompt for running queries.

4. At the > prompt, enter the following to create a SPARQL repository for your Neptune DB instance:

```
create sparql
```

5. The RDF4J Console prompts you for values for the variables required to connect to the SPARQL endpoint.

```
Please specify values for the following variables:
```

Specify the following values:

Variable Name	Value
SPARQL query endpoint	`https://your-neptune-endpoint :port/sparql`

SPARQL update endpoint	`https://your-neptune-endpoint :port/sparql`
Local repository ID [endpoint@localhost]	`neptune`
Repository title [SPARQL endpoint repository @localhost]	`Neptune DB instance`

For information about finding the address of your Neptune DB instance, see the Connecting to Amazon Neptune Endpoints section.

If the operation is successful, you see the following message:

```
Repository created
```

6. At the > prompt, enter the following to connect to the Neptune DB instance:

```
open neptune
```

If the operation is successful, you see the following message:

```
Opened repository 'neptune'
```

You are now at the `neptune>` prompt. At this prompt, you can run queries against the Neptune graph.

> **Note**
>
> Now that you have added the repository, the next time you run `bin/console.sh`, you can immediately run the `open neptune` command to connect to the Neptune DB instance.

7. At the `neptune>` prompt, enter the following to run a SPARQL query that returns up to 10 of the triples (subject-predicate-object) in the graph by using the `?s ?p ?o` query with a limit

of 10. To query for something else, replace the text after the `sparql` command with another SPARQL query.

```
sparql select ?s ?p ?o where {?s ?p ?o} limit 10
```

Using RDF4J Workbench to connect to a Neptune DB instance

This section walks you through connecting to an Amazon Neptune DB instance using RDF4J Workbench and RDF4J Server. RDF4J Server is required because it acts as a proxy between the Neptune SPARQL HTTP REST endpoint and RDF4J Workbench.

RDF4J Workbench provides an easy interface for experimenting with a graph, including loading local files. For information, see the Add section in the RDF4J documentation.

Prerequisites

Before you begin, do the following:

- Install Java 1.8 or later.
- Install RDF4J Server and RDF4J Workbench. For information, see Installing RDF4J Server and RDF4J Workbench.

To use RDF4J Workbench to connect to Neptune

1. In a web browser, navigate to the URL where the RDF4J Workbench web app is deployed. For example, if you are using Apache Tomcat, the URL is: https://*ec2_hostname*:8080/rdf4j-workbench/.
2. If you are asked to **Connect to RDF4J Server**, verify that **RDF4J Server** is installed, running, and that the server URL is correct. Then, proceed to the next step.
3. In the left pane, choose **New repository**.

   In **New repository**:

   - In the **Type** drop-down list, choose **SPARQL endpoint proxy**.
   - For **ID**, type **neptune**.
   - For **Title**, type **Neptune DB instance**.

Choose **Next**.

4. In **New repository**:

   - For **SPARQL query endpoint URL**, type `https://your-neptune-endpoint:port/sparql`.
   - For **SPARQL update endpoint URL**, type `https://your-neptune-endpoint:port/sparql`.

   For information about finding the address of your Neptune DB instance, see the Connecting to Amazon Neptune Endpoints section.

   Choose **Create**.

5. The **neptune** repository now appears in the list of repositories. It might take a few minutes before you can use the new repository.
6. In the **Id** column of the table, choose the **neptune** link.
7. In the left pane, choose **Query**.

> **Note**
>
> If the menu items under **Explore** are disabled, you might need to reconnect to the RDF4J Server and choose the **neptune** repository again.
> You can do this by using the **[change]** links in the upper-right corner.

8. In the query field, type the following SPARQL query, and then choose **Execute**.

```
select ?s ?p ?o where {?s ?p ?o} limit 10
```

The preceding example returns up to 10 of the triples (subject-predicate-object) in the graph by using the `?s ?p ?o` query with a limit of 10.

Using Java to connect to a Neptune DB instance

This section walks you through the running of a complete Java sample that connects to an Amazon Neptune DB instance and performs a SPARQL query.

Follow these instructions from an Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB instance.

To connect to Neptune using Java

1. Install Apache Maven on your EC2 instance. If using Amazon Linux 2023 (preferred), use:

   ```
   sudo dnf update -y
   sudo dnf install maven -y
   ```

   If using Amazon Linux 2, download the latest binary from https://maven.apache.org/download.cgi:

   ```
   sudo yum remove maven -y
   wget https://dlcdn.apache.org/maven/maven-3/ <version>/binaries/apache-maven-
   <version>-bin.tar.gz
   sudo tar -xzf apache-maven-<version>-bin.tar.gz -C /opt/
   sudo ln -sf /opt/apache-maven-<version> /opt/maven
   echo 'export MAVEN_HOME=/opt/maven' >> ~/.bashrc
   echo 'export PATH=$MAVEN_HOME/bin:$PATH' >> ~/.bashrc
   source ~/.bashrc
   ```

2. This example was tested with Java 8 only. Enter the following to install Java 8 on your EC2 instance:

   ```
   sudo yum install java-1.8.0-devel
   ```

3. Enter the following to set Java 8 as the default runtime on your EC2 instance:

   ```
   sudo /usr/sbin/alternatives --config java
   ```

   When prompted, enter the number for Java 8.

4. Enter the following to set Java 8 as the default compiler on your EC2 instance:

   ```
   sudo /usr/sbin/alternatives --config javac
   ```

When prompted, enter the number for Java 8.

5. In a new directory, create a `pom.xml` file, and then open it in a text editor.
6. Copy the following into the `pom.xml` file and save it (you can usually adjust the version numbers to the latest stable version):

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.amazonaws</groupId>
  <artifactId>RDFExample</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>RDFExample</name>
  <url>https://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>org.eclipse.rdf4j</groupId>
      <artifactId>rdf4j-runtime</artifactId>
      <version>3.6</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>exec-maven-plugin</artifactId>
          <version>1.2.1</version>
          <configuration>
            <mainClass>com.amazonaws.App</mainClass>
          </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
```

```
  </build>
</project>
```

> **Note**
>
> If you are modifying an existing Maven project, the required dependency is highlighted in the preceding code.

7. To create subdirectories for the example source code (`src/main/java/com/amazonaws/`), enter the following at the command line:

```
mkdir -p src/main/java/com/amazonaws/
```

8. In the `src/main/java/com/amazonaws/` directory, create a file named `App.java`, and then open it in a text editor.
9. Copy the following into the `App.java` file. Replace *`your-neptune-endpoint`* with the address of your Neptune DB instance.

> **Note**
>
> For information about finding the hostname of your Neptune DB instance, see the Connecting to Amazon Neptune Endpoints section.

```
package com.amazonaws;

import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.http.HTTPRepository;
import org.eclipse.rdf4j.repository.sparql.SPARQLRepository;

import java.util.List;
import org.eclipse.rdf4j.RDF4JException;
import org.eclipse.rdf4j.repository.RepositoryConnection;
import org.eclipse.rdf4j.query.TupleQuery;
import org.eclipse.rdf4j.query.TupleQueryResult;
import org.eclipse.rdf4j.query.BindingSet;
import org.eclipse.rdf4j.query.QueryLanguage;
import org.eclipse.rdf4j.model.Value;

public class App
```

```
{
    public static void main( String[] args )
    {
        String sparqlEndpoint = "https://your-neptune-endpoint:port/sparql";
        Repository repo = new SPARQLRepository(sparqlEndpoint);
        repo.initialize();

        try (RepositoryConnection conn = repo.getConnection()) {
           String queryString = "SELECT ?s ?p ?o WHERE { ?s ?p ?o } limit 10";

           TupleQuery tupleQuery = conn.prepareTupleQuery(QueryLanguage.SPARQL,
 queryString);

           try (TupleQueryResult result = tupleQuery.evaluate()) {
              while (result.hasNext()) {  // iterate over the result
                   BindingSet bindingSet = result.next();

                   Value s = bindingSet.getValue("s");
                   Value p = bindingSet.getValue("p");
                   Value o = bindingSet.getValue("o");

                   System.out.print(s);
                   System.out.print("\t");
                   System.out.print(p);
                   System.out.print("\t");
                   System.out.println(o);
              }
           }
        }
    }
}
```

10. Use the following Maven command to compile and run the sample:

```
mvn compile exec:java
```

The preceding example returns up to 10 of the triples (subject-predicate-object) in the graph by using the `?s ?p ?o` query with a limit of 10. To query for something else, replace the query with another SPARQL query.

The iteration of the results in the example prints the value of each variable returned. The `Value` object is converted to a `String` and then printed. If you change the `SELECT` part of the query, you must modify the code.

SPARQL HTTP API

SPARQL HTTP requests are accepted at the following endpoint: `https://your-neptune-endpoint:port/sparql`

For more information about connecting to Amazon Neptune with SPARQL, see Accessing the Neptune graph with SPARQL.

For more information about the SPARQL protocol and query language, see the SPARQL 1.1 Protocol and the SPARQL 1.1 Query Language specification.

The following topics provide information about SPARQL RDF serialization formats and how to use the SPARQL HTTP API with Neptune.

Contents

- Using the HTTP REST endpoint to connect to a Neptune DB instance
- Optional HTTP trailing headers for multi-part SPARQL responses
- RDF media types used by SPARQL in Neptune
  - RDF serialization formats used by Neptune SPARQL
  - SPARQL result serialization formats used by Neptune SPARQL
  - Media-Types that Neptune can use to import RDF data
  - Media-Types that Neptune can use to export query results
- Using SPARQL UPDATE LOAD to import data into Neptune
- Using SPARQL UPDATE UNLOAD to delete data from Neptune

Using the HTTP REST endpoint to connect to a Neptune DB instance

> **Note**
>
> Neptune does not currently support HTTP/2 for REST API requests. Clients must use HTTP/1.1 when connecting to endpoints.

The following instructions walk you through connecting to the SPARQL endpoint using the **curl** command, connecting through HTTPS, and using HTTP syntax. Follow these instructions from an Amazon EC2 instance in the same virtual private cloud (VPC) as your Neptune DB instance.

The HTTP endpoint for SPARQL queries to a Neptune DB instance is: `https://your-neptune-endpoint:port/sparql`.

> **Note**
>
> For information about finding the hostname of your Neptune DB instance, see the Connecting to Amazon Neptune Endpoints section.

Amazon Neptune provides an HTTP endpoint for SPARQL queries. The REST interface is compatible with SPARQL version 1.1.

QUERY Using HTTP POST

The following example uses **curl** to submit a SPARQL **QUERY** through HTTP **POST**.

```
curl -X POST --data-binary 'query=select ?s ?p ?o where {?s ?p ?o} limit 10'
 https://your-neptune-endpoint:port/sparql
```

The preceding example returns up to 10 of the triples (subject-predicate-object) in the graph by using the `?s ?p ?o` query with a limit of 10. To query for something else, replace it with another SPARQL query.

> **Note**
>
> The default MIME media type of a response is `application/sparql-results+json` for `SELECT` and `ASK` queries.
> The default MIME type of a response is `application/n-quads` for `CONSTRUCT` and `DESCRIBE` queries.
> For a list of the media types used by Neptune for serialization, see RDF serialization formats used by Neptune SPARQL.

UPDATE Using HTTP POST

The following example uses **curl** to submit a SPARQL **UPDATE** through HTTP **POST**.

```
curl -X POST --data-binary 'update=INSERT DATA { <https://test.com/s> <https://
test.com/p> <https://test.com/o> . }' https://your-neptune-endpoint:port/sparql
```

The preceding example inserts the following triple into the SPARQL default graph: `<https://test.com/s> <https://test.com/p> <https://test.com/o>`

Optional HTTP trailing headers for multi-part SPARQL responses

The HTTP response to SPARQL queries and updates is often returned in more than one part or chunk. It can be hard to diagnose a failure that occurs after a query or update begins sending these chunks, especially since the first one arrives with an HTTP status code of `200`.

Unless you explicitly request trailing headers, Neptune only reports such a failure by appending an error message to the message body, which is usually corrupted.

To make detection and diagnosis of this kind of problem easier, you can include a transfer-encoding (TE) trailers header (`te: trailers`) in your request (see, for example, the MDN page about TE request headers). Doing this will cause Neptune to include two new header fields within the trailing headers of the response chunks:

- `X-Neptune-Status` – contains the response code followed by a short name. For instance, in case of success the trailing header would be: `X-Neptune-Status: 200 OK`. In the case of failure, the response code would be an Neptune engine error code such as `X-Neptune-Status: 500 TimeLimitExceededException`.
- `X-Neptune-Detail` – is empty for successful requests. In the case of errors, it contains the JSON error message. Because only ASCII characters are allowed in HTTP header values, the JSON string is URL encoded. The error message is also still appended to the response message body.

RDF media types used by SPARQL in Neptune

Resource Description Framework (RDF) data can be serialized in many different ways, most of which SPARQL can consume or output:

RDF serialization formats used by Neptune SPARQL

- **RDF/XML** – XML serialization of RDF, defined in RDF 1.1 XML Syntax. Media type: `application/rdf+xml`. Typical file extension: `.rdf`.

- **N-Triples** – A line-based, plain-text format for encoding an RDF graph, defined in RDF 1.1 N-Triples. Media type: `application/n-triples`, `text/turtle`, or `text/plain`. Typical file extension: `.nt`.
- **N-Quads** – A line-based, plain-text format for encoding an RDF graph, defined in RDF 1.1 N-Quads. It is an extension of N-Triples. Media type: `application/n-quads`, or `text/x-nquads` when encoded with 7-bit US-ASCII. Typical file extension: `.nq`.
- **Turtle** – A textual syntax for RDF defined in RDF 1.1 Turtle that allows an RDF graph to be completely written in a compact and natural text form, with abbreviations for common usage patterns and datatypes. Turtle provides levels of compatibility with the N-Triples format as well as SPARQL's triple pattern syntax. Media type: `text/turtle`Typical file extension: `.ttl`.
- **TriG** – A textual syntax for RDF defined in RDF 1.1 TriG that allows an RDF graph to be completely written in a compact and natural text form, with abbreviations for common usage patterns and datatypes. TriG is an extension of the Turtle format. Media type: `application/trig`. Typical file extension: `.trig`.
- **N3 (Notation3)** – An assertion and logic language defined in Notation3 (N3): A readable RDF syntax. N3 extends the RDF data model by adding formulae (literals which are graphs themselves), variables, logical implication, and functional predicates, and provides a textual syntax alternative to RDF/XML. Media type: `text/n3`. Typical file extension: `.n3`.
- **JSON-LD** – A data serialization and messaging format defined in JSON-LD 1.0.Media type: `application/ld+json`. Typical file extension: `.jsonld`.
- **TriX** – A serialization of RDF in XML, defined in TriX: RDF Triples in XML. Media type: `application/trix`. Typical file extension: `.trix`.
- **SPARQL JSON Results** – A serialization of RDF using the SPARQL 1.1 Query Results JSON Format. Media type: `application/sparql-results+json`. Typical file extension: `.srj`.
- **RDF4J Binary Format** – A binary format for encoding RDF data, documented in RDF4J Binary RDF Format. Media type: `application/x-binary-rdf`.

SPARQL result serialization formats used by Neptune SPARQL

- **SPARQL XML Results** – An XML format for the variable binding and boolean results formats provided by the SPARQL query language, defined in SPARQL Query Results XML Format (Second Edition). Media type: `application/sparql-results+xml`. Typical file extension: `.srx`.
- **SPARQL CSV and TSV Results** – The use of comma-separated values and tab-separated values to express SPARQL query results from `SELECT` queries, defined in SPARQL 1.1 Query

Results CSV and TSV Formats. Media type: `text/csv` for comma-separated values, and `text/tab-separated-values` for tab-separated values. Typical file extensions: `.csv` for comma-separated values, and `.tsv` for tab-separated values.

- **Binary Results Table** – A binary format for encoding the output of SPARQL queries. Media type: `application/x-binary-rdf-results-table`.
- **SPARQL JSON Results** – A serialization of RDF using the SPARQL 1.1 Query Results JSON Format. Media type: `application/sparql-results+json`.

Media-Types that Neptune can use to import RDF data

Media-types supported by the Neptune bulk-loader

- N-Triples
- N-Quads
- RDF/XML
- Turtle

Media-types that SPARQL UPDATE LOAD can import

- N-Triples
- N-Quads
- RDF/XML
- Turtle
- TriG
- N3
- JSON-LD

Media-Types that Neptune can use to export query results

To specify the output format for a SPARQL query response, send an "`Accept: media-type`" header with the query request. For example:

```
curl -H "Accept: application/nquads" ...
```

RDF media-types that SPARQL SELECT can output from Neptune

- SPARQL JSON Results (This is the default)
- SPARQL XML Results
- **Binary Results Table** (media type: `application/x-binary-rdf-results-table`)
- Comma-Separated Values (CSV)
- Tab-Separated Values (TSV)

RDF media-types that SPARQL ASK can output from Neptune

- SPARQL JSON Results (This is the default)
- SPARQL XML Results
- **Boolean** (media type: `text/boolean`, meaning "true" or "false")

RDF media-types that SPARQL CONSTRUCT can output from Neptune

- N-Quads (This is the default)
- RDF/XML
- JSON-LD
- N-Triples
- Turtle
- N3
- TriX
- TriG
- SPARQL JSON Results
- RDF4J Binary RDF Format

RDF media-types that SPARQL DESCRIBE can output from Neptune

- N-Quads (This is the default)
- RDF/XML
- JSON-LD
- N-Triples

- Turtle
- N3
- TriX
- TriG
- SPARQL JSON Results
- RDF4J Binary RDF Format

Using SPARQL UPDATE LOAD to import data into Neptune

The syntax of the SPARQL UPDATE LOAD command is specified in the SPARQL 1.1 Update recommendation:

```
LOAD SILENT (URL of data to be loaded) INTO GRAPH (named graph into which to load the
 data)
```

- **`SILENT`** – (*Optional*) Causes the operation to return success even if there was an error during processing.

  This can be useful when a single transaction contains multiple statements like "`LOAD ...; LOAD ...; UNLOAD ...; LOAD ...;`" and you want the transaction to complete even if some of the remote data could not be processed.
- *`URL of data to be loaded`* – (*Required*) Specifies a remote data file containing data to be loaded into a graph.

  The remote file must have one of the following extensions:

  - `.nt` for NTriples.
  - `.nq` for NQuads.
  - `.trig` for Trig.
  - `.rdf` for RDF/XML.
  - `.ttl` for Turtle.
  - `.n3` for N3.
  - `.jsonld` for JSON-LD.
- **`INTO GRAPH`***`(named graph into which to load the data)`* – (*Optional*) Specifies the graph into which the data should be loaded.

Neptune associates every triple with a named graph. You can specify the default named graph using the fallback named-graph URI, `http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph`, like this:

```
INTO GRAPH <http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph>
```

> **Note**
>
> When you need to load a lot of data, we recommend that you use the Neptune bulk loader rather than UPDATE LOAD. For more information about the bulk loader, see Using the Amazon Neptune bulk loader to ingest data.

You can use `SPARQL UPDATE LOAD` to load data directly from Amazon S3, or from files obtained from a self-hosted web server. The resources to be loaded must reside in the same region as the Neptune server, and the endpoint for the resources must be allowed in the VPC. For information about creating an Amazon S3 endpoint, see Creating an Amazon S3 VPC Endpoint.

All `SPARQL UPDATE LOAD` URIs must start with `https://`. This includes Amazon S3 URLs.

In contrast to the Neptune bulk loader, a call to `SPARQL UPDATE LOAD` is fully transactional.

Loading files directly from Amazon S3 into Neptune using SPARQL UPDATE LOAD

Because Neptune does not allow you to pass an IAM role to Amazon S3 when using SPARQL UPDATE LOAD, either the Amazon S3 bucket in question must be public or you must use a pre-signed Amazon S3 URL in the LOAD query.

To generate a pre-signed URL for an Amazon S3 file, you can use an AWS CLI command like this:

```
aws s3 presign --expires-in (number of seconds) s3://(bucket name)/(path to file of
 data to load)
```

Then you can use the resulting pre-signed URL in your `LOAD` command:

```
curl https://(a Neptune endpoint URL):8182/sparql \
  --data-urlencode 'update=load (pre-signed URL of the remote Amazon S3 file of data to
 be loaded) \
```

```
                                into graph (named graph)'
```

For more information, see Authenticating Requests: Using Query Parameters. The Boto3 documentation shows how to use a Python script to generate a presigned URL.

Also, the content type of the files to be loaded must be set correctly.

1. Set the content type of files when you upload them into Amazon S3 by using the `-metadata` parameter, like this:

   ```
   aws s3 cp test.nt s3://bucket-name/my-plain-text-input/test.nt --metadata Content-
   Type=text/plain
   aws s3 cp test.rdf s3://bucket-name/my-rdf-input/test.rdf --metadata Content-
   Type=application/rdf+xml
   ```

2. Confirm that the media-type information is actually present. Run:

   ```
   curl -v bucket-name/folder-name
   ```

   The output of this command should show the media-type information that you set when uploading the files.

3. Then you can use the `SPARQL UPDATE LOAD` command to import these files into Neptune:

   ```
   curl https://your-neptune-endpoint:port/sparql \
     -d "update=LOAD <https://s3.amazonaws.com/bucket-name/my-rdf-input/test.rdf>"
   ```

The steps above work only for a public Amazon S3 bucket, or for a bucket that you access using a pre-signed Amazon S3 URL in the LOAD query.

You can also set up a web proxy server to load from a private Amazon S3 bucket, as shown below:

Using a web server to load files into Neptune with SPARQL UPDATE LOAD

1. Install a web server on a machine running within the VPC that is hosting Neptune and the files to be loaded. For example, using Amazon Linux, you might install Apache as follows:

   ```
   sudo yum install httpd mod_ssl
   sudo /usr/sbin/apachectl start
   ```

2. Define the MIME type(s) of the RDF file-content that you are going to load. SPARQL uses the `Content-type` header sent by the web server to determine the input format of the content, so you must define the relevant MIME types for the web Server.

   For example, suppose you use the following file extensions to identify file formats:

   - `.nt` for NTriples.
   - `.nq` for NQuads.
   - `.trig` for Trig.
   - `.rdf` for RDF/XML.
   - `.ttl` for Turtle.
   - `.n3` for N3.
   - `.jsonld` for JSON-LD.

   If you are using Apache 2 as the web server, you would edit the file `/etc/mime.types` and add the following types:

   ```
   text/plain nt
   application/n-quads nq
   application/trig trig
   application/rdf+xml rdf
   application/x-turtle ttl
   text/rdf+n3 n3
   application/ld+json jsonld
   ```

3. Confirm that the MIME-type mapping works. Once you have your web server up and running and hosting RDF files in the format(s) of your choice, you can test the configuration by sending a request to the web server from your local host.

   For instance, you might send a request such as this:

   ```
   curl -v http://localhost:80/test.rdf
   ```

   Then, in the detailed output from `curl`, you should see a line such as:

   ```
   Content-Type: application/rdf+xml
   ```

This shows that the content-type mapping was defined successfully.

4. You are now ready to load data using the SPARQL UPDATE command:

```
curl https://your-neptune-endpoint:port/sparql \
    -d "update=LOAD <http://web_server_private_ip:80/test.rdf>"
```

> **Note**
>
> Using `SPARQL UPDATE LOAD` can trigger a timeout on the web server when the source file being loaded is large. Neptune processes the file data as it is streamed in, and for a big file that can take longer than the timeout configured on the server. This in turn may cause the server to close the connection, which can result in the following error message when Neptune encounters an unexpected EOF in the stream:
>
> ```
> {
>  "detailedMessage":"Invalid syntax in the specified file",
>  "code":"InvalidParameterException"
> }
> ```
>
> If you receive this message and don't believe your source file contains invalid syntax, try increasing the timeout settings on the web server. You can also diagnose the problem by enabling debug logs on the server and looking for timeouts.

Using SPARQL UPDATE UNLOAD to delete data from Neptune

Neptune also provides a custom SPARQL operation, `UNLOAD`, for removing data that is specified in a remote source. `UNLOAD` can be regarded as a counterpart to the `LOAD` operation. Its syntax is:

```
UNLOAD SILENT (URL of the remote data to be unloaded) FROM GRAPH (named graph from
 which to remove the data)
```

- **`SILENT`** – (*Optional*) Causes the operation to return success even if there was an error when processing the data.

This can be useful when a single transaction contains multiple statements like `"LOAD ...; LOAD ...; UNLOAD ...; LOAD ...;"` and you want the transaction to complete even if some of the remote data could not be processed.

- *`URL of the remote data to be unloaded`* – (*Required*) Specifies a remote data file containing data to be unloaded from a graph.

  The remote file must have one of the following extensions (these are the same formats that UPDATE-LOAD supports):

  - `.nt` for NTriples.
  - `.nq` for NQuads.
  - `.trig` for Trig.
  - `.rdf` for RDF/XML.
  - `.ttl` for Turtle.
  - `.n3` for N3.
  - `.jsonld` for JSON-LD.

  All the data that this file contains will be removed from your DB cluster by the `UNLOAD` operation.

  Any Amazon S3 authentication must be included in the URL for the data to unload. You can pre-sign an Amazon S3 file and then use the resulting URL to access it securely. For example:

```
aws s3 presign --expires-in (number of seconds) s3://(bucket name)/(path to file of
 data to unload)
```

  Then:

```
curl https://(a Neptune endpoint URL):8182/sparql \
  --data-urlencode 'update=unload (pre-signed URL of the remote Amazon S3 data to be
 unloaded) \
                            from graph (named graph)'
```

  For more information, see Authenticating Requests: Using Query Parameters.

- **`FROM GRAPH`** *`(named graph from which to remove the data)`* – (*Optional*) Specifies the named graph from which the remote data should be unloaded.

Neptune associates every triple with a named graph. You can specify the default named graph using the fallback named-graph URI, `http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph`, like this:

```
FROM GRAPH <http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph>
```

In the same way that `LOAD` corresponds to `INSERT DATA { (inline data) }`, `UNLOAD` corresponds to `DELETE DATA { (inline data) }`. Like `DELETE DATA`, `UNLOAD` does not work on data that contains blank nodes.

For example, if a local web server serves a file named `data.nt` that contains the following 2 triples:

```
<http://example.org/resource#a> <http://example.org/resource#p> <http://example.org/
resource#b> .
<http://example.org/resource#a> <http://example.org/resource#p> <http://example.org/
resource#c> .
```

The following `UNLOAD` command would delete those two triples from the named graph, `<http://example.org/graph1>`:

```
UNLOAD <http://localhost:80/data.nt> FROM GRAPH <http://example.org/graph1>
```

This would have the same effect as using the following `DELETE DATA` command:

```
DELETE DATA {
  GRAPH <http://example.org/graph1> {
    <http://example.org/resource#a> <http://example.org/resource#p> <http://
example.org/resource#b> .
    <http://example.org/resource#a> <http://example.org/resource#p> <http://
example.org/resource#c> .
  }
}
```

Exceptions thrown by the `UNLOAD` command

- **`InvalidParameterException`** – There were blank nodes in the data. *HTTP status*: 400 Bad Request.

Message: `Blank nodes are not allowed for UNLOAD`

- **`InvalidParameterException`** – There was broken syntax in the data. *HTTP status*: 400 Bad Request.

  Message: `Invalid syntax in the specified file.`

- **`UnloadUrlAccessDeniedException`** – Access was denied. *HTTP status*: 400 Bad Request.

  Message: `Update failure: Endpoint (Neptune endpoint) reported access denied error. Please verify access.`

- **`BadRequestException`** – The remote data cannot be retrieved. *HTTP status*: 400 Bad Request.

  Message: *(depends on the HTTP response).*

SPARQL query hints

You can use query hints to specify optimization and evaluation strategies for a particular SPARQL query in Amazon Neptune.

Query hints are expressed using additional triple patterns that are embedded in the SPARQL query with the following parts:

```
scope hint value
```

- *scope* – Determines the part of the query that the query hint applies to, such as a certain group in the query or the full query.
- *hint* – Identifies the type of the hint to apply.
- *value* – Determines the behavior of the system aspect under consideration.

The query hints and scopes are exposed as predefined terms in the Amazon Neptune namespace `http://aws.amazon.com/neptune/vocab/v01/QueryHints#`. The examples in this section include the namespace as a `hint` prefix that is defined and included in the query:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
```

For example, the following shows how to include a `joinOrder` hint in a `SELECT` query:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
SELECT ... {
 hint:Query hint:joinOrder "Ordered" .
 ...
}
```

The preceding query instructs the Neptune engine to evaluate joins in the query in the *given* order and disables any automatic reordering.

Consider the following when using query hints:

- You can combine different query hints in a single query. For example, you can use the `bottomUp` query hint to annotate a subquery for bottom-up evaluation and a `joinOrder` query hint to fix the join order inside the subquery.
- You can use the same query hint multiple times, in different non-overlapping scopes.
- Query hints are hints. Although the query engine generally aims to consider given query hints, it might also ignore them.
- Query hints are semantics preserving. Adding a query hint does not change the output of the query (except for the potential result order when no ordering guarantees are given—that is, when the result order is not explicitly enforced by using ORDER BY).

The following sections provide more information about the available query hints and their usage in Neptune.

Topics

- Scope of SPARQL query hints in Neptune
- The joinOrder SPARQL query hint
- The evaluationStrategy SPARQL query hint

- The queryTimeout SPARQL query hint
- The rangeSafe SPARQL query hint
- The queryId SPARQL Query Hint
- The useDFE SPARQL query hint
- SPARQL query hints used with DESCRIBE

Scope of SPARQL query hints in Neptune

The following table shows the available scopes, associated hints, and descriptions for SPARQL query hints in Amazon Neptune. The `hint` prefix in these entries represents the Neptune namespace for hints:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
```

Scope	Supported Hint	Description
`hint:Query`	joinOrder	The query hint applies to the whole query.
`hint:Query`	queryTimeout	The time-out value applies to the entire query.
`hint:Query`	rangeSafe	Type promotion is disabled for the entire query.
`hint:Query`	queryId	The query ID value applies to the entire query.
`hint:Query`	useDFE	Use of the DFE is enabled (or disabled) for the entire query.
`hint:Group`	joinOrder	The query hint applies to the top-level elements in the specified group, but not to nested elements (such as subqueries) or parent elements.

Scope	Supported Hint	Description
`hint:SubQuery`	evaluationStrategy	The hint is specified and applied to a nested SELECT subquery. The subquery is evaluated independe ntly, without considering solutions computed before the subquery.

The `joinOrder` SPARQL query hint

When you submit a SPARQL query, the Amazon Neptune query engine investigates the structure of the query. It reorders parts of the query and tries to minimize the amount of work required for evaluation and query response time.

For example, a sequence of connected triple patterns is typically not evaluated in the given order. It is reordered using heuristics and statistics such as the selectivity of the individual patterns and how they are connected through shared variables. Additionally, if your query contains more complex patterns such as subqueries, FILTERs, or complex OPTIONAL or MINUS blocks, the Neptune query engine reorders them where possible, aiming for an efficient evaluation order.

For more complex queries, the order in which Neptune chooses to evaluate the query might not always be optimal. For instance, Neptune might miss instance data-specific characteristics (such as hitting power nodes in the graph) that emerge during query evaluation.

If you know the exact characteristics of the data and want to manually dictate the order of the query execution, use the Neptune `joinOrder` query hint to specify that the query be evaluated in the given order.

`joinOrder` SPARQL hint syntax

The `joinOrder` query hint is specified as a triple pattern included in a SPARQL query.

For clarity, the following syntax uses a `hint` prefix defined and included in the query to specify the Neptune query-hint namespace:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
```

```
scope hint:joinOrder "Ordered" .
```

Available Scopes

- `hint:Query`
- `hint:Group`

For more information about query hint scopes, see Scope of SPARQL query hints in Neptune.

`joinOrder` SPARQL hint example

This section shows a query written with and without the `joinOrder` query hint and related optimizations.

For this example, assume that the dataset contains the following:

- A single person named `John` that `:likes` 1,000 persons, including `Jane`.
- A single person named `Jane` that `:likes` 10 persons, including `John`.

No Query Hint

The following SPARQL query extracts all the pairs of people named `John` and `Jane` who both like each other from a set of social networking data:

```
PREFIX : <https://example.com/>
SELECT ?john ?jane {
  ?person1 :name "Jane" .
  ?person1 :likes ?person2 .
  ?person2 :name "John" .
  ?person2 :likes ?person1 .
}
```

The Neptune query engine might evaluate the statements in a different order than written. For example, it might choose to evaluate in the following order:

1. Find all persons named `John`.
2. Find all persons connected to `John` by a `:likes` edge.
3. Filter this set by persons named `Jane`.
4. Filter this set by those connected to `John` by a `:likes` edge.

According to the dataset, evaluating in this order results in 1,000 entities being extracted in the second step. The third step narrows this down to the single node, `Jane`. The final step then determines that `Jane` also `:likes` the `John` node.

Query Hint

It would be favorable to start with the `Jane` node because she has only 10 outgoing `:likes` edges. This reduces the amount of work during the evaluation of the query by avoiding the extraction of the 1,000 entities during the second step.

The following example uses the **joinOrder** query hint to ensure that the `Jane` node and its outgoing edges are processed first by disabling all automatic join reordering for the query:

```
PREFIX : <https://example.com/>
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
SELECT ?john ?jane {
  hint:Query hint:joinOrder "Ordered" .
  ?person1 :name "Jane" .
  ?person1 :likes ?person2 .
  ?person2 :name "John" .
  ?person2 :likes ?person1 .
}
```

An applicable real-world scenario might be a social network application in which persons in the network are classified as either influencers with many connections or normal users with few connections. In such a scenario, you could ensure that the normal user (`Jane`) is processed before the influencer (`John`) in a query like the preceding example.

Query Hint and Reorder

You can take this example one step further. If you know that the `:name` attribute is unique to a single node, you could speed up the query by reordering and using the `joinOrder` query hint. This step ensures that the unique nodes are extracted first.

```
PREFIX : <https://example.com/>
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
SELECT ?john ?jane {
  hint:Query hint:joinOrder "Ordered" .
  ?person1 :name "Jane" .
  ?person2 :name "John" .
  ?person1 :likes ?person2 .
```

```
  ?person2 :likes ?person1 .
}
```

In this case, you can reduce the query to the following single actions in each step:

1. Find the single person node with `:name Jane`.
2. Find the single person node with `:name John`.
3. Check that the first node is connected to the second with a `:likes` edge.
4. Check that the second node is connected to the first with a `:likes` edge.

> **Important**
>
> If you choose the wrong order, the `joinOrder` query hint can lead to significant performance drops. For example, the preceding example would be inefficient if the `:name` attributes were not unique. If all 100 nodes were named `Jane` and all 1,000 nodes were named `John`, then the query would end up checking 1,000 * 100 (100,000) pairs for `:likes` edges.

The `evaluationStrategy` SPARQL query hint

The `evaluationStrategy` query hint tells the Amazon Neptune query engine that the fragment of the query annotated should be evaluated from the bottom up, as an independent unit. This means that no solutions from previous evaluation steps are used to compute the query fragment. The query fragment is evaluated as a standalone unit, and its produced solutions are joined with the remainder of the query after it is computed.

Using the `evaluationStrategy` query hint implies a blocking (non-pipelined) query plan, meaning that the solutions of the fragment annotated with the query hint are materialized and buffered in main memory. Using this query hint might significantly increase the amount of main memory needed to evaluate the query, especially if the annotated query fragment computes a large number of results.

`evaluationStrategy` SPARQL hint syntax

The `evaluationStrategy` query hint is specified as a triple pattern included in a SPARQL query.

For clarity, the following syntax uses a `hint` prefix defined and included in the query to specify the Neptune query-hint namespace:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
hint:SubQuery hint:evaluationStrategy "BottomUp" .
```

Available Scopes

- `hint:SubQuery`

> **Note**
>
> This query hint is supported only in nested subqueries.

For more information about query hint scopes, see Scope of SPARQL query hints in Neptune.

`evaluationStrategy` SPARQL hint example

This section shows a query written with and without the `evaluationStrategy` query hint and related optimizations.

For this example, assume that the dataset has the following characteristics:

- It contains 1,000 edges labeled `:connectedTo`.
- Each `component` node is connected to an average of 100 other `component` nodes.
- The typical number of four-hop cyclical connections between nodes is around 100.

As a typical example, the `evaluationStrategy` hint can be helpful to optimize query patterns containing cycles.

No Query Hint

The following SPARQL query extracts all `component` nodes that are cyclically connected to each other via four hops:

```
PREFIX : <https://example.com/>
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
```

```
SELECT * {
  ?component1 :connectedTo ?component2 .
  ?component2 :connectedTo ?component3 .
  ?component3 :connectedTo ?component4 .
  ?component4 :connectedTo ?component1 .
}
```

The approach of the Neptune query engine is to evaluate this query using the following steps:

- Extract all 1,000 `connectedTo` edges in the graph.
- Expand by 100x (the number of outgoing `connectedTo` edges from component2).

  Intermediate results: 100,000 nodes.
- Expand by 100x (the number of outgoing `connectedTo` edges from component3).

  Intermediate results: 10,000,000 nodes.
- Scan the 10,000,000 nodes for the cycle close.

This results in a streaming query plan, which has a constant amount of main memory.

Query Hint and Subqueries

You might want to trade off main memory space for accelerated computation. By rewriting the query using an `evaluationStrategy` query hint, you can force the engine to compute a join between two smaller, materialized subsets.

```
PREFIX : <https://example.com/>
          PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
SELECT * {
  {
    SELECT * WHERE {
      hint:SubQuery hint:evaluationStrategy "BottomUp" .
      ?component1 :connectedTo ?component2 .
      ?component2 :connectedTo ?component3 .
    }
  }
  {
    SELECT * WHERE {
      hint:SubQuery hint:evaluationStrategy "BottomUp" .
      ?component3 :connectedTo ?component4 .
      ?component4 :connectedTo ?component1 .
```

```
    }
  }
}
```

Instead of evaluating the triple patterns in sequence while iteratively using results from the previous triple pattern as input for the upcoming patterns, the `evaluationStrategy` hint causes the two subqueries to be evaluated independently. Both subqueries produce 100,000 nodes for intermediate results, which are then joined together to form the final output.

In particular, when you run Neptune on the larger instance types, temporarily storing these two 100,000 subsets in main memory increases memory usage in return for significantly speeding up evaluation.

The `queryTimeout` SPARQL query hint

The `queryTimeout` query hint specifies a timeout that is shorter than the `neptune_query_timeout` value set in the DB parameters group.

If the query terminates as a result of this hint, a `TimeLimitExceededException` is thrown, with an `Operation terminated (deadline exceeded)` message. Whether to retry a timed-out query depends on the nature of the failure and your workload. For guidance, see the section called "Exceptions and Retries".

`queryTimeout` SPARQL hint syntax

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
SELECT ... WHERE {
    hint:Query hint:queryTimeout 10 .
    # OR
    hint:Query hint:queryTimeout "10" .
    # OR
    hint:Query hint:queryTimeout "10"^^xsd:integer .
 ...
}
```

The time-out value is expressed in milliseconds.

The time-out value must be smaller than the `neptune_query_timeout` value set in the DB parameters group. Otherwise, a `MalformedQueryException` exception is thrown with a `Malformed query: Query hint 'queryTimeout' must be less than neptune_query_timeout DB Parameter Group` message.

The `queryTimeout` query hint should be specified in the `WHERE` clause of the main query, or in the `WHERE` clause of one of the subqueries as shown in the example below.

It must be set only once across all the queries/subqueries and SPARQL Updates sections (such as INSERT and DELETE). Otherwise, a `MalformedQueryException` exception is thrown with a `Malformed query: Query hint 'queryTimeout' must be set only once` message.

Available Scopes

The `queryTimeout` hint can be applied both to SPARQL queries and updates.

- In a SPARQL query, it can appear in the WHERE clause of the main query or a subquery.
- In a SPARQL update, it can be set in the INSERT, DELETE, or WHERE clause. If there are multiple update clauses, it can only be set in one of them.

For more information about query hint scopes, see Scope of SPARQL query hints in Neptune.

`queryTimeout` SPARQL hint example

Here is an example of using `hint:queryTimeout` in the main `WHERE` clause of an `UPDATE` query:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
INSERT {
    ?s ?p ?o
} WHERE {
    hint:Query hint:queryTimeout 100 .
    ?s ?p ?o .
}
```

Here, the `hint:queryTimeout` is in the `WHERE` clause of a subquery:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
SELECT * {
   ?s ?p ?o .
   {
      SELECT ?s WHERE {
         hint:Query hint:queryTimeout 100 .
         ?s ?p1 ?o1 .
      }
   }
```

```
}
```

The `rangeSafe` SPARQL query hint

Use this query hint to turn off type promotion for a SPARQL query.

When you submit a SPARQL query that includes a `FILTER` over a numerical value or range, the Neptune query engine must normally use type promotion when it executes the query. This means that it has to examine values of every type that could hold the value you are filtering on.

For example, if you are filtering for values equal to 55, the engine must look for integers equal to 55, long integers equal to 55L, floats equal to 55.0, and so forth. Each type promotion requires an additional lookup on storage, which can cause an apparently simple query to take an unexpectedly long time to complete.

Often type promotion is unnecessary because you know in advance that you only need to find values of one specific type. When this is the case, you can speed up your queries dramatically by using the `rangeSafe` query hint to turn off type promotion.

`rangeSafe` SPARQL hint syntax

The `rangeSafe` query hint takes a value of `true` to turn off type promotion. It also accepts a value of `false` (the default).

Example. The following example shows how to turn off type promotion when filtering for an integer value of o greater than 1:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
SELECT * {
   ?s ?p ?o .
   hint:Prior hint:rangeSafe 'true' .
   FILTER (?o > '1'^^<http://www.w3.org/2001/XMLSchema#int>)
```

The `queryId` SPARQL Query Hint

Use this query hint to assign your own queryId value to a SPARQL query.

Example:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
SELECT * WHERE {
```

```
  hint:Query hint:queryId "4d5c4fae-aa30-41cf-9e1f-91e6b7dd6f47"
  {?s ?p ?o}}
```

The value you assign must be unique across all queries in the Neptune DB.

The `useDFE` SPARQL query hint

Use this query hint to enable use of the DFE for executing the query. By default Neptune does not use the DFE without this query hint being set to `true`, because the neptune_dfe_query_engine instance parameter defaults to `viaQueryHint`. If you set that instance parameter to `enabled`, the DFE engine is used for all queries except those having the `useDFE` query hint set to `false`.

Example of enabling use of the DFE for a query:

```
PREFIX : <https://example.com/>
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>

SELECT ?john ?jane
{
  hint:Query hint:useDFE true .
  ?person1 :name "Jane" .
  ?person1 :likes ?person2 .
  ?person2 :name "John" .
  ?person2 :likes ?person1 .
}
```

SPARQL query hints used with DESCRIBE

A SPARQL `DESCRIBE` query provides a flexible mechanism for requesting resource descriptions. However, the SPARQL specifications do not define the precise semantics of `DESCRIBE`.

Starting with engine release 1.2.0.2, Neptune supports several different `DESCRIBE` modes and algorithms that are suited to different situations.

This sample dataset can help illustrate the different modes:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <https://example.com/> .

:JaneDoe :firstName "Jane" .
:JaneDoe :knows :JohnDoe .
:JohnDoe :firstName "John" .
```

```
:JaneDoe :knows _:b1 .
_:b1 :knows :RichardRoe .

:RichardRoe :knows :JaneDoe .
:RichardRoe :firstName "Richard" .

_:s1 rdf:type rdf:Statement .
_:s1 rdf:subject :JaneDoe .
_:s1 rdf:predicate :knows .
_:s1 rdf:object :JohnDoe .
_:s1 :knowsFrom "Berlin" .

:ref_s2 rdf:type rdf:Statement .
:ref_s2 rdf:subject :JaneDoe .
:ref_s2 rdf:predicate :knows .
:ref_s2 rdf:object :JohnDoe .
:ref_s2 :knowsSince 1988 .
```

The examples below assume that a description of the resource `:JaneDoe` is being requested using a SPARQL query like this:

```
DESCRIBE <https://example.com/JaneDoe>
```

The `describeMode` SPARQL query hint

The `hint:describeMode` SPARQL query hint is used to select one of the following SPARQL `DESCRIBE` modes supported by Neptune:

The `ForwardOneStep` DESCRIBE mode

You invoke the `ForwardOneStep` mode with the `describeMode` query hint like this:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
DESCRIBE <https://example.com/JaneDoe>
{
  hint:Query hint:describeMode "ForwardOneStep"
}
```

The `ForwardOneStep` mode only returns the attributes and forward links of the resource to be described. In the example case, this means it returns the triples that have `:JaneDoe`, the resource to be described, as subject:

```
:JaneDoe :firstName "Jane" .
:JaneDoe :knows :JohnDoe .
:JaneDoe :knows _:b301990159 .
```

Note that the DESCRIBE query may return triples with blank nodes, such as `_:b301990159`, which have different IDs each time, compared to the input dataset.

The `SymmetricOneStep` DESCRIBE mode

`SymmetricOneStep` is the default DESCRIBE mode if you don't provide a query hint. You can also invoke it explicitly with the `describeMode` query hint like this:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
DESCRIBE <https://example.com/JaneDoe>
{
  hint:Query hint:describeMode "SymmetricOneStep"
}
```

Under `SymmetricOneStep` semantics, `DESCRIBE` returns the attributes, forward links, and reverse links of the resource to be described:

```
:JaneDoe :firstName "Jane" .
:JaneDoe :knows :JohnDoe .
:JaneDoe :knows _:b318767375 .

_:b318767631 rdf:subject :JaneDoe .

:RichardRoe :knows :JaneDoe .

:ref_s2 rdf:subject :JaneDoe .
```

The Concise Bounded Description (CBD) DESCRIBE mode

The Concise Bounded Description (CBD) mode is invoked using the `describeMode` query hint like this:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
DESCRIBE <https://example.com/JaneDoe>
{
  hint:Query hint:describeMode "CBD"
```

```
}
```

Under CBD semantics, `DESCRIBE` returns the Concise Bounded Description (as defined by W3C) of the resource to be described:

```
:JaneDoe :firstName "Jane" .
:JaneDoe :knows :JohnDoe .
:JaneDoe :knows _:b285212943 .
_:b285212943 :knows :RichardRoe .

_:b285213199 rdf:subject :JaneDoe .
_:b285213199 rdf:type rdf:Statement .
_:b285213199 rdf:predicate :knows .
_:b285213199 rdf:object :JohnDoe .
_:b285213199 :knowsFrom "Berlin" .

:ref_s2 rdf:subject :JaneDoe .
```

The Concise Bounded Description of an RDF resource (that is, a node in an RDF graph) is the smallest subgraph centered around that node that can stand alone. In practice this means that if you think of this graph as a tree, with the designated node as the root, there are no blank nodes (bnodes) as leaves of that tree. Since bnodes can't be addressed externally or used in subsequent queries, it's not enough for browsing the graph just to find the next single hop(s) from the current node. You also have to go far enough to find something that can be used in subsequent queries (that is, something other than a bnode).

Computing the CBD

Given a particular node (the starting node or root) in the source RDF graph, the CBD of that node is computed as follows:

1. Include in the subgraph all statements in the source graph where the *subject* of the statement is the starting node.
2. Recursively, for all statements in the subgraph thus far that have a blank node *object*, include in the subgraph all statements in the source graph where the *subject* of the statement is that blank node, and which are not already included in the subgraph.
3. Recursively, for all statements included in the subgraph thus far, for all reifications of these statements in the source graph, include the CBD beginning from the `rdf:Statement` node of each reification.

This results in a subgraph where the *object* nodes are either IRI references or literals, or blank nodes not serving as the *subject* of any statement in the graph. Note that the CBD cannot be computed using a single SPARQL SELECT or CONSTRUCT query.

The Symmetric Concise Bounded Description (SCBD) DESCRIBE mode

The Symmetric Concise Bounded Description (SCBD) mode is invoked using the `describeMode` query hint like this:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
DESCRIBE <https://example.com/JaneDoe>
{
  hint:Query hint:describeMode "SCBD"
}
```

Under SCBD semantics, `DESCRIBE` returns the Symmetric Concise Bounded Description of the resource (as defined by W3C in [Describing Linked Datasets with the VoID Vocabulary](#):

```
:JaneDoe :firstName "Jane" .
:JaneDoe :knows :JohnDoe .
:JaneDoe :knows _:b335544591 .
_:b335544591 :knows :RichardRoe .

:RichardRoe :knows :JaneDoe .

_:b335544847 rdf:subject :JaneDoe .
_:b335544847 rdf:type rdf:Statement .
_:b335544847 rdf:predicate :knows .
_:b335544847 rdf:object :JohnDoe .
_:b335544847 :knowsFrom "Berlin" .

:ref_s2 rdf:subject :JaneDoe .
```

The advantage of CBD and SCBD over the `ForwardOneStep` and `SymmetricOneStep` modes is that blank nodes are always expanded to include their representation. This may be an important advantage because you can't query a blank node using SPARQL. In addition, CBD and SCBD modes also consider reifications.

Note that the `describeMode` query hint can also be part of a `WHERE` clause:

```
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
```

```
DESCRIBE ?s
WHERE {
  hint:Query hint:describeMode "CBD" .
  ?s rdf:type <https://example.com/Person>
}
```

The `describeIterationLimit` SPARQL query hint

The `hint:describeIterationLimit` SPARQL query hint provides an **optional** constraint on the maximum number of iterative expansions to be performed for iterative DESCRIBE algorithms such as CBD and SCBD.

DESCRIBE limits are ANDed together. Therefore, if both the iteration limit and the statements limit are specified, then both limits must be met before the DESCRIBE query is cut off.

The default for this value is 5. You may set it to ZERO (0) to specify NO limit on the number of iterative expansions.

The `describeStatementLimit` SPARQL query hint

The `hint:describeStatementLimit` SPARQL query hint provides an **optional** constraint on the maximum number of statements that may be present in a DESCRIBE query response. It is only applied for iterative DESCRIBE algorithms such as CBD and SCBD.

DESCRIBE limits are ANDed together. Therefore, if both the iteration limit and the statements limit are specified, then both limits must be met before the DESCRIBE query is cut off.

The default for this value is 5000. You may set it to ZERO (0) to specify NO limit on the number of statements returned.

SPARQL DESCRIBE behavior with respect to the default graph

The SPARQL `DESCRIBE` query form lets you retrieve information about resources without knowing the structure of the data and without having to compose a query. How this information is assembled is left up to the SPARQL implementation. Neptune provides several query hints that invoke different modes and algorithms for `DESCRIBE` to use.

In Neptune's implementation, regardless of the mode, `DESCRIBE` only uses data present in the SPARQL default graph. This is consistent with the way SPARQL treats datasets (see Specifying RDF Datasets in the SPARQL specification).

In Neptune, the default graph contains all unique triples in the union of all named graphs in the database, unless particular named graphs are specified using `FROM` and/or `FROM NAMED` clauses. All RDF data in Neptune is stored in a named graph. If a triple is inserted without a named-graph context, Neptune stores it in a named graph designated `http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph`.

When one or more named graphs are specified using the `FROM` clause, the default graph is the union of all unique triples in those named graphs. If there is no `FROM` clause and there are one or more `FROM NAMED` clauses, then the default graph is empty.

SPARQL `DESCRIBE` examples

Consider the following data:

```
PREFIX ex: <https://example.com/>

GRAPH ex:g1 {
    ex:s ex:p1 "a" .
    ex:s ex:p2 "c" .
}

GRAPH ex:g2 {
    ex:s ex:p3 "b" .
    ex:s ex:p2 "c" .
}

ex:s ex:p3 "d" .
```

For this query:

```
PREFIX ex: <https://example.com/>
DESCRIBE ?s
FROM ex:g1
FROM NAMED ex:g2
WHERE {
  GRAPH ex:g2 { ?s ?p "b" . }
}
```

Neptune would return:

```
ex:s ex:p1 "a" .
```

```
ex:s ex:p2 "c" .
```

Here, the graph pattern `GRAPH ex:g2 { ?s ?p "b" }` is evaluated first, resulting in bindings for `?s`, and then the `DESCRIBE` part is evaluated over the default graph, which is now just `ex:g1`.

However, for this query:

```
PREFIX ex: <https://example.com/>
DESCRIBE ?s
FROM NAMED ex:g1
WHERE {
  GRAPH ex:g1 { ?s ?p "a" . }
}
```

Neptune would return nothing, because when a `FROM NAMED` clause is present without any `FROM` clause, the default graph is empty.

In the following query, `DESCRIBE` is used with no `FROM` or `FROM NAMED` clause present:

```
PREFIX ex: <https://example.com/>
DESCRIBE ?s
WHERE {
  GRAPH ex:g1 { ?s ?p "a" . }
}
```

In this situation, the default graph is composed of all the unique triples in the union of all the named graphs in the database (formally, the RDF merge), so Neptune would return:

```
ex:s ex:p1 "a" .
ex:s ex:p2 "c" .
ex:s ex:p3 "b" .
ex:s ex:p3 "d" .
```

SPARQL query status API

To get the status of SPARQL queries, use HTTP `GET` or `POST` to make a request to the `https://`*`your-neptune-endpoint`*`:`*`port`*`/sparql/status` endpoint.

SPARQL query status request parameters

queryId (optional)

The ID of a running SPARQL query. Only displays the status of the specified query.

SPARQL query status response syntax

```
{
    "acceptedQueryCount": integer,
    "runningQueryCount": integer,
    "queries": [
      {
        "queryId":"guid",
        "queryEvalStats":
          {
            "subqueries": integer,
            "elapsed": integer,
            "cancelled": boolean
          },
        "queryString": "string"
      }
    ]
}
```

SPARQL query status response values

acceptedQueryCount

The number of queries accepted since the last restart of the Neptune engine.

runningQueryCount

The number of currently running SPARQL queries.

queries

A list of the current SPARQL queries.

queryId

A GUID id for the query. Neptune automatically assigns this ID value to each query, or you can also assign your own ID (see Inject a Custom ID Into a Neptune Gremlin or SPARQL Query).

queryEvalStats

Statistics for this query.

subqueries

Number of subqueries in this query.

elapsed

The number of milliseconds the query has been running so far.

cancelled

True indicates that the query was cancelled.

queryString

The submitted query.

SPARQL query status example

The following is an example of the status command using `curl` and HTTP `GET`.

```
curl https://your-neptune-endpoint:port/sparql/status
```

This output shows a single running query.

```
{
    "acceptedQueryCount":9,
    "runningQueryCount":1,
    "queries": [
        {
            "queryId":"fb34cd3e-f37c-4d12-9cf2-03bb741bf54f",
            "queryEvalStats":
                {
                    "subqueries": 0,
                    "elapsed": 29256,
                    "cancelled": false
                },
            "queryString": "SELECT ?s ?p ?o WHERE {?s ?p ?o}"
        }
    ]
}
```

SPARQL query cancellation

To get the status of SPARQL queries, use HTTP `GET` or `POST` to make a request to the `https://your-neptune-endpoint:port/sparql/status` endpoint.

SPARQL query cancellation request parameters

cancelQuery

(Required) Tells the status command to cancel a query. This parameter does not take a value.

queryId

(Required) The ID of the running SPARQL query to cancel.

silent

(Optional) If `silent=true` then the running query is cancelled and the HTTP response code is 200. If `silent` is not present or `silent=false`, the query is cancelled with an HTTP 500 status code.

SPARQL query cancellation examples

Example 1: Cancellation with `silent=false`

The following is an example of the status command using `curl` to cancel a query with the `silent` parameter set to `false`:

```
curl https://your-neptune-endpoint:port/sparql/status \
  -d "cancelQuery" \
  -d "queryId=4d5c4fae-aa30-41cf-9e1f-91e6b7dd6f47" \
  -d "silent=false"
```

Unless the query has already started streaming results, the cancelled query would then return an HTTP 500 code with a response like this:

```
{
  "code": "CancelledByUserException",
  "requestId": "4d5c4fae-aa30-41cf-9e1f-91e6b7dd6f47",
  "detailedMessage": "Operation terminated (cancelled by user)"
```

```
}
```

If the query already returned an HTTP 200 code (OK) and has started streaming results before being cancelled, the timeout exception information is sent to the regular output stream.

Example 2: Cancellation with `silent=true`

The following is an example of the same status command as above except with the `silent` parameter now set to `true`:

```
curl https://your-neptune-endpoint:port/sparql/status \
  -d "cancelQuery" \
  -d "queryId=4d5c4fae-aa30-41cf-9e1f-91e6b7dd6f47" \
  -d "silent=true"
```

This command would return the same response as when `silent=false`, but the cancelled query would now return an HTTP 200 code with a response like this:

```
{
  "head" : {
    "vars" : [ "s", "p", "o" ]
  },
  "results" : {
    "bindings" : [ ]
  }
}
```

Using the SPARQL 1.1 Graph Store HTTP Protocol (GSP) in Amazon Neptune

In the SPARQL 1.1 Graph Store HTTP Protocol recommendation, the W3C defined an HTTP protocol for managing RDF graphs. It defines operations for removing, creating, and replacing RDF graph content as well as for adding RDF statements to existing content.

The graph-store protocol (GSP) provides a convenient way to manipulate your entire graph without having to write complex SPARQL queries.

Neptune fully supports this protocol.

The endpoint for the graph-store protocol (GSP) is:

```
https://your-neptune-cluster:port/sparql/gsp/
```

To access the default graph with GSP, use:

```
https://your-neptune-cluster:port/sparql/gsp/?default
```

To access a named graph with GSP, use:

```
https://your-neptune-cluster:port/sparql/gsp/?graph=named-graph-URI
```

Special details of the Neptune GSP implementation

Neptune fully implements the W3C recommendation that defines GSP. However, there are a few situations that the specification doesn't cover.

One of these is the case where a PUT or POST request specifies one or more named graphs in the request body that differ from the graph specified by the request URL. This can only happen when the request body RDF format supports named graphs, as, for example, using `Content-Type: application/n-quads` or `Content-Type: application/trig`.

In this situation, Neptune adds or updates all the named graphs present in the body, as well as the named graph specified in the URL.

For example, suppose that starting with an empty database, you send a PUT request to upsert votes into three graphs. One, named `urn:votes`, contains all votes from all election years. Two others, named `urn:votes:2005` and `urn:votes:2019`, contain votes from specific election years. The request and its payload look like this:

```
PUT "http://your-Neptune-cluster:port/sparql/gsp/?graph=urn:votes"
  Host: example.com
  Content-Type: application/n-quads

  PAYLOAD:

  <urn:JohnDoe> <urn:votedFor> <urn:Labour> <urn:votes:2005>
  <urn:JohnDoe> <urn:votedFor> <urn:Conservative> <urn:votes:2019>
  <urn:JaneSmith> <urn:votedFor> <urn:LiberalDemocrats> <urn:votes:2005>
  <urn:JaneSmith> <urn:votedFor> <urn:Conservative> <urn:votes:2019>
```

After the request is executed, the data in the database looks like this:

```
<urn:JohnDoe>   <urn:votedFor> <urn:Labour>           <urn:votes:2005>
<urn:JohnDoe>   <urn:votedFor> <urn:Conservative>     <urn:votes:2019>
<urn:JaneSmith> <urn:votedFor> <urn:LiberalDemocrats> <urn:votes:2005>
<urn:JaneSmith> <urn:votedFor> <urn:Conservative>     <urn:votes:2019>
<urn:JohnDoe>   <urn:votedFor> <urn:Labour>           <urn:votes>
<urn:JohnDoe>   <urn:votedFor> <urn:Conservative>     <urn:votes>
<urn:JaneSmith> <urn:votedFor> <urn:LiberalDemocrats> <urn:votes>
<urn:JaneSmith> <urn:votedFor> <urn:Conservative>     <urn:votes>
```

Another ambiguous situation is where more than one graph is specied in the request URL itself, using any of `PUT`, `POST`, `GET` or `DELETE`. For example:

```
POST "http://your-Neptune-cluster:port/sparql/gsp/?
graph=urn:votes:2005&graph=urn:votes:2019"
```

Or:

```
GET "http://your-Neptune-cluster:port/sparql/gsp/?default&graph=urn:votes:2019"
```

In this situation, Neptune returns an HTTP 400 with a message indicating that only one graph can be specified in the request URL.

Analyzing Neptune query execution using SPARQL `explain`

Amazon Neptune has added a SPARQL feature named *explain*. This feature is a self-service tool for understanding the execution approach taken by the Neptune engine. You invoke it by adding an `explain` parameter to an HTTP call that submits a SPARQL query.

The `explain` feature provides information about the logical structure of query execution plans. You can use this information to identify potential evaluation and execution bottlenecks. You can then use query hints to improve your query execution plans.

Topics

- How the SPARQL query engine works in Neptune
- How to use SPARQL explain to analyze Neptune query execution
- Examples of invoking SPARQL explain in Neptune
- Neptune SPARQL explain operators

- Limitations of SPARQL explain in Neptune

How the SPARQL query engine works in Neptune

To use the information that the SPARQL `explain` feature provides, you need to understand some details about how the Amazon Neptune SPARQL query engine works.

The engine translates every SPARQL query into a pipeline of operators. Starting from the first operator, intermediate solutions known as *binding lists* flow through this operator pipeline. You can think of a binding list as a table in which the table headers are a subset of the variables used in the query. Each row in the table represents a result, up to the point of evaluation.

Let's assume that two namespace prefixes have been defined for our data:

```
@prefix ex:   <http://example.com> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

The following would be an example of a simple binding list in this context:

```
?person       | ?firstName
-------------------------------------------------------
ex:JaneDoe    | "Jane"
ex:JohnDoe    | "John"
ex:RichardRoe | "Richard"
```

For each of three people, the list binds the `?person` variable to an identifier of the person, and the `?firstName` variable to the person's first name.

In the general case, variables can remain unbound, if, for example, there is an `OPTIONAL` selection of a variable in a query for which no value is present in the data.

The `PipelineJoin` operator is an example of a Neptune query engine operator present in the `explain` output. It takes as input an incoming binding set from the previous operator and joins it against a triple pattern, say `(?person, foaf:lastName, ?lastName)`. This operation uses the bindings for the `?person` variable in its input stream, substitutes them into the triple pattern, and looks up triples from the database.

When executed in the context of the incoming bindings from the previous table, `PipelineJoin` would evaluate three lookups, namely the following:

```
(ex:JaneDoe,    foaf:lastName, ?lastName)
(ex:JohnDoe,    foaf:lastName, ?lastName)
(ex:RichardRoe, foaf:lastName, ?lastName)
```

This approach is called *as-bound* evaluation. The solutions from this evaluation process are joined back against the incoming solutions, padding the detected `?lastName` in the incoming solutions. Assuming that you find a last name for all three persons, the operator would produce an outgoing binding list that would look something like this:

```
?person       | ?firstName | ?lastName
---------------------------------------
ex:JaneDoe    | "Jane"     | "Doe"
ex:JohnDoe    | "John"     | "Doe"
ex:RichardRoe | "Richard"  | "Roe"
```

This outgoing binding list then serves as input for the next operator in the pipeline. At the end, the output of the last operator in the pipeline defines the query result.

Operator pipelines are often linear, in the sense that every operator emits solutions for a single connected operator. However, in some cases, they can have more complex structures. For example, a `UNION` operator in a SPARQL query is mapped to a `Copy` operation. This operation duplicates the bindings and forwards the copies into two subplans, one for the left side and the other for the right side of the `UNION`.

For more information about operators, see Neptune SPARQL `explain` operators.

How to use SPARQL `explain` to analyze Neptune query execution

The SPARQL `explain` feature is a self-service tool in Amazon Neptune that helps you understand the execution approach taken by the Neptune engine. To invoke `explain`, you pass a parameter to an HTTP or HTTPS request in the form `explain=`*`mode`*.

The mode value can be one of `static` `dynamic`, or `details`:

- In *static* mode, `explain` prints only the static structure of the query plan.
- In *dynamic* mode, `explain` also includes dynamic aspects of the query plan. These aspects might include the number of intermediate bindings flowing through the operators, the ratio of incoming bindings to outgoing bindings, and the total time taken by operators.

- In *details* mode, `explain` prints the information shown in `dynamic` mode plus additional details such as the actual SPARQL query string and the estimated range count for the pattern underlying a join operator.

Neptune supports using `explain` with all three SPARQL query access protocols listed in the W3C SPARQL 1.1 Protocol specification, namely:

1. HTTP GET
2. HTTP POST using URL-encoded parameters
3. HTTP POST using text parameters

For information about the SPARQL query engine, see How the SPARQL query engine works in Neptune.

For information about the kind of output produced by invoking SPARQL `explain`, see Examples of invoking SPARQL `explain` in Neptune.

Examples of invoking SPARQL `explain` in Neptune

The examples in this section show the various kinds of output you can produce by invoking the SPARQL `explain` feature to analyze query execution in Amazon Neptune.

Topics

- Understanding Explain Output
- Example of details mode output
- Example of static mode output
- Different ways of encoding parameters
- Other output types besides text/plain
- Example of SPARQL explain output when the DFE is enabled

Understanding Explain Output

In this example, Jane Doe knows two people, namely John Doe and Richard Roe:

```
@prefix ex: <http://example.com> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

```
ex:JaneDoe foaf:knows ex:JohnDoe .
ex:JohnDoe foaf:firstName "John" .
ex:JohnDoe foaf:lastName "Doe" .
ex:JaneDoe foaf:knows ex:RichardRoe .
ex:RichardRoe foaf:firstName "Richard" .
ex:RichardRoe foaf:lastName "Roe" .
.
```

To determine the first names of all the people whom Jane Doe knows, you can write the following query:

```
 curl http(s)://your_server:your_port/sparql \
   -d "query=PREFIX foaf: <https://xmlns.com/foaf/0.1/> PREFIX ex: <https://
www.example.com/> \
       SELECT ?firstName WHERE { ex:JaneDoe foaf:knows ?person . ?person
 foaf:firstName ?firstName }" \
   -H "Accept: text/csv"
```

This simple query returns the following:

```
firstName
John
Richard
```

Next, change the `curl` command to invoke `explain` by adding `-d "explain=dynamic"` and using the default output type instead of `text/csv`:

```
 curl http(s)://your_server:your_port/sparql \
   -d "query=PREFIX foaf: <https://xmlns.com/foaf/0.1/> PREFIX ex: <https://
www.example.com/> \
       SELECT ?firstName WHERE { ex:JaneDoe foaf:knows ?person . ?person
 foaf:firstName ?firstName }" \
   -d "explain=dynamic"
```

The query now returns output in pretty-printed ASCII format (HTTP content type `text/plain`), which is the default output type:

```
#######################################################################################
# ID # Out #1 # Out #2 # Name              # Arguments
            # Mode     # Units In # Units Out # Ratio # Time (ms) #
```

```
##########################################################################################
# 0  # 1      # -      # SolutionInjection # solutions=[{}]
            # -        # 0        # 1         # 0.00  # 0         #
##########################################################################################
# 1  # 2      # -      # PipelineJoin      # pattern=distinct(ex:JaneDoe, foaf:knows, ?
person)     # -        # 1        # 2         # 2.00  # 1         #
#    #        #        #                   # joinType=join
            #          #          #           #       #           #
#    #        #        #                   # joinProjectionVars=[?person]
            #          #          #           #       #           #
##########################################################################################
# 2  # 3      # -      # PipelineJoin      # pattern=distinct(?person,
 foaf:firstName, ?firstName) # -        # 2        # 2         # 1.00  # 1         #
#    #        #        #                   # joinType=join
            #          #          #           #       #           #
#    #        #        #                   # joinProjectionVars=[?person, ?firstName]
            #          #          #           #       #           #
##########################################################################################
# 3  # 4      # -      # Projection        # vars=[?firstName]
            # retain   # 2        # 2         # 1.00  # 0         #
##########################################################################################
# 4  # -      # -      # TermResolution    # vars=[?firstName]
            # id2value # 2        # 2         # 1.00  # 1         #
##########################################################################################
```

For details about the operations in the `Name` column and their arguments, see explain operators.

The following describes the output row by row:

1. The first step in the main query always uses the `SolutionInjection` operator to inject a solution. The solution is then expanded to the final result through the evaluation process.

   In this case, it injects the so-called universal solution `{ }`. In the presence of `VALUES` clauses or a `BIND`, this step might also inject more complex variable bindings to start out with.

   The `Units Out` column indicates that this single solution flows out of the operator. The `Out #1` column specifies the operator into which this operator feeds the result. In this example, all operators are connected to the operator that follows in the table.
2. The second step is a `PipelineJoin`. It receives as input the single universal (fully unconstrained) solution produced by the previous operator (`Units In := 1`). It joins it against the tuple pattern defined by its `pattern` argument. This corresponds to a simple lookup for the pattern. In this case, the triple pattern is defined as the following:

```
distinct( ex:JaneDoe, foaf:knows, ?person )
```

The `joinType := join` argument indicates that this is a normal join (other types include `optional` joins, `existence check` joins, and so on).

The `distinct := true` argument says that you extract only distinct matches from the database (no duplicates), and you bind the distinct matches to the variable `joinProjectionVars := ?person`, deduplicated.

The fact that the `Units Out` column value is 2 indicates that there are two solutions flowing out. Specifically, these are the bindings for the `?person` variable, reflecting the two people that the data shows that Jane Doe knows:

```
?person
-------------
ex:JohnDoe
ex:RichardRoe
```

3. The two solutions from stage 2 flow as input (`Units In := 2`) into the second `PipelineJoin`. This operator joins the two previous solutions with the following triple pattern:

```
distinct(?person, foaf:firstName, ?firstName)
```

The `?person` variable is known to be bound either to `ex:JohnDoe` or to `ex:RichardRoe` by the operator's incoming solution. Given that, the `PipelineJoin` extracts the first names, John and Richard. The outgoing two solutions (Units Out := 2) are then as follows:

```
?person       | ?firstName
---------------------------
ex:JohnDoe    | John
ex:RichardRoe | Richard
```

4. The next projection operator takes as input the two solutions from stage 3 (`Units In := 2`) and projects onto the `?firstName` variable. This eliminates all other variable bindings in the mappings and passes on the two bindings (`Units Out := 2`):

```
?firstName
----------
John
```

```
Richard
```

5. To improve performance, Neptune operates where possible on internal identifiers that it assigns to terms such as URIs and string literals, rather than on the strings themselves. The final operator, `TermResolution`, performs a mapping from these internal identifiers back to the corresponding term strings.

   In regular (non-explain) query evaluation, the result computed by the last operator is then serialized into the requested serialization format and streamed to the client.

Example of details mode output

Suppose that you run the same query as the previous in *details* mode instead of *dynamic* mode:

```
 curl http(s)://your_server:your_port/sparql \
   -d "query=PREFIX foaf: <https://xmlns.com/foaf/0.1/> PREFIX ex: <https://
www.example.com/> \
       SELECT ?firstName WHERE { ex:JaneDoe foaf:knows ?person . ?person
 foaf:firstName ?firstName }" \
   -d "explain=details"
```

As this example shows, the output is the same with some additional details such as the query string at the top of the output, and the `patternEstimate` count for the `PipelineJoin` operator:

```
Query:
PREFIX foaf: <https://xmlns.com/foaf/0.1/> PREFIX ex: <https://www.example.com/>
SELECT ?firstName WHERE { ex:JaneDoe foaf:knows ?person . ?person foaf:firstName ?
firstName }

##########################################################################################
# ID # Out #1 # Out #2 # Name              # Arguments
            # Mode     # Units In # Units Out # Ratio # Time (ms) #
##########################################################################################
# 0  # 1      # -      # SolutionInjection # solutions=[{}]
            # -        # 0        # 1         # 0.00  # 0         #
##########################################################################################
# 1  # 2      # -      # PipelineJoin      # pattern=distinct(ex:JaneDoe, foaf:knows, ?
person)     # -        # 1        # 2         # 2.00  # 13        #
#    #        #        #                   # joinType=join
            #          #          #           #       #           #
#    #        #        #                   # joinProjectionVars=[?person]
            #          #          #           #       #           #
```

```
#    #        #        #                   # patternEstimate=2
            #          #          #           #       #           #
#######################################################################################
# 2  # 3      # -      # PipelineJoin      # pattern=distinct(?person,
 foaf:firstName, ?firstName) # -        # 2        # 2         # 1.00  # 3         #
#    #        #        #                   # joinType=join
            #          #          #           #       #           #
#    #        #        #                   # joinProjectionVars=[?person, ?firstName]
            #          #          #           #       #           #
#    #        #        #                   # patternEstimate=2
            #          #          #           #       #           #
#######################################################################################
# 3  # 4      # -      # Projection        # vars=[?firstName]
            # retain   # 2        # 2         # 1.00  # 1         #
#######################################################################################
# 4  # -      # -      # TermResolution    # vars=[?firstName]
            # id2value # 2        # 2         # 1.00  # 7         #
#######################################################################################
```

Example of static mode output

Suppose that you run the same query as the previous in *static* mode (the default) instead of *details* mode:

```
 curl http(s)://your_server:your_port/sparql \
   -d "query=PREFIX foaf: <https://xmlns.com/foaf/0.1/> PREFIX ex: <https://
www.example.com/> \
       SELECT ?firstName WHERE { ex:JaneDoe foaf:knows ?person . ?person
 foaf:firstName ?firstName }" \
   -d "explain=static"
```

As this example shows, the output is the same, except that it omits the last three columns:

```
#######################################################################################
# ID # Out #1 # Out #2 # Name              # Arguments
            # Mode     #
#######################################################################################
# 0  # 1      # -      # SolutionInjection # solutions=[{}]
            # -        #
#######################################################################################
# 1  # 2      # -      # PipelineJoin      # pattern=distinct(ex:JaneDoe, foaf:knows, ?
person)     # -        #
```

```
#    #        #        #                   # joinType=join
            #          #
#    #        #        #                   # joinProjectionVars=[?person]
            #          #
##########################################################################################
# 2  # 3      # -      # PipelineJoin      # pattern=distinct(?person,
 foaf:firstName, ?firstName) # -        #
#    #        #        #                   # joinType=join
            #          #
#    #        #        #                   # joinProjectionVars=[?person, ?firstName]
            #          #
##########################################################################################
# 3  # 4      # -      # Projection        # vars=[?firstName]
            # retain   #
##########################################################################################
# 4  # -      # -      # TermResolution    # vars=[?firstName]
            # id2value #
##########################################################################################
```

Different ways of encoding parameters

The following example queries illustrate two different ways to encode parameters when invoking SPARQL `explain`.

Using URL encoding – This example uses URL encoding of parameters, and specifies *dynamic* output:

```
curl -XGET "http(s)://your_server:your_port/sparql?query=SELECT%20*%20WHERE%20%7B
%20%3Fs%20%3Fp%20%3Fo%20%7D%20LIMIT%20%31&explain=dynamic"
```

Specifying the parameters directly – This is the same as the previous query except that it passes the parameters through POST directly:

```
curl http(s)://your_server:your_port/sparql \
  -d "query=SELECT * WHERE { ?s ?p ?o } LIMIT 1" \
  -d "explain=dynamic"
```

Other output types besides text/plain

The preceding examples use the default `text/plain` output type. Neptune can also format SPARQL `explain` output in two other MIME-type formats, namely `text/csv` and `text/html`.

You invoke them by setting the HTTP `Accept` header, which you can do using the `-H` flag in `curl`, as follows:

```
  -H "Accept: output type"
```

Here are some examples:

`text/csv` Output

This query calls for CSV MIME-type output by specifying `-H "Accept: text/csv"`:

```
curl http(s)://your_server:your_port/sparql \
  -d "query=SELECT * WHERE { ?s ?p ?o } LIMIT 1" \
  -d "explain=dynamic" \
  -H "Accept: text/csv"
```

The CSV format, which is handy for importing into a spreadsheet or database, separates the fields in each `explain` row by semicolons (`;`), like this:

```
ID;Out #1;Out #2;Name;Arguments;Mode;Units In;Units Out;Ratio;Time (ms)
0;1;-;SolutionInjection;solutions=[{}];-;0;1;0.00;0
1;2;-;PipelineJoin;pattern=distinct(?s, ?p, ?o),joinType=join,joinProjectionVars=[?s, ?
p, ?o];-;1;6;6.00;1
2;3;-;Projection;vars=[?s, ?p, ?o];retain;6;6;1.00;2
3;-;-;Slice;limit=1;-;1;1;1.00;1
```

`text/html` Output

If you specify `-H "Accept: text/html"`, then `explain` generates an HTML table:

```
<!DOCTYPE html>
<html>
  <body>
    <table border="1px">
      <thead>
        <tr>
          <th>ID</th>
          <th>Out #1</th>
          <th>Out #2</th>
          <th>Name</th>
```

```
          <th>Arguments</th>
          <th>Mode</th>
          <th>Units In</th>
          <th>Units Out</th>
          <th>Ratio</th>
          <th>Time (ms)</th>
        </tr>
      </thead>

      <tbody>
        <tr>
          <td>0</td>
          <td>1</td>
          <td>-</td>
          <td>SolutionInjection</td>
          <td>solutions=[{}]</td>
          <td>-</td>
          <td>0</td>
          <td>1</td>
          <td>0.00</td>
          <td>0</td>
        </tr>

        <tr>
          <td>1</td>
          <td>2</td>
          <td>-</td>
          <td>PipelineJoin</td>
          <td>pattern=distinct(?s, ?p, ?o)<br>
              joinType=join<br>
              joinProjectionVars=[?s, ?p, ?o]</td>
          <td>-</td>
          <td>1</td>
          <td>6</td>
          <td>6.00</td>
          <td>1</td>
        </tr>

        <tr>
          <td>2</td>
          <td>3</td>
          <td>-</td>
          <td>Projection</td>
          <td>vars=[?s, ?p, ?o]</td>
```

```
          <td>retain</td>
          <td>6</td>
          <td>6</td>
          <td>1.00</td>
          <td>2</td>
        </tr>

        <tr>
          <td>3</td>
          <td>-</td>
          <td>-</td>
          <td>Slice</td>
          <td>limit=1</td>
          <td>-</td>
          <td>1</td>
          <td>1</td>
          <td>1.00</td>
          <td>1</td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```

The HTML renders in a browser something like the following:

ID	Out #1	Out #2	Name	Arguments	Mode	Units In	Units Out	Ratio	Time (ms)
0	1	-	SolutionInjection	solutions=[{}]	-	0	1	0.00	0
1	2	-	PipelineJoin	pattern=distinct(?s, ?p, ?o) joinType=join joinProjectionVars=[?s, ?p, ?o]	-	1	6	6.00	1
2	3	-	Projection	vars=[?s, ?p, ?o]	retain	6	6	1.00	2
3	-	-	Slice	limit=1	-	1	1	1.00	1

Example of SPARQL `explain` output when the DFE is enabled

The following is an example of SPARQL `explain` output when the Neptune DFE alternative query engine is enabled:

```
#######################################################################################
# ID # Out #1 # Out #2 # Name              # Arguments


 # Mode     # Units In # Units Out # Ratio # Time (ms) #
#######################################################################################
# 0  # 1      # -      # SolutionInjection # solutions=[{}]
```

```
 # -        # 0        # 1         # 0.00  # 0         #
##########################################################################################
# 1  # 2      # -      # HashIndexBuild    # solutionSet=solutionSet1


 # -        # 1        # 1         # 1.00  # 22        #
#    #        #        #                   # joinVars=[]


 #          #          #           #       #           #
#    #        #        #                   # sourceType=pipeline


 #          #          #           #       #           #
##########################################################################################
# 2  # 3      # -      # DFENode           # DFE Stats=


      # -        # 101      # 100       # 0.99  # 32        #
#    #        #        #                   # ====> DFE execution time (measured by
 DFEQueryEngine)

     #          #          #           #       #           #
#    #        #        #                   # accepted [micros]=127


 #          #          #           #       #           #
#    #        #        #                   # ready [micros]=2


 #          #          #           #       #           #
#    #        #        #                   # running [micros]=5627


 #          #          #           #       #           #
#    #        #        #                   # finished [micros]=0


 #          #          #           #       #           #
#    #        #        #                   #


 #          #          #           #       #           #
```

```
#    #        #        #                   #


 #          #          #           #       #           #
#    #        #        #                   # ===> DFE execution time (measured in
 DFENode)

      #          #          #           #       #           #
#    #        #        #                   # -> setupTime [ms]=1


 #          #          #           #       #           #
#    #        #        #                   # -> executionTime [ms]=14


 #          #          #           #       #           #
#    #        #        #                   # -> resultReadTime [ms]=0


 #          #          #           #       #           #
#    #        #        #                   #


 #          #          #           #       #           #
#    #        #        #                   #


 #          #          #           #       #           #
#    #        #        #                   # ===> Static analysis statistics


 #          #          #           #       #           #
#    #        #        #                   # --> 35907 micros spent in parser.


 #          #          #           #       #           #
#    #        #        #                   # --> 7643 micros spent in range count
 estimation

      #          #          #           #       #           #
#    #        #        #                   # --> 2895 micros spent in value resolution


 #          #          #           #       #           #
```

```
#    #        #        #                   #


 #          #          #           #       #           #
#    #        #        #                   # --> 39974925 micros spent in optimizer
 loop

    #          #          #           #       #           #
#    #        #        #                   #


 #          #          #           #       #           #
#    #        #        #                   #


 #          #          #           #       #           #
#    #        #        #                   # DFEJoinGroupNode[ children={


  #          #          #           #       #           #
#    #        #        #                   #   DFEPatternNode[(?1, TERM[117442062], ?
2, ?3) . project DISTINCT[?1, ?2] {rangeCountEstimate=100},

  #          #          #           #       #           #
#    #        #        #                   #     OperatorInfoWithAlternative[


  #          #          #           #       #           #
#    #        #        #                   #       rec=OperatorInfo[


  #          #          #           #       #           #
#    #        #        #                   #         type=INCREMENTAL_PIPELINE_JOIN,


 #          #          #           #       #           #
#    #        #        #                   #
 costEstimates=OperatorCostEstimates[

                                     #          #          #           #       #
  #
#    #        #        #                   #
 costEstimate=OperatorCostEstimate[in=1.0000,out=100.0000,io=0.0002,comp=0.0000,mem=0],
```

```
                                   #           #           #            #       #
 #
#    #         #         #                    #
 worstCaseCostEstimate=OperatorCostEstimate[in=1.0000,out=100.0000,io=0.0002,comp=0.0000,mem=0]

                      #           #           #            #       #            #
#    #         #         #                    #       alt=OperatorInfo[


  #           #           #            #       #            #
#    #         #         #                    #         type=INCREMENTAL_HASH_JOIN,


 #           #           #            #       #            #
#    #         #         #                    #
 costEstimates=OperatorCostEstimates[

                                   #           #           #            #       #
  #
#    #         #         #                    #
 costEstimate=OperatorCostEstimate[in=1.0000,out=100.0000,io=0.0003,comp=0.0000,mem=3212],

                               #           #           #            #       #            #
#    #         #         #                    #
 worstCaseCostEstimate=OperatorCostEstimate[in=1.0000,out=100.0000,io=0.0003,comp=0.0000,mem=32

                  #           #           #            #       #            #
#    #         #         #                    #   DFEPatternNode[(?1, TERM[150997262], ?
4, ?5) . project DISTINCT[?1, ?4] {rangeCountEstimate=100},

  #           #           #            #       #            #
#    #         #         #                    #     OperatorInfoWithAlternative[


  #           #           #            #       #            #
#    #         #         #                    #       rec=OperatorInfo[


  #           #           #            #       #            #
#    #         #         #                    #         type=INCREMENTAL_HASH_JOIN,


 #           #           #            #       #            #
```

```
#    #        #        #                   #
 costEstimates=OperatorCostEstimates[

                                   #          #          #           #       #
  #
#    #        #        #                   #
 costEstimate=OperatorCostEstimate[in=100.0000,out=100.0000,io=0.0003,comp=0.0000,mem=6400],

                              #          #          #           #       #           #
#    #        #        #                   #
 worstCaseCostEstimate=OperatorCostEstimate[in=100.0000,out=100.0000,io=0.0003,comp=0.0000,mem=

                 #          #          #           #       #           #
#    #        #        #                   #       alt=OperatorInfo[


  #          #          #           #       #           #
#    #        #        #                   #         type=INCREMENTAL_PIPELINE_JOIN,


 #          #          #           #       #           #
#    #        #        #                   #
 costEstimates=OperatorCostEstimates[

                                   #          #          #           #       #
  #
#    #        #        #                   #
 costEstimate=OperatorCostEstimate[in=100.0000,out=100.0000,io=0.0010,comp=0.0000,mem=0],

                                #          #          #           #       #           #
#    #        #        #                   #
 worstCaseCostEstimate=OperatorCostEstimate[in=100.0000,out=100.0000,io=0.0010,comp=0.0000,mem=

                    #          #          #           #       #           #
#    #        #        #                   # },


 #          #          #           #       #           #
#    #        #        #                   # ]


 #          #          #           #       #           #
#    #        #        #                   #
```

```
 #          #          #           #       #           #
#    #        #        #                   # ===> DFE configuration:


 #          #          #           #       #           #
#    #        #        #                   # solutionChunkSize=5000


 #          #          #           #       #           #
#    #        #        #                   # ouputQueueSize=20


 #          #          #           #       #           #
#    #        #        #                   # numComputeCores=3


 #          #          #           #       #           #
#    #        #        #                   # maxParallelIO=10


 #          #          #           #       #           #
#    #        #        #                   # numInitialPermits=12


 #          #          #           #       #           #
#    #        #        #                   #


 #          #          #           #       #           #
#    #        #        #                   #


 #          #          #           #       #           #
#    #        #        #                   # ====> DFE configuration (reported back)


 #          #          #           #       #           #
#    #        #        #                   # numComputeCores=3


 #          #          #           #       #           #
#    #        #        #                   # maxParallelIO=2
```

```
 #          #          #           #       #           #
#    #        #        #                   # numInitialPermits=12


 #          #          #           #       #           #
#    #        #        #                   #


 #          #          #           #       #           #
#    #        #        #                   # ===> Statistics & operator histogram


 #          #          #           #       #           #
#    #        #        #                   # ==> Statistics


 #          #          #           #       #           #
#    #        #        #                   # -> 3741 / 3668 micros total elapsed (incl.
 wait / excl. wait)

 #          #          #           #       #           #
#    #        #        #                   # -> 3741 / 3 millis total elapse (incl.
 wait / excl. wait)

    #          #          #           #       #           #
#    #        #        #                   # -> 3741 / 0 secs total elapsed (incl.
 wait / excl. wait)

     #          #          #           #       #           #
#    #        #        #                   # ==> Operator histogram


 #          #          #           #       #           #
#    #        #        #                   # -> 47.66% of total time (excl. wait):
 pipelineScan (2 instances)

     #          #          #           #       #           #
#    #        #        #                   # -> 10.99% of total time (excl. wait):
 merge (1 instances)

     #          #          #           #       #           #
#    #        #        #                   # -> 41.17% of total time (excl. wait):
 symmetricHashJoin (1 instances)
```

```
     #          #          #           #       #           #
#    #        #        #                   # -> 0.19% of total time (excl. wait): drain
 (1 instances)

 #          #          #           #       #           #
#    #        #        #                   #



 #          #          #           #       #           #
#    #        #        #                   # nodeId | out0   | out1 | opName
   | args                                            | rowsIn | rowsOut | chunksIn |
 chunksOut | elapsed* | outWait | outBlocked | ratio    | rate* [M/s] | rate [M/s] | %
    #          #          #           #       #           #
#    #        #        #                   # ------ | ------ | ---- | -----------------
 | ------------------------------------------------ | ------ | ------- | -------- |
 --------- | -------- | ------- | ---------- | -------- | ----------- | ---------- |
 ----- #          #          #           #       #           #
#    #        #        #                   # node_0 | node_2 | -    | pipelineScan
  | (?1, TERM[117442062], ?2, ?3) DISTINCT [?1, ?2]  | 0      | 100     | 0        | 1
        | 874      | 0       | 0          | Infinity | 0.1144      | 0.1144     | 23.83
 #          #          #           #       #           #
#    #        #        #                   # node_1 | node_2 | -    | pipelineScan
  | (?1, TERM[150997262], ?4, ?5) DISTINCT [?1, ?4]  | 0      | 100     | 0        | 1
        | 874      | 0       | 0          | Infinity | 0.1144      | 0.1144     | 23.83
 #          #          #           #       #           #
#    #        #        #                   # node_2 | node_4 | -    | symmetricHashJoin
 |                                                  | 200    | 100     | 2        | 2
        | 1510     | 73      | 0          | 0.50     | 0.0662      | 0.0632     | 41.17
 #          #          #           #       #           #
#    #        #        #                   # node_3 | -      | -    | drain
  |                                                  | 100    | 0       | 1        | 0
        | 7        | 0       | 0          | 0.00     | 0.0000      | 0.0000     | 0.19
 #          #          #           #       #           #
#    #        #        #                   # node_4 | node_3 | -    | merge
  |                                                  | 100    | 100     | 2        | 1
        | 403      | 0       | 0          | 1.00     | 0.2481      | 0.2481     | 10.99
 #          #          #           #       #           #
##########################################################################################
# 3  # 4      # -      # HashIndexJoin     # solutionSet=solutionSet1


 # -        # 100      # 100       # 1.00  # 4         #
```

```
#    #        #        #                   # joinType=join


 #          #          #           #       #           #
##########################################################################################
# 4  # 5      # -      # Distinct          # vars=[?s, ?o, ?o1]


 # -        # 100      # 100       # 1.00  # 9         #
##########################################################################################
# 5  # 6      # -      # Projection        # vars=[?s, ?o, ?o1]


 # retain   # 100      # 100       # 1.00  # 2         #
##########################################################################################
# 6  # -      # -      # TermResolution    # vars=[?s, ?o, ?o1]


 # id2value # 100      # 100       # 1.00  # 11        #
##########################################################################################
```

Neptune SPARQL `explain` operators

The following sections describe the operators and parameters for the SPARQL `explain` feature currently available in Amazon Neptune.

> **Important**
>
> The SPARQL `explain` feature is still being refined. The operators and parameters documented here might change in future versions.

Topics

- Aggregation operator
- ConditionalRouting operator
- Copy operator
- DFENode operator
- Distinct operator
- Federation operator

- Filter operator
- HashIndexBuild operator
- HashIndexJoin operator
- MergeJoin operator
- NamedSubquery operator
- PipelineJoin operator
- PipelineCountJoin operator
- PipelinedHashIndexJoin operator
- Projection operator
- PropertyPath operator
- TermResolution operator
- Slice operator
- SolutionInjection operator
- Sort operator
- VariableAlignment operator

`Aggregation` operator

Performs one or more aggregations, implementing the semantics of SPARQL aggregation operators such as `count`, `max`, `min`, `sum`, and so on.

`Aggregation` comes with optional grouping using `groupBy` clauses, and optional `having` constraints.

Arguments

- `groupBy` – (*Optional*) Provides a `groupBy` clause that specifies the sequence of expressions according to which the incoming solutions are grouped.
- `aggregates` – (*Required*) Specifies an ordered list of aggregation expressions.
- `having` – (*Optional*) Adds constraints to filter on groups, as implied by the `having` clause in the SPARQL query.

`ConditionalRouting` operator

Routes incoming solutions based on a given condition. Solutions that satisfy the condition are routed to the operator ID referenced by `Out #1`, whereas solutions that do not are routed to the operator referenced by `Out #2`.

Arguments

- `condition` – (*Required*) The routing condition.

Copy operator

Delegates the solution stream as specified by the specified mode.

Modes

- `forward` – Forwards the solutions to the downstream operator identified by `Out #1`.
- `duplicate` – Duplicates the solutions and forwards them to each of the two operators identified by `Out #1` and `Out #2`.

`Copy` has no arguments.

`DFENode` operator

This operator is an abstraction of the plan that is run by the DFE alternative query engine. The detailed DFE plan is outlined in the arguments for this operator. The argument is currently overloaded to contain the detailed runtime statistics of the DFE plan. It contains the time spent in the various steps of query execution by DFE.

The logical optimized abstract syntax tree (AST) for the DFE query plan is printed with information about the operator types that were considered while planning and the associated best- and worst-case costs to run the operators. The AST consists of the following type of nodes at the moment:

- `DFEJoinGroupNode` – Represents a join of one or more `DFEPatternNodes`.
- `DFEPatternNode` – Encapsulates an underlying pattern using which matching tuples are projected out of the underlying database.

The sub-section, `Statistics & Operator histogram`, contains details about the execution time of the `DataflowOp` plan and the breakdown of CPU time used by each operator. Below this there is a table which prints detailed runtime statistics of the plan executed by DFE.

> **Note**
>
> Because DFE support for SPARQL is an experimental feature, the exact format of its `explain` output may change.

`Distinct` operator

Computes the distinct projection on a subset of the variables, eliminating duplicates. As a result, the number of solutions flowing in is larger than or equal to the number of solutions flowing out.

Arguments

- `vars` – (*Required*) The variables to which to apply the `Distinct` projection.

`Federation` operator

Passes a specified query to a specified remote SPARQL endpoint.

Arguments

- `endpoint` – (*Required*) The endpoint URL in the SPARQL `SERVICE` statement. This can be a constant string, or if the query endpoint is determined based on a variable within the same query, it can be the variable name.
- `query` – (*Required*) The reconstructed query string to be sent to the remote endpoint. The engine adds default prefixes to this query even when the client doesn't specify any.
- `silent` – (*Required*) A Boolean that indicates whether the `SILENT` keyword appeared after the keyword. `SILENT` tells the engine not to fail the whole query even if the remote `SERVICE` portion fails.

`Filter` operator

Filters the incoming solutions. Only those solutions that satisfy the filter condition are forwarded to the upstream operator, and all others are dropped.

Arguments

- `condition` – (*Required*) The filter condition.

`HashIndexBuild` operator

Takes a list of bindings and spools them into a hash index whose name is defined by the `solutionSet` argument. Typically, subsequent operators perform joins against this solution set, referring it by that name.

Arguments

- `solutionSet` – (*Required*) The name of the hash index solution set.
- `sourceType` – (*Required*) The type of the source from which the bindings to store in the hash index are obtained:
  - `pipeline` – Spools the incoming solutions from the downstream operator in the operator pipeline into the hash index.
  - `binding set` – Spools the fixed binding set specified by the `sourceBindingSet` argument into the hash index.
- `sourceBindingSet` – (*Optional*) If the `sourceType` argument value is `binding set`, this argument specifies the static binding set to be spooled into the hash index.

`HashIndexJoin` operator

Joins the incoming solutions against the hash index solution set identified by the `solutionSet` argument.

Arguments

- `solutionSet` – (*Required*) Name of the solution set to join against. This must be a hash index that has been constructed in a prior step using the `HashIndexBuild` operator.
- `joinType` – (*Required*) The type of join to be performed:
  - `join` – A normal join, requiring an exact match between all shared variables.
  - `optional` – An `optional` join that uses the SPARQL `OPTIONAL` operator semantics.
  - `minus` – A `minus` operation retains a mapping for which no join partner exists, using the SPARQL `MINUS` operator semantics.

- `existence check` – Checks whether there is a join partner or not, and binds the `existenceCheckResultVar` variable to the result of this check.
- `constraints` – (*Optional*) Additional join constraints that are considered during the join. Joins that do not satisfy these constraints are discarded.
- `existenceCheckResultVar` – (*Optional*) Only used for joins where `joinType` equals `existence check` (see the `joinType` argument earlier).

`MergeJoin` operator

A merge join over multiple solution sets, as identified by the `solutionSets` argument.

Arguments

- `solutionSets` – (*Required*) The solution sets to join together.

`NamedSubquery` operator

Triggers evaluation of the subquery identified by the `subQuery` argument and spools the result into the solution set specified by the `solutionSet` argument. The incoming solutions for the operator are forwarded to the subquery and then to the next operator.

Arguments

- `subQuery` – (*Required*) Name of the subquery to evaluate. The subquery is rendered explicitly in the output.
- `solutionSet` – (*Required*) The name of the solution set in which to store the subquery result.

`PipelineJoin` operator

Receives as input the output of the previous operator and joins it against the tuple pattern defined by the `pattern` argument.

Arguments

- `pattern` – (*Required*) The pattern, which takes the form of a subject-predicate-object, and optionally -graph tuple that underlies the join. If `distinct` is specified for the pattern, the join only extracts distinct solutions from projection variables specified by the `projectionVars` argument, rather than all matching solutions.

- `inlineFilters` – (*Optional*) A set of filters to be applied to the variables in the pattern. The pattern is evaluated in conjunction with these filters.
- `joinType` – (*Required*) The type of join to be performed:
  - `join` – A normal join, requiring an exact match between all shared variables.
  - `optional` – An `optional` join that uses the SPARQL `OPTIONAL` operator semantics.
  - `minus` – A `minus` operation retains a mapping for which no join partner exists, using the SPARQL `MINUS` operator semantics.
  - `existence check` – Checks whether there is a join partner or not, and binds the `existenceCheckResultVar` variable to the result of this check.
- `constraints` – (*Optional*) Additional join constraints that are considered during the join. Joins that do not satisfy these constraints are discarded.
- `projectionVars` – (*Optional*) The projection variables. Used in combination with `distinct := true` to enforce the extraction of distinct projections over a specified set of variables.
- `cutoffLimit` – (*Optional*) A cutoff limit for the number of join partners extracted. Although there is no limit by default, you can set this to 1 when performing joins to implement `FILTER (NOT) EXISTS` clauses, where it is sufficient to prove or disprove that there is a join partner.

`PipelineCountJoin` operator

Variant of the `PipelineJoin`. Instead of joining, it just counts the matching join partners and binds the count to the variable specified by the `countVar` argument.

Arguments

- `countVar` – (*Required*) The variable to which the count result, namely the number of join partners, should be bound.
- `pattern` – (*Required*) The pattern, which takes the form of a subject-predicate-object, and optionally -graph tuple that underlies the join. If `distinct` is specified for the pattern, the join only extracts distinct solutions from projection variables specified by the `projectionVars` argument, rather than all matching solutions.
- `inlineFilters` – (*Optional*) A set of filters to be applied to the variables in the pattern. The pattern is evaluated in conjunction with these filters.
- `joinType` – (*Required*) The type of join to be performed:
  - `join` – A normal join, requiring an exact match between all shared variables.
  - `optional` – An `optional` join that uses the SPARQL `OPTIONAL` operator semantics.

  - `minus` – A `minus` operation retains a mapping for which no join partner exists, using the SPARQL `MINUS` operator semantics.
  - `existence check` – Checks whether there is a join partner or not, and binds the `existenceCheckResultVar` variable to the result of this check.
- `constraints` – (*Optional*) Additional join constraints that are considered during the join. Joins that do not satisfy these constraints are discarded.
- `projectionVars` – (*Optional*) The projection variables. Used in combination with `distinct := true` to enforce the extraction of distinct projections over a specified set of variables.
- `cutoffLimit` – (*Optional*) A cutoff limit for the number of join partners extracted. Although there is no limit by default, you can set this to 1 when performing joins to implement `FILTER (NOT) EXISTS` clauses, where it is sufficient to prove or disprove that there is a join partner.

`PipelinedHashIndexJoin` operator

This is an all-in-one build hash index and join operator. It takes a list of bindings, spools them into a hash index, and then joins the incoming solutions against the hash index.

Arguments

- `sourceType` – (*Required*) The type of the source from which the bindings to store in the hash index are obtained, one of:
  - `pipeline` – Causes `PipelinedHashIndexJoin` to spool the incoming solutions from the downstream operator in the operator pipeline into the hash index.
  - `binding set` – Causes `PipelinedHashIndexJoin` to spool the fixed binding set specified by the `sourceBindingSet` argument into the hash index.
- `sourceSubQuery` – (*Optional*) If the `sourceType` argument value is `pipeline`, this argument specifies the subquery that is evaluated and spooled into the hash index.
- `sourceBindingSet` – (*Optional*) If the `sourceType` argument value is `binding set`, this argument specifies the static binding set to be spooled into the hash index.
- `joinType` – (*Required*) The type of join to be performed:
  - `join` – A normal join, requiring an exact match between all shared variables.
  - `optional` – An `optional` join that uses the SPARQL `OPTIONAL` operator semantics.
  - `minus` – A `minus` operation retains a mapping for which no join partner exists, using the SPARQL `MINUS` operator semantics.

- `existence check` – Checks whether there is a join partner or not, and binds the `existenceCheckResultVar` variable to the result of this check.
- `existenceCheckResultVar` – (*Optional*) Only used for joins where `joinType` equals `existence check` (see the joinType argument above).

`Projection` operator

Projects over a subset of the variables. The number of solutions flowing in equals the number of solutions flowing out, but the shape of the solution differs, depending on the mode setting.

Modes

- `retain` – Retain in solutions only the variables that are specified by the `vars` argument.
- `drop` – Drop all the variables that are specified by the `vars` argument.

Arguments

- `vars` – (*Required*) The variables to retain or drop, depending on the mode setting.

`PropertyPath` operator

Enables recursive property paths such as `+` or `*`. Neptune implements a fixed-point iteration approach based on a template specified by the `iterationTemplate` argument. Known left-side or right-side variables are bound in the template for every fixed-point iteration, until no more new solutions can be found.

Arguments

- `iterationTemplate` – (*Required*) Name of the subquery template used to implement the fixed-point iteration.
- `leftTerm` – (*Required*) The term (variable or constant) on the left side of the property path.
- `rightTerm` – (*Required*) The term (variable or constant) on the right side of the property path.
- `lowerBound` – (*Required*) The lower bound for fixed-point iteration (either `0` for `*` queries, or `1` for `+` queries).

`TermResolution` operator

Translates internal string identifier values back to their corresponding external strings, or translates external strings to internal string identifier values, depending on the mode.

Modes

- `value2id` – Maps terms such as literals and URIs to corresponding internal ID values (encoding to internal values).
- `id2value` – Maps internal ID values to the corresponding terms such as literals and URIs (decoding of internal values).

Arguments

- `vars` – (*Required*) Specifies the variables whose strings or internal string IDs should be mapped.

`Slice` operator

Implements a slice over the incoming solution stream, using the semantics of SPARQL's `LIMIT` and `OFFSET` clauses.

Arguments

- `limit` – (*Optional*) A limit on the solutions to be forwarded.
- `offset` – (*Optional*) The offset at which solutions are evaluated for forwarding.

`SolutionInjection` operator

Receives no input. Statically injects solutions into the query plan and records them in the `solutions` argument.

Query plans always begin with this static injection. If static solutions to inject can be derived from the query itself by combining various sources of static bindings (for example, from `VALUES` or `BIND` clauses), then the `SolutionInjection` operator injects these derived static solutions. In the simplest case, these reflect bindings that are implied by an outer `VALUES` clause.

If no static solutions can be derived from the query, `SolutionInjection` injects the empty, so-called universal solution, which is expanded and multiplied throughout the query-evaluation process.

Arguments

- `solutions` – (*Required*) The sequence of solutions injected by the operator.

`Sort` operator

Sorts the solution set using specified sort conditions.

Arguments

- `sortOrder` – (*Required*) An ordered list of variables, each containing an `ASC` (ascending) or `DESC` (descending) identifier, used sequentially to sort the solution set.

`VariableAlignment` operator

Inspects solutions one by one, performing alignment on each one over two variables: a specified `sourceVar` and a specified `targetVar`.

If `sourceVar` and `targetVar` in a solution have the same value, the variables are considered aligned and the solution is forwarded, with the redundant `sourceVar` projected out.

If the variables bind to different values, the solution is filtered out entirely.

Arguments

- `sourceVar` – (*Required*) The source variable, to be compared to the target variable. If alignment succeeds in a solution, meaning that the two variables have the same value, the source variable is projected out.
- `targetVar` – (*Required*) The target variable, with which the source variable is compared. Is retained even when alignment succeeds.

Limitations of SPARQL `explain` in Neptune

The release of the Neptune SPARQL `explain` feature has the following limitations.

Neptune Currently Supports Explain Only in SPARQL SELECT Queries

For information about the evaluation process for other query forms, such as `ASK`, `CONSTRUCT`, `DESCRIBE`, and `SPARQL UPDATE` queries, you can transform these queries into a SELECT query. Then use `explain` to inspect the corresponding SELECT query instead.

For example, to obtain `explain` information about an `ASK WHERE {...}` query, run the corresponding `SELECT WHERE {...} LIMIT 1` query with `explain`.

Similarly, for a `CONSTRUCT {...} WHERE {...}` query, drop the `CONSTRUCT {...}` part and run a `SELECT` query with `explain` on the second `WHERE {...}` clause. Evaluating the second `WHERE` clause generally reveals the main challenges of processing the `CONSTRUCT` query, because solutions flowing out of the second `WHERE` into the `CONSTRUCT` template generally only require straightforward substitution.

Explain Operators May Change in Future Releases

The SPARQL `explain` operators and their parameters may change in future releases.

Explain Output May Change in Future Releases

For example, column headers could change, and more columns might be added to the tables.

SPARQL federated queries in Neptune using the `SERVICE` extension

Amazon Neptune fully supports the SPARQL federated query extension that uses the `SERVICE` keyword. (For more information, see SPARQL 1.1 Federated Query.)

The `SERVICE` keyword instructs the SPARQL query engine to execute a portion of the query against a remote SPARQL endpoint and compose the final query result. Only `READ` operations are possible. `WRITE` and `DELETE` operations are not supported. Neptune can only run federated queries against SPARQL endpoints that are accessible within its virtual private cloud (VPC). However, you can also use a reverse proxy in the VPC to make an external data source accessible within the VPC.

> **Note**
>
> When SPARQL `SERVICE` is used to federate a query to two or more Neptune clusters in the same VPC, the security groups must be configured to allow all those Neptune clusters to talk to each another.

> **Important**
>
> SPARQL 1.1 Federation makes service requests on your behalf when passing queries and parameters to external SPARQL endpoints. It is your responsibility to verify that

the external SPARQL endpoints satisfy your application's data handling and security requirements.

Example of a Neptune federated query

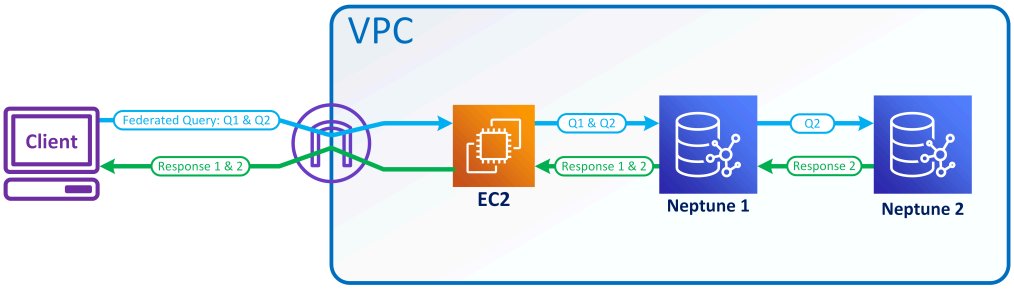
The following simple example shows how SPARQL federated queries work.

Suppose that a customer sends the following query to *Neptune-1* at `http://neptune-1:8182/sparql`.

```
SELECT * WHERE {
   ?person rdf:type foaf:Person .
   SERVICE <http://neptune-2:8182/sparql> {
       ?person foaf:knows ?friend .
    }
}
```

1. *Neptune-1* evaluates the first query pattern (*Q-1*) which is `?person rdf:type foaf:Person`, uses the results to resolve `?person` in *Q-2* (`?person foaf:knows ?friend`), and forwards the resulting pattern to *Neptune-2* at `http://neptune-2:8182/sparql`.
2. *Neptune-2* evaluates *Q-2* and sends the results back to *Neptune-1*.
3. *Neptune-1* joins the solutions for both patterns and sends the results back to the customer.

This flow is shown in the following diagram.



> **Note**
>
> "By default, the optimizer determines at what point in query execution that the `SERVICE` instruction is executed. You can override this placement using the joinOrder query hint.

Access control for federated queries in Neptune

Neptune uses AWS Identity and Access Management (IAM) for authentication and authorization. Access control for a federated query can involve more than one Neptune DB instance. These instances might have different requirements for access control. In certain circumstances, this can limit your ability to make a federated query.

Consider the simple example presented in the previous section. *Neptune-1* calls *Neptune-2* with the same credentials it was called with.

- If *Neptune-1* requires IAM authentication and authorization, but *Neptune-2* does not, all you need is appropriate IAM permissions for *Neptune-1* to make the federated query.
- If *Neptune-1* and *Neptune-2* both require IAM authentication and authorization, you need to attach IAM permissions for both databases to make the federated query. Both clusters must also be in the same AWS account and in the same region. Cross-region and/or cross-account federated query architectures are not currently supported.
- However, in the case where *Neptune-1* is not IAM-enabled but *Neptune-2* is, you can't make a federated query. The reason is that *Neptune-1* can't retrieve your IAM credentials and pass them on to *Neptune-2* to authorize the second part of the query.

Exporting data from a Neptune DB cluster

Neptune provides several ways to export data from your DB cluster, depending on the data model, format, and scale of export you need.

Native export to Amazon S3

Use the built-in Neptune export API to export RDF data directly to Amazon S3 in N-Triples or N-Quads format. This is the simplest approach for large-scale RDF exports because it requires no external infrastructure.

Exporting query results to Amazon S3

Export the results of a Gremlin traversal directly to Amazon S3 using the `call()` step. Use this when you need to export query results that are too large to return as a response, or when you want to store Gremlin query output for downstream processing.

External `neptune-export` tool

Use the open-source `neptune-export` tool – a separate application from Neptune that you deploy and operate yourself – to export property graph or RDF data in CSV and other formats. Can be run as a managed service (Neptune-Export service) or as a command-line tool. Use this when you need CSV output or formats not yet supported by native export.

For small amounts of data, you can also simply use the results of a query. For RDF data, the Graph Store Protocol (GSP) provides a simple way to export individual named graphs.

Topics

- Native export for RDF data
- Exporting Gremlin query results to Amazon S3
- External neptune-export tool

Native export for RDF data

The Neptune export API enables you to export data from your Neptune database to Amazon S3. You can export RDF data in N-Triples or N-Quads format.

How native export works

Native export runs on the writer instance of your Neptune cluster and writes exported data to Amazon S3 using multipart uploads. Because export uses the writer instance's compute resources, we strongly recommend running exports against a cloned cluster to avoid impacting production workloads. See Recommendations for more information.

Export throughput

Export throughput scales approximately linearly with instance size up to `r7i.16xlarge`. As a conservative planning estimate, expect approximately **50,000 statements per second per vCPU**.

Use this formula to estimate export duration:

```
export_seconds = total_statements / (vCPUs × 50,000)
```

Actual throughput depends on dataset characteristics including predicate cardinality, statement complexity, and cluster size.

Prerequisites

Before using the export API, you must:

- Have a Neptune database cluster running version 1.4.6.0 or later (version 1.4.8.0 or later for `exportFilter` support)
- Create an IAM role with permissions to access your S3 bucket
- Associate the IAM role with your Neptune cluster
- Configure an S3 VPC endpoint in your Neptune cluster's VPC

Recommendations

We strongly recommend running export operation on a cloned cluster without any read/write workloads to avoid impact on production performance.

For optimal price-to-performance ratio, we recommend using **16xlarge** instances for export operations. This instance type provides:

- Sufficient memory to handle large datasets without performance degradation

- Optimal CPU resources for concurrent export processing
- Best cost efficiency for export workloads

IAM permissions

The export feature involves two separate IAM roles:

- **Caller's IAM role** – The IAM principal (user or role) that sends requests to the export API endpoint. This role needs Neptune data-access permissions.
- **S3 access IAM role** – The role that Neptune assumes to write exported data to Amazon S3. You pass this role's ARN in the `iamRoleArn` parameter of the export request, and it must be associated with your Neptune cluster.

Caller permissions (Neptune data-access actions)

The IAM principal calling the export API must have the following Neptune data-access actions in its IAM policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowNeptuneExportActions",
      "Effect": "Allow",
      "Action": [
        "neptune-db:StartExportJob",
        "neptune-db:GetExportJobStatus",
        "neptune-db:ListExportJobs",
        "neptune-db:CancelExportJob"
      ],
      "Resource": "arn:aws:neptune-db:eusc-de-east-1:123456789012:cluster-resource-id/
*"
    }
  ]
}
```

For more information, see Using IAM data-access policy statements.

S3 access role permissions

The IAM role passed in the `iamRoleArn` request parameter must be associated with your Neptune cluster and must grant Neptune permission to write to the target S3 bucket. For steps on creating an IAM role and associating it with your cluster, see Create an IAM role to allow Neptune to access Amazon S3.

> **Note**
>
> The export API requires *write* permissions to S3, unlike the bulk loader which only requires read access. Use the following permissions policy instead of the `AmazonS3ReadOnlyAccess` managed policy described on that page.

Attach the following permissions policy to the S3 access role:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowS3WriteForNeptuneExport",
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket",
        "s3:GetObject",
        "s3:PutObject",
        "s3:AbortMultipartUpload",
        "s3:GetBucketPublicAccessBlock"
      ],
      "Resource": [
        "arn:aws:s3:::amzn-s3-demo-bucket",
        "arn:aws:s3:::amzn-s3-demo-bucket/*"
      ]
    }
  ]
}
```

Optional KMS permissions

If you specify a `kmsKeyIdentifier` in the export request, add the following permissions to the S3 access role:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowKMSForNeptuneExport",
      "Effect": "Allow",
      "Action": [
        "kms:Decrypt",
        "kms:Encrypt",
        "kms:GenerateDataKey"
      ],
      "Resource": "arn:aws:kms:eusc-de-east-1:123456789012:key/key-id"
    }
  ]
}
```

Export endpoint

To export data, you send HTTP requests to the `https://your-neptune-endpoint:port/export` endpoint.

Export request syntax

```
POST https://your-neptune-endpoint:port/export
```

Request headers

- `Content-Type: application/json`

Request body

```
{
  "destination": "string",
  "format": "string",
  "iamRoleArn": "string",
  "region": "string",
  "compression": "string",
  "kmsKeyIdentifier": "string",
  "exportFilter": {
    "namedGraphUris": ["string"]
```

```
  }
}
```

Request parameters

destination (string)

Required. The S3 URI where the exported data will be stored. Must be in the format `s3://bucket-name/optional-prefix/`.

format (string)

Required. The format for the exported data. Valid values:

- `ntriples` – Export data in N-Triples format
- `nquads` – Export data in N-Quads format

iamRoleArn (string)

Required. The Amazon Resource Name (ARN) of the IAM role that Neptune assumes to access the S3 bucket.

region (string)

Required. The AWS Region of the S3 bucket. Must be the same region as your Neptune cluster.

compression (string)

Optional. Format of compression for exported files. Valid values:

- `gz` - Compress files in .gz format

kmsKeyIdentifier (string)

Optional. The ARN of the AWS KMS key to use for encrypting the exported data.

exportFilter (object)

Optional. Filters to selectively export a subset of RDF data. Available in engine version 1.4.8.0 and later.

namedGraphUris (array of strings)

Named graph URIs to export (maximum 100). Only data in the specified graphs is exported. If a specified graph is empty, the export succeeds with an empty result. Invalid URIs fail with `InvalidParameterException`.

Response syntax

```
{
  "status": "string",
  "payload": {
    "exportId": "string"
  }
}
```

status (string)

The HTTP status of the request.

exportId (string)

A unique identifier for the export task.

Export status endpoint

To check the status of an export task, you send an HTTP GET request to the export endpoint with the export ID.

```
GET https://your-neptune-endpoint:port/export?exportId=export-id
```

Request parameters

exportId (string)

Required. The unique identifier of the export task.

Response syntax

```
{
  "status": "string",
  "payload": {
    "exportId": "string",
    "destination": "string",
    "status": "string",
    "statusReason": "string",
```

```
    "format": "string",
    "iamRoleArn": "string",
    "kmsKeyIdentifier": "string",
    "exportFilter": {
      "namedGraphUris": ["string"]
    },
    "exportTaskDetails": {
      "timeElapsedSeconds": number,
      "startTime": number,
      "numRecordsWritten": number,
      "progressPercentage": number
    }
  }
}
```

exportId (string)

The unique identifier of the export task.

destination (string)

The S3 URI where the data is being exported.

status (string)

The current status of the export task. Valid values:

- `EXPORT_NOT_STARTED` – Export has been queued but not started
- `EXPORT_IN_PROGRESS` – Export is currently running
- `EXPORT_COMPLETED` – Export completed successfully
- `EXPORT_CANCELLING` – Export is being cancelled
- `EXPORT_CANCELLED_BY_USER` – Export was cancelled by user
- `EXPORT_S3_ERROR` – Export failed due to S3 access error
- `EXPORT_FAILED` – Export failed due to other error

statusReason (string)

Additional information about the export status.

format (string)

The format of the exported data.

iamRoleArn (string)

The ARN of the IAM role used for S3 access.

kmsKeyIdentifier (string)

The ARN of the KMS key used for encryption, if specified.

exportFilter (object)

The export filter that was applied, if one was specified in the request.

- **namedGraphUris** (array of strings) – The named graph URIs used to filter the export.

exportTaskDetails (object)

Details about the export task progress:

- **timeElapsedSeconds** (number) – Time elapsed since export started
- **startTime** (number) – Epoch time when export started
- **numRecordsWritten** (number) – Number of records written to S3
- **progressPercentage** (number) – Percentage of export completed

List exports endpoint

To list all export tasks, you send an HTTP GET request to the export endpoint.

```
GET https://your-neptune-endpoint:port/export
```

Response syntax

```
{
  "status": "string",
  "payload": [
    "string"
  ]
}
```

payload (array)

An array of export IDs for all export tasks.

Cancel export endpoint

To cancel an export task, you send an HTTP DELETE request to the export endpoint with the export ID.

```
DELETE https://your-neptune-endpoint:port/export?exportId=export-id
```

Request parameters

exportId (string)

Required. The unique identifier of the export task to cancel.

Response syntax

```
{
  "status": "string",
  "payload": {
    "message": "string"
  }
}
```

Export output format

S3 directory structure

Exported data is organized in your S3 bucket as follows:

```
s3://your-bucket/export-id/
### data/
#   ### part-00000.nt
#   ### part-00001.nt
#   ### ...
### export_status.json
```

data/ directory

Contains the exported graph data files in the specified format.

export_status.json file

Contains metadata about the export operation.

Error responses

Common error codes

BadRequestException

The request contains invalid parameters or an export is already in progress.

AccessDeniedException

The IAM role lacks required permissions for S3 or KMS access.

Example error response

```
{
  "code": "BadRequestException",
  "requestId": "request-id",
  "message": "Export already in progress with ID 'existing-id'. Please wait or cancel
 it first.",
  "detailedMessage": "Detailed error description"
}
```

Examples

Start an export

```
curl -X POST https://your-cluster-endpoint:8182/export \
  -H "Content-Type: application/json" \
  -d '{
    "destination": "s3://my-bucket/exports/",
    "format": "ntriples",
    "iamRoleArn": "arn:aws:iam::123456789012:role/neptune-export-role",
    "region": "us-west-2"
  }'
```

Start a filtered export (named graphs)

```
curl -X POST https://your-cluster-endpoint:8182/export \
  -H "Content-Type: application/json" \
  -d '{
    "destination": "s3://my-bucket/exports/",
    "format": "nquads",
    "iamRoleArn": "arn:aws:iam::123456789012:role/neptune-export-role",
    "region": "us-west-2",
    "exportFilter": {
      "namedGraphUris": [
        "http://example.com/graph1",
        "http://example.com/graph2"
      ]
    }
  }'
```

Check export status

```
curl -X GET "https://your-cluster-endpoint:8182/export?exportId=<id>"
```

Cancel an export

```
curl -X DELETE "https://your-cluster-endpoint:8182/export?exportId=<id>"
```

Limitations

Native export has the following limitations:

- **Instance support** – Native export is not supported on Neptune Serverless instances or on read replicas. Export always runs on the writer instance of a provisioned cluster.
- **One export per cluster** – Only one export can be active on a cluster at a time. If you submit a new export request while another export is in progress, the request fails.
- **No auto-resume** – If the engine restarts during an export, the export fails and must be restarted from the beginning. Export progress is not preserved across engine restarts.
- **Status availability after engine events** – If the engine crashes, you can't retrieve the export status through the status API.

- **Consistency during writes** – If your cluster is servicing write traffic during an export, the exported data may reflect a partial or inconsistent view of the graph. To guarantee a consistent export, run the export against a cloned cluster.

Exporting Gremlin query results to Amazon S3

Starting in engine release 1.4.3.0, Amazon Neptune supports exporting Gremlin query results directly to Amazon S3. This feature allows you to handle large query results efficiently by exporting them to an Amazon S3 bucket instead of returning them as a query response.

To export query results to Amazon S3, use the `call()` step with the `neptune.query.exportToS3` service name as the final step in your Gremlin query. Terminal step in Tinkerpop drivers using Bytecode can be added after the `call()` step. The export parameters must be provided as string values.

> **Note**
>
> The query with the `call()` step having `neptune.query.exportToS3` will fail if not used as the final step. The Gremlin clients using bytecode can use terminal steps. See Gremlin best practices in the Amazon Neptune documentation for more information.

```
g.V()
  ...
  .call('neptune.query.exportToS3', [
    'destination': 's3://your-bucket/path/result.json',
    'format': 'GraphSONv3',
    'kmskeyArn': 'optional-kms-key-arn'
  ])
```

Parameters

- `destination`: required - The Amazon S3 URI where results will be written.
- `format`: required - The output format, currently only supports 'GraphSONv3'.
- `keyArn`: optional - The ARN of a AWS KMS key for Amazon S3 server-side encryption.

Examples

Example query

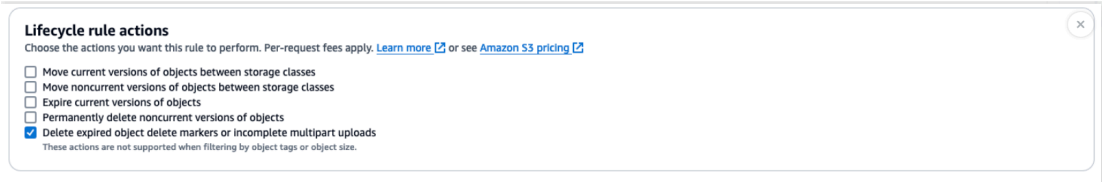
```
g.V().
    hasLabel('Comment').
    valueMap().
    call('neptune.query.exportToS3', [
    'destination': 's3://your-bucket/path/result.json',
    'format': 'GraphSONv3',
    'keyArn': 'optional-kms-key-arn'
  ])
```

Example query response

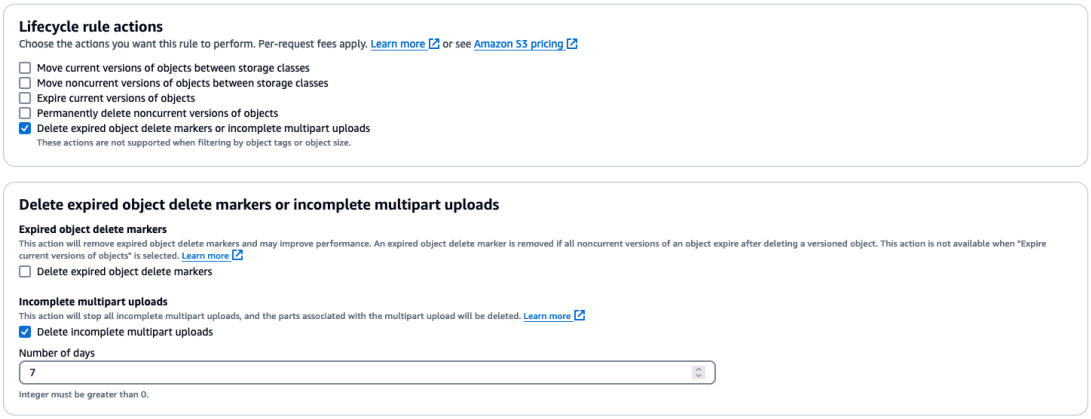
```
{
    "destination":"s3://your-bucket/path/result.json,
    "exportedResults": 100,
    "exportedBytes": 102400
}
```

Prerequisites

- Your Neptune DB instance must have access to Amazon S3 through a VPC endpoint of type gateway.
- To use custom AWS KMS encryption in the query, an Interface-type VPC endpoint for AWS KMS is required to allow Neptune to communicate with AWS KMS.
- You must enable IAM auth on Neptune, and have appropriate IAM permissions to write to the target Amazon S3 bucket. Not having this will cause a 400 bad request error "Cluster must have IAM authentication enabled for S3 Export".
- The target Amazon S3 bucket:
  - The target Amazon S3 bucket must not be public. `Block public access` must be enabled.
  - The target Amazon S3 destination must be empty.
  - The target Amazon S3 bucket must have a lifecycle rule on `Delete expired object delete markers or incomplete multipart uploads` with `Delete incomplete multipart uploads`. See Amazon S3 lifecycle management update - support for multipart uploads and delete markers for more information.



- The target Amazon S3 bucket must have the a lifecycle rule on `Delete expired object delete markers or incomplete multipart uploads` with `Delete incomplete multipart uploads` set to a value higher than query evaluation will take (e.g., 7 days). This is required for deleting incomplete uploads (which are not directly visible but would incur costs) in case they cannot be completed or aborted by Neptune (e.g., due to instance/engine failures). See Amazon S3 lifecycle management update - support for multipart uploads and delete markers for more information.



Important considerations

- The export step must be the last step in your Gremlin query.
- If an object already exists at the specified Amazon S3 location, the query will fail.
- Maximum query execution time for export queries is limited to 11 hours and 50 minutes. This feature uses Forward access sessions. It is currently limited to 11 hours and 50 minutes to avoid token expiry issues.

> **Note**
>
> The export query still honors the query timeout. For large exports, you should use an appropriate query timeout.

- All new object uploads to Amazon S3 are automatically encrypted.

- To avoid storage costs from incomplete multipart uploads in the event of errors or crashes, we recommend setting up a lifecycle rule with `Delete incomplete multipart uploads` on your Amazon S3 bucket.

Response format

Rather than returning the query results directly, the query returns metadata about the export operation, including status and export details. The query results in Amazon S3 will be in GraphSONv3 format.

```
{
  "data": {
    "@type": "g:List",
    "@value": [
      {
        "@type": "g:Map",
        "@value": [
          "browserUsed",
          {
            "@type": "g:List",
            "@value": [
              "Safari"
            ]
          },
          "length",
          {
            "@type": "g:List",
            "@value": [
              {
                "@type": "g:Int32",
                "@value": 7
              }
            ]
          },
          "locationIP",
          {
            "@type": "g:List",
            "@value": [
              "192.0.2.0/24"
            ]
          },
          "creationDate",
```

```
          {
            "@type": "g:List",
            "@value": [
              {
                "@type": "g:Date",
                "@value": 1348341961000
              }
            ]
          },
          "content",
          {
            "@type": "g:List",
            "@value": [
              "no way!"
            ]
          }
        ]
      },
      {
        "@type": "g:Map",
        "@value": [
          "browserUsed",
          {
            "@type": "g:List",
            "@value": [
              "Firefox"
            ]
          },
          "length",
          {
            "@type": "g:List",
            "@value": [
              {
                "@type": "g:Int32",
                "@value": 2
              }
            ]
          },
          "locationIP",
          {
            "@type": "g:List",
            "@value": [
              "203.0.113.0/24"
            ]
```

```
          },
          "creationDate",
          {
            "@type": "g:List",
            "@value": [
              {
                "@type": "g:Date",
                "@value": 1348352960000
              }
            ]
          },
          "content",
          {
            "@type": "g:List",
            "@value": [
              "ok"
            ]
          }
        ]
      },


      ...


    ]
  }
}
```

Security

- All data transferred to Amazon S3 is encrypted in transit using SSL.
- You can specify a AWS KMS key for server-side encryption of the exported data. Amazon S3 encrypts new data by default. If the bucket is configured to use a specific AWS KMS key, then that key is used.
- Neptune verifies that the target bucket is not public before starting the export.
- Cross-account and cross-region exports are not supported.

Error handling

- The target Amazon S3 bucket is public.

- The specified object already exists.
- You don't have sufficient permissions to write to the Amazon S3 bucket.
- The query execution exceeds the maximum time limit.

Best practices

- Use Amazon S3 bucket lifecycle rules to clean up incomplete multipart uploads.
- Monitor your export operations using Neptune logs and metrics. You can check the Gremlin status endpoint to see if a query is currently running. As long as the client has not received a response, the query will be assumed to be running.

Granting access for Gremlin Amazon S3 export feature

Required IAM policies

1. **Neptune query read access**

```
{
  "Sid": "NeptuneQueryRead",
  "Effect": "Allow",
  "Action": ["neptune-db:Read*"],
  "Resource": "arn:aws:neptune-db:us-east-1:123456789012:cluster-ABCD12/*"
}
```

   Why it's needed: This permission allows reading data from Neptune databases, which is necessary to execute the Gremlin queries that will be exported. The previous example allows read queries. For a read/write query, write/delete permissions are required.

2. **Amazon S3 export permissions**

```
{
  "Sid": "NeptuneS3Export",
  "Effect": "Allow",
  "Action": [
    "s3:ListBucket",
    "s3:PutObject",
    "s3:AbortMultipartUpload",
    "s3:GetBucketPublicAccessBlock"
  ],
```

```
  "Resource": "arn:aws:s3:::neptune-export-bucket/*"
}
```

Why each permission is needed:

- `s3:ListBucket`: Required to verify bucket existence and list contents.
- `s3:PutObject`: Required to write the exported data to Amazon S3.
- `s3:AbortMultipartUpload`: Required to clean up incomplete multipart uploads if the export fails.
- `s3:GetBucketPublicAccessBlock`: Required as a security measure to verify that the bucket is not public before exporting data.

3. **AWS KMS permissios** - optional. Only required if using custom AWS KMS encryption.

```
{
  "Sid": "NeptuneS3ExportKMS",
  "Effect": "Allow",
  "Action": [
    "kms:Decrypt",
    "kms:GenerateDataKey",
    "kms:DescribeKey"
  ],
  "Resource": "arn:aws:kms:<REGION>:<AWS_ACCOUNT_ID>:key/mrk-48971c37"
    "Condition": {
    "StringEquals": {
      "kms:ViaService": [
        "s3.<REGION>.amazonaws.com",
        "rds.<REGION>.amazonaws.com"
      ]
    }
  }
}
```

Why each permission is needed:

- `kms:Decrypt`: Required to decrypt the AWS KMS key for data encryption.
- `kms:GenerateDataKey`: Required to generate data keys for encrypting the exported data.
- `kms:DescribeKey`: Required to verify and retrieve information about the AWS KMS key.
- `kms:ViaService`: Increases security by enforcing that the key is not usable by this role for any other AWS service.

Important prerequisites

- **IAM authentication:** Must be enabled on the Neptune cluster to enforce these permissions.
- **VPC endpoint:**
  - A Gateway-type VPC endpoint for Amazon S3 is required to allow Neptune to communicate with Amazon S3.
  - To use custom AWS KMS encryption in the query, an Interface-type VPC endpoint for AWS KMS is required to allow Neptune to communicate with AWS KMS.
- **Amazon S3 bucket configuration:**
  - Must not be public.
  - Should have a lifecycle rule to clean up incomplete multipart uploads.
  - Will automatically encrypt new objects.

These permissions and prerequisites ensure secure and reliable export of Gremlin query results while maintaining proper access controls and data protection measures.

External `neptune-export` tool

The open-source `neptune-export` tool is a separate application from Neptune. Unlike native export (which runs within the Neptune engine), `neptune-export` is external tooling that you deploy and operate yourself. Use it when you need output formats not yet supported by native export, such as CSV or property graph data.

You can use `neptune-export` in two different ways:

- **As the Neptune-Export service**. When you export data from Neptune using the Neptune-Export service, you trigger and monitor export jobs through a REST API.
- **As the `neptune-export` Java command-line utility**. To use this command-line tool to export Neptune data, you have to run it in an environment where your Neptune DB cluster is accessible.

Both the Neptune-Export service and the `neptune-export` command line tool publish data to Amazon Simple Storage Service (Amazon S3), encrypted using Amazon S3 server-side encryption (`SSE-S3`).

> **Note**
>
> It is a best practice to enable access logging on all Amazon S3 buckets, to let you audit all access to those buckets.

If you try to export data from a Neptune DB cluster whose data is changing while the export is happening, the consistency of the exported data is not guaranteed. That is, if your cluster is servicing write traffic while an export job is in progress, there may be inconsistencies in the exported data. This is true whether you export from the primary instance in the cluster or from one or more read replicas.

To guarantee that exported data is consistent, it is best to export from a clone of your DB cluster. This both provides the export tool with a static version of your data and ensures that the export job doesn't slow down queries in your original DB cluster.

To make this easier, you can indicate that you want to clone the source DB cluster when you trigger an export job. If you do, the export process automatically creates the clone, uses it for the export, and then deletes it when the export is finished.

Topics

- Using the Neptune-Export service to export Neptune data
- Using the neptune-export command-line tool to export data from Neptune
- Files exported by Neptune-Export and neptune-export
- Parameters used to control the Neptune export process
- Troubleshooting the Neptune export process

Using the Neptune-Export service to export Neptune data

You can use the following steps to export data from your Neptune DB cluster to Amazon S3 using the Neptune-Export service:

Installing the Neptune-Export service

Use an AWS CloudFormation template to create the stack:

To install the Neptune-Export service

1. Launch the CloudFormation stack on the CloudFormation console by choosing one of the **Launch Stack** buttons in the following table:

Region	View	View in Designer	Launch
US East (N. Virginia)	View	View in Designer	Launch Stack
US East (Ohio)	View	View in Designer	Launch Stack
US West (N. California)	View	View in Designer	Launch Stack
US West (Oregon)	View	View in Designer	Launch Stack
Canada (Central)	View	View in Designer	Launch Stack
South America (São Paulo)	View	View in Designer	Launch Stack
Europe (Stockholm)	View	View in Designer	Launch Stack
Europe (Ireland)	View	View in Designer	Launch Stack
Europe (London)	View	View in Designer	Launch Stack
Europe (Paris)	View	View in Designer	Launch Stack
Europe (Frankfurt)	View	View in Designer	Launch Stack
Middle East (Bahrain)	View	View in Designer	Launch Stack

Region	View	View in Designer	Launch
Middle East (UAE)	View	View in Designer	Launch Stack
Israel (Tel Aviv)	View	View in Designer	Launch Stack
Africa (Cape Town)	View	View in Designer	Launch Stack
Asia Pacific (Hong Kong)	View	View in Designer	Launch Stack
Asia Pacific (Tokyo)	View	View in Designer	Launch Stack
Asia Pacific (Seoul)	View	View in Designer	Launch Stack
Asia Pacific (Singapore)	View	View in Designer	Launch Stack
Asia Pacific (Sydney)	View	View in Designer	Launch Stack
Asia Pacific (Mumbai)	View	View in Designer	Launch Stack
China (Beijing)	View	View in Designer	Launch Stack
China (Ningxia)	View	View in Designer	Launch Stack
AWS GovCloud (US-West)	View	View in Designer	Launch Stack
AWS GovCloud (US-East)	View	View in Designer	Launch Stack

2. On the **Select Template** page, choose **Next**.
3. On the **Specify Details** page, the template, set the following parameters:

- **`VPC`** – The easiest way to set up the Neptune-Export service is to install it in the same Amazon VPC as your Neptune database. If you want to install it in a separate VPC you can use VPC peering to establish connectivity between the Neptune DB cluster's VPC and the Neptune-Export service VPC.
- **`Subnet1`** – The Neptune-Export service must be installed in a subnet in your VPC that allows outbound IPv4 HTTPS traffic from the subnet to the internet. This is so that the Neptune-Export service can call the AWS Batch API to create and run an export job.

  If you created your Neptune cluster using the CloudFormation template on the Create Neptune cluster page in the Neptune documentation, you can use the `PrivateSubnet1` and `PrivateSubnet2` outputs from that stack to populate this and the next parameter.
- **`Subnet2`** – A second subnet in the VPC that allows outbound IPv4 HTTPS traffic from the subnet to the internet.
- **`EnableIAM`** – Set this to `true` to secure the Neptune-Endpoint API using AWS Identity and Access Management (IAM). We recommend that you do so.

  If you do enable IAM authentication, you must `Sigv4` sign all HTTPS requests to the endpoint. You can use a tool such as awscurl to sign requests on your behalf.
- **`VPCOnly`** – Setting this to `true` makes the export endpoint VPC-only, so that you can only access it from within the VPC where the Neptune-Export service is installed. This restricts the Neptune-Export API to being used only from within that VPC.

  We recommend that you set `VPCOnly` to `true`.
- **`NumOfFilesULimit`** – Specify a value between 10,000 and 1,000,000 for `nofile` in the `ulimits` container property. The default is 10,000, and we recommend keeping the default unless your graph contains a large number of unique labels.
- **`PrivateDnsEnabled`** (Boolean) – Indicates whether to associate a private hosted zone with the specified VPC or not. The default value is `true`.

  When a VPC endpoint is created with this flag enabled, all API Gateway traffic is routed through the VPC endpoint, and public API Gateway endpoint calls becomes disabled. If you set `PrivateDnsEnabled` to `false`, the public API Gateway endpoint is enabled, but the Neptune export service cannot be connected through the private DNS endpoint. You can then use a public DNS endpoint for the VPC endpoint to call the export service, as detailed here.

- **`NeptuneExportVersion`** – Specify the version of the Neptune Export Utility to be used. All versions greater than or equal to `v1.1.11` are supported. A version of `v2.latest` may be used to automatically receive minor updates. The full list of available versions, as well as patch notes can be found in the open source GitHub releases.

4. Choose **Next**.
5. On the **Options** page, choose **Next**.
6. On the **Review** page, select the first check box to acknowledge that CloudFormation will create IAM resources. Select the second check box to acknowledge `CAPABILITY_AUTO_EXPAND` for the new stack.

   > **Note**
   >
   > `CAPABILITY_AUTO_EXPAND` explicitly acknowledges that macros will be expanded when creating the stack, without prior review. Users often create a change set from a processed template so that the changes made by macros can be reviewed before actually creating the stack. For more information, see the CloudFormation CreateStack API.

   Then choose **Create**.

Enable access to Neptune from Neptune-Export

After the Neptune-Export installation has completed, update your Neptune VPC security group to allow access from Neptune-Export. When the Neptune-Export CloudFormation stack has been created, the **Outputs** tab includes a `NeptuneExportSecurityGroup` ID. Update your Neptune VPC security group to allow access from this Neptune-Export security group.

Enable access to the Neptune-Export endpoint from a VPC-based EC2 instance

If you make your Neptune-Export endpoint VPC-only, you can only access it from within the VPC in which the Neptune-Export service is installed. To allow connectivity from an Amazon EC2 instance in the VPC from which you can make Neptune-Export API calls, attach the `NeptuneExportSecurityGroup` created by the CloudFormation stack to that Amazon EC2 instance.

Run a Neptune-Export job using the Neptune-Export API

The **Outputs** tab of the CloudFormation stack also includes the `NeptuneExportApiUri`. Use this URI whenever you send a request to the Neptune-Export endpoint.

Run an export job

- Be sure that the user or role under which the export runs has been granted `execute-api:Invoke` permission.
- If you set the `EnableIAM` parameter to `true` in the CloudFormation stack when you installed Neptune-Export, you need to `Sigv4` sign all requests to the Neptune-Export API. We recommend using awscurl to make requests to the API. All the examples here assume that IAM auth is enabled.
- If you set the `VPCOnly` parameter to `true` in the CloudFormation stack when you installed Neptune-Export, you must call the Neptune-Export API from within the VPC, typically from an Amazon EC2 instance located in the VPC.

To start exporting data, send a request to the `NeptuneExportApiUri` endpoint with `command` and `outputS3Path` request parameters and an `endpoint` export parameter.

The following is an example of a request that exports property-graph data from Neptune and publishes it to Amazon S3:

```
curl \
  (your NeptuneExportApiUri) \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "command": "export-pg",
        "outputS3Path": "s3://(your Amazon S3 bucket)/neptune-export",
        "params": { "endpoint": "(your Neptune endpoint DNS name)" }
      }'
```

Similarly, here is an example of a request that exports RDF data from Neptune to Amazon S3:

```
curl \
  (your NeptuneExportApiUri) \
  -X POST \
  -H 'Content-Type: application/json' \
```

```
  -d '{
        "command": "export-rdf",
        "outputS3Path": "s3:////(your Amazon S3 bucket)/neptune-export",
        "params": { "endpoint": "(your Neptune endpoint DNS name)" }
      }'
```

If you omit the `command` request parameter, by default Neptune-Export attempts to export property-graph data from Neptune.

If the previous command ran successfully, the output would look like this:

```
{
  "jobName": "neptune-export-abc12345-1589808577790",
  "jobId": "c86258f7-a9c9-4f8c-8f4c-bbfe76d51c8f"
}
```

Monitor the export job you just started

To monitor a running job, append its jobID to your `NeptuneExportApiUri`, something like this:

```
curl \
  (your NeptuneExportApiUri)/(the job ID)
```

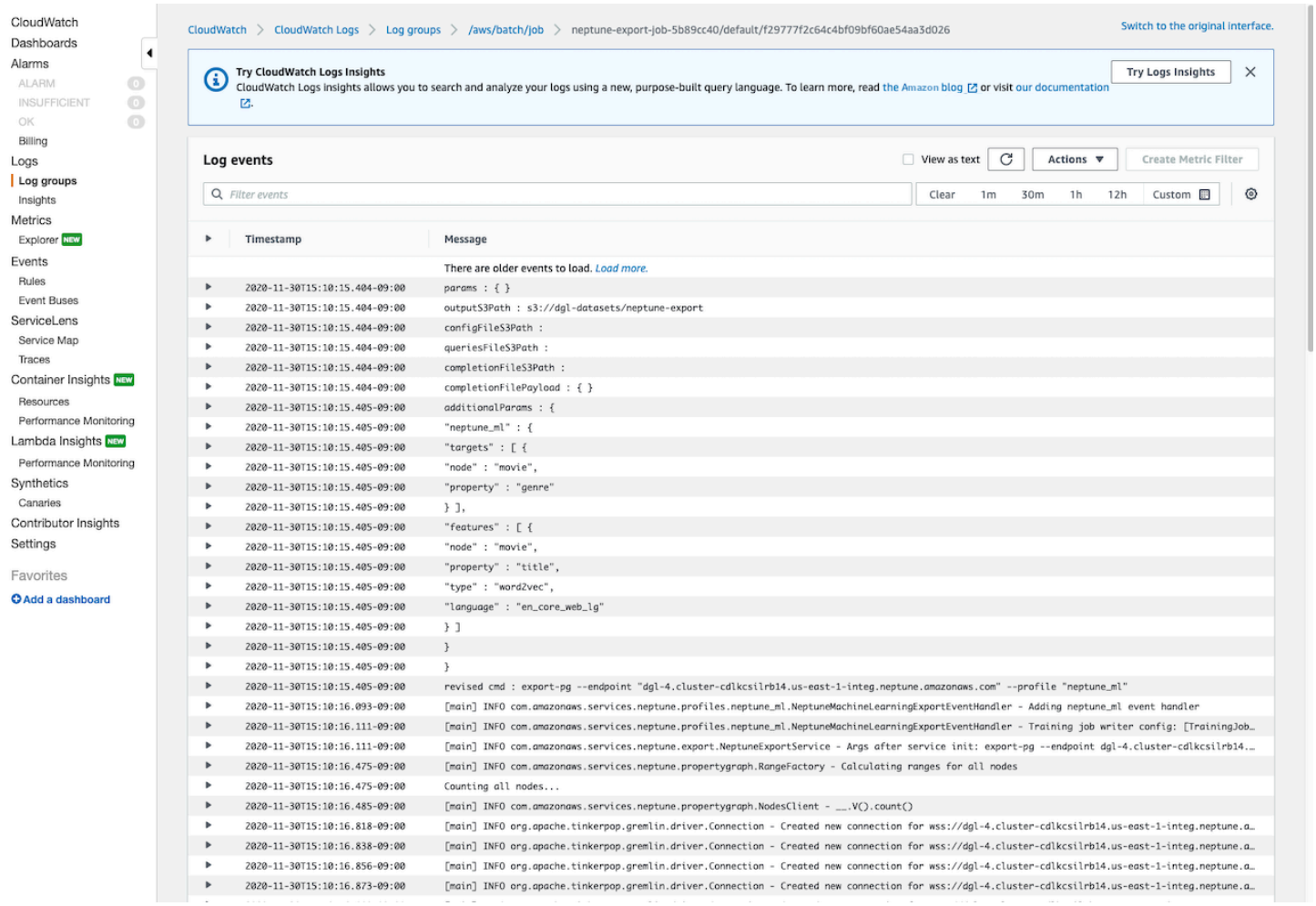
If the service had not yet started the export job, the response would look like this:

```
{
  "jobId": "c86258f7-a9c9-4f8c-8f4c-bbfe76d51c8f",
  "status": "pending"
}
```

When you repeat the command after the export job has started, the response would look something like this:

```
{
  "jobId": "c86258f7-a9c9-4f8c-8f4c-bbfe76d51c8f",
  "status": "running",
  "logs": "https://us-east-1.console.aws.amazon.com/cloudwatch/home?..."
}
```

If you open the logs in CloudWatch Logs using the URI provided by the status call, you can then monitor the progress of the export in detail:



Cancel a running export job

To cancel a running export job using the AWS Management Console

1. Open the AWS Batch console at https://eusc-de-east-1.console.amazonaws-eusc.eu/batch/.
2. Choose **Jobs**.
3. Locate the running job that you want to cancel, based on its `jobID`.
4. Select **Cancel job**.

To cancel a running export job using the Neptune export API:

Send an `HTTP DELETE` request to the `NeptuneExportApiUri` with the `jobID` appended, like this:

```
curl -X DELETE \
```

```
(your NeptuneExportApiUri)/(the job ID)
```

Using the `neptune-export` command-line tool to export data from Neptune

You can use the following steps to export data from your Neptune DB cluster to Amazon S3 using the `neptune-export` command-line utility:

Prerequisites for using the `neptune-export` command-line utility

Before you start

- **Have version 8 of the JDK** – You need version 8 of the Java SE Development Kit (JDK) installed.
- **Download the neptune-export utility** – Download and install the neptune-export.jar file.
- **Make sure `neptune-export` has access to your Neptune VPC** – Run neptune-export from a location where it can access the VPC where your Neptune DB cluster is located.

  For example, you can run it on an Amazon EC2 instance within the Neptune VPC, or in a separate VPC that is peered with the Neptune VPC, or on a separate bastion host.
- **Make sure the VPC security groups grant access to `neptune-export`** – Check that the VPC security group(s) attached to the Neptune VPC allow access to your DB cluster from the IP address or security group associated with the `neptune-export` environment.
- **Set up the necessary IAM permissions** – If your database has AWS Identity and Access Management (IAM) database authentication enabled, make sure that the role under which `neptune-export` runs is associated with an IAM policy that allows connections to Neptune. For information about Neptune policies, see Using IAM policies.

  If you want to use the `clusterId` export parameter in your query requests, the role under which `neptune-export` runs requires the following IAM permissions:

  - `rds:DescribeDBClusters`
  - `rds:DescribeDBInstances`
  - `rds:ListTagsForResource`

  If you want to export from a cloned cluster, the role under which `neptune-export` runs requires the following IAM permissions:

  - `rds:AddTagsToResource`

- `rds:DescribeDBClusters`
- `rds:DescribeDBInstances`
- `rds:ListTagsForResource`
- `rds:DescribeDBClusterParameters`
- `rds:DescribeDBParameters`
- `rds:ModifyDBParameterGroup`
- `rds:ModifyDBClusterParameterGroup`
- `rds:RestoreDBClusterToPointInTime`
- `rds:DeleteDBInstance`
- `rds:DeleteDBClusterParameterGroup`
- `rds:DeleteDBParameterGroup`
- `rds:DeleteDBCluster`
- `rds:CreateDBInstance`
- `rds:CreateDBClusterParameterGroup`
- `rds:CreateDBParameterGroup`

  To publish the exported data to Amazon S3, the role under which `neptune-export` runs requires the following IAM permissions for the Amazon S3 location(s):

  - `s3:PutObject`
  - `s3:PutObjectTagging`
  - `s3:GetObject`

- **Set the `SERVICE_REGION` environment variable** – Set the `SERVICE_REGION` environment variable to identify the Region where your DB cluster is located (see Connecting to Neptune for a list of Region identifiers).

Running the `neptune-export` utility to initiate an export operation

Use the following command to run neptune-export from the command line and start an export operation:

```
java -jar neptune-export.jar nesvc \
  --root-path (path to a local directory) \
  --json (the JSON file that defines the export)
```

The command has two parameters:

Parameters for neptune-export when starting an export

- **`--root-path`** – Path to a local directory where export files are written before being published to Amazon S3.
- **`--json`** – A JSON object that defines the export.

Example commands using the `neptune-export` command line utility

To export property-graph data directly from your source DB cluster:

```
java -jar neptune-export.jar nesvc \
  --root-path /home/ec2-user/neptune-export \
  --json '{
            "command": "export-pg",
            "outputS3Path" : "s3:////(your Amazon S3 bucket)/neptune-export",
            "params": {
              "endpoint" : "(your neptune DB cluster endpoint)"
            }
          }'
```

To export RDF data directly from your source DB cluster:

```
java -jar neptune-export.jar nesvc \
  --root-path /home/ec2-user/neptune-export \
  --json '{
            "command": "export-rdf",
            "outputS3Path" : "s3:////(your Amazon S3 bucket)/neptune-export",
            "params": {
              "endpoint" : "(your neptune DB cluster endpoint)"
            }
          }'
```

If you omit the `command` request parameter, the `neptune-export` utility exports property-graph data from Neptune by default.

To export from a clone of your DB cluster:

```
java -jar neptune-export.jar nesvc \
  --root-path /home/ec2-user/neptune-export \
```

```
  --json '{
            "command": "export-pg",
            "outputS3Path" : "s3://(your Amazon S3 bucket)/neptune-export",
            "params": {
              "endpoint" : "(your neptune DB cluster endpoint)",
              "cloneCluster" : true
            }
          }'
```

To export from your DB cluster using IAM authentication:

```
java -jar neptune-export.jar nesvc \
  --root-path /home/ec2-user/neptune-export \
  --json '{
            "command": "export-pg",
            "outputS3Path" : "s3://(your Amazon S3 bucket)/neptune-export",
            "params": {
              "endpoint" : "(your neptune DB cluster endpoint)"
              "useIamAuth" : true
            }
          }'
```

Files exported by Neptune-Export and `neptune-export`

When an export is complete, the export files are published to the Amazon S3 location you have specified. All files published to Amazon S3 are encrypted using Amazon S3 server-side encryption (`SSE-S3`). The folders and files published to Amazon S3 vary depending on whether you are exporting property-graph or RDF data. If you open the Amazon Amazon S3 location where the files are published, you see the following content:

Locations of exported files in Amazon S3

- **`nodes/`** – This folder contains node data files in either a comma-separated value (CSV) or a JSON format.

  In Neptune, nodes can have one or more labels. Nodes with different individual labels (or different combinations of multiple labels) are written to different files, meaning that no individual file contains data for nodes with different combinations of labels. If a node has multiple labels, these labels are sorted alphabetically before they are assigned to a file.
- **`edges/`** – This folder contains edge data files in either a comma-separated value (CSV) or a JSON format.

As with the nodes files, edge data is written to different files based on a combinations of their labels. For purposes of model-training, edge data is assigned to different files based on a combination of the edge's label plus the labels of the edge's start and end nodes.

- **`statements/`** – This folder contains **RDF** data files in Turtle, N-Quads, N-Triples, or JSON format.
- **`config.json`** – This file contains the *schema* of the graph as inferred by the export process.
- **`lastEventId.json`** – This file contains the `commitNum` and `opNum` of the last event on the database's Neptune streams. The export process only includes this file if you set the `includeLastEventId` export parameter to `true`, and the database from which you are exporting data has Neptune streams enabled.

Parameters used to control the Neptune export process

Whether you are using the Neptune-Export service or the `neptune-export` command line utility, the parameters you use to control the export are mostly the same. They contain a JSON object passed to the Neptune-Export endpoint or to `neptune-export` on the command line.

The object passed in to the export process has the following top-level fields:

```
-d '{
      "command" : "(either export-pg or export-rdf)",
      "outputS3Path" : "s3:/(your Amazon S3 bucket)/(path to the folder for exported
 data)",
      "expectedBucketOwner" : "(your AWS account ID for Amazon S3 bucket)",
      "jobSize" : "(for Neptune-Export service only)",
      "params" : { (a JSON object that contains export-process parameters) },
      "additionalParams": { (a JSON object that contains parameters for training
 configuration) }
    }'
```

Contents

- The command parameter
- The outputS3Path parameter
- The expectedBucketOwner parameter
- The jobSize parameter
- The params object

- The additionalParams object
- Export parameter fields in the params top-level JSON object
  - List of possible fields in the export parameters params object
    - List of fields common to all types of export
    - List of fields for property-graph exports
    - List of fields for RDF exports
  - Fields common to all types of export
    - cloneCluster field in params
    - cloneClusterInstanceType field in params
    - cloneClusterReplicaCount field in params
    - cloneClusterEnableAuditLogs field in params
    - clusterId field in params
    - endpoint field in params
    - endpoints field in params
    - profile field in params
    - useIamAuth field in params
    - includeLastEventId field in params
  - Fields for property-graph export
    - concurrency field in params
    - edgeLabels field in params
    - filter field in params
    - filterConfigFile field in params
    - format field used for property-graph data in params
    - gremlinFilter field in params
    - gremlinNodeFilter field in params
    - gremlinEdgeFilter field in params
    - nodeLabels field in params
    - scope field in params
  - Fields for RDF export
    - format field used for RDF data in params

    - rdfExportScope field in params
    - sparql field in params
    - namedGraph field in params
- Examples of filtering what is exported
  - Filtering the export of property-graph data
    - Example of using scope to export only edges
    - Example of using nodeLabels and edgeLabels to export only nodes and edges having specific labels
    - Example of using filter to export only specified nodes, edges and properties
    - Example that uses gremlinFilter
    - Example that uses gremlinNodeFilter
    - Example that uses gremlinEdgeFilter
    - Combining filter, gremlinNodeFilter, nodeLabels, edgeLabels and scope
  - Filtering the export of RDF data
    - Using rdfExportScope and sparql to export specific edges
    - Using namedGraph to export a single named graph

The `command` parameter

The `command` top-level parameter determines whether to export property-graph data or RDF data. If you omit the `command` parameter, the export process defaults to exporting property-graph data.

- **`export-pg`** – Export property-graph data.
- **`export-rdf`** – Export RDF data.

The `outputS3Path` parameter

The `outputS3Path` top-level parameter is required, and must contain the URI of an Amazon S3 location to which the exported files can be published:

```
"outputS3Path" : "s3:////(your Amazon S3 bucket)/(path to output folder)"
```

The value must begin with `s3://`, followed by a valid bucket name and optionally a folder path within the bucket.

The `expectedBucketOwner` parameter

The `expectedBucketOwner` top-level parameter is optional. When set to an AWS account ID, it validates that the Amazon S3 bucket specified in `outputS3Path` is owned by that account. If omitted, it defaults to the account ID resolved from the credentials used for Amazon S3 operations.

```
"expectedBucketOwner" : "(your AWS account ID for Amazon S3 bucket)"
```

The `jobSize` parameter

The `jobSize` top-level parameter is only used with the the Neptune-Export service, not with the `neptune-export` command line utility, and is optional. It lets you characterize the size of the export job you are starting, which helps determine the amount of compute resources devoted to the job and its maximum concurrency level.

```
"jobSize" : "(one of four size descriptors)"
```

The four valid size descriptors are:

- `small` – Maximum concurrency: 8. Suitable for storage volumes up to 10 GB.
- `medium` – Maximum concurrency: 32. Suitable for storage volumes up to 100 GB.
- `large` – Maximum concurrency: 64. Suitable for storage volumes over 100 GB but less than 1 TB.
- `xlarge` – Maximum concurrency: 96. Suitable for storage volumes over 1 TB.

By default, an export initiated on the Neptune-Export service runs as a `small` job.

The performance of an export depends not only on the `jobSize` setting, but also on the number of database instances that you're exporting from, the size of each instance, and the effective concurrency level of the job.

For property-graph exports, you can configure the number of database instances using the cloneClusterReplicaCount parameter, and you can configure the job's effective concurrency level using the concurrency parameter.

The `params` object

The `params` top-level parameter is a JSON object that contains parameters that you use to control the export process itself, as explained in Export parameter fields in the `params` top-level JSON

object. Some of the fields in the `params` object are specific to property-graph exports, some to RDF.

The `additionalParams` object

The `additionalParams` top-level parameter is a JSON object that contains parameters you can use to control actions that are applied to the data after it has been exported. At present, `additionalParams` is used only for exporting training data for Neptune ML.

Export parameter fields in the `params` top-level JSON object

The Neptune export `params` JSON object allows you to control the export, including the type and format of the exported data.

List of possible fields in the export parameters `params` object

Listed below are all the possible top-level fields that can appear in a `params` object. Only a subset of these fields appear in any one object.

List of fields common to all types of export

- `cloneCluster`
- `cloneClusterInstanceType`
- `cloneClusterReplicaCount`
- `cloneClusterEnableAuditLogs`
- `clusterId`
- `endpoint`
- `endpoints`
- `profile`
- `useIamAuth`
- `includeLastEventId`

List of fields for property-graph exports

- `concurrency`
- `edgeLabels`
- `filter`
- `filterConfigFile`
- `gremlinFilter`
- `gremlinNodeFilter`
- `gremlinEdgeFilter`
- `format`
- `nodeLabels`
- `scope`

List of fields for RDF exports

- `format`
- `rdfExportScope`
- `sparql`
- `namedGraph`

Fields common to all types of export

`cloneCluster` field in `params`

(Optional). Default: `false`.

If the `cloneCluster` parameter is set to `true`, the export process uses a fast clone of your DB cluster:

```
"cloneCluster" : true
```

By default, the export process exports data from the DB cluster that you specify using the `endpoint`, `endpoints` or `clusterId` parameters. However, if your DB cluster is in use while the export is going on, and data is changing, the export process cannot guarantee the consistency of the data being exported.

To ensure that the exported data is consistent, use the `cloneCluster` parameter to export from a static clone of your DB cluster instead.

The cloned DB cluster is created in the same VPC as the source DB cluster and inherits the security group, subnet group and IAM database authentication settings of the source. When the export is complete, Neptune deletes the cloned DB cluster.

By default, a cloned DB cluster consists of a single instance of the same instance type as the primary instance in the source DB cluster. You can change the instance type used for the cloned DB cluster by specifying a different one using `cloneClusterInstanceType`.

> **Note**
>
> If you don't use the `cloneCluster` option, and are exporting directly from your main DB cluster, you might need to increase the timeout on the instances from which data is being exported. For large data sets, the timeout should be set to several hours.

`cloneClusterInstanceType` field in `params`

(Optional).

If the `cloneCluster` parameter is present and set to `true`, you can use the `cloneClusterInstanceType` parameter to specify the instance type used for the cloned DB cluster:

By default, a cloned DB cluster consists of a single instance of the same instance type as the primary instance in the source DB cluster.

```
"cloneClusterInstanceType" : "(for example, r5.12xlarge)"
```

`cloneClusterReplicaCount` field in `params`

(Optional).

If the `cloneCluster` parameter is present and set to `true`, you can use the `cloneClusterReplicaCount` parameter to specify the number of read-replica instances created in the cloned DB cluster:

```
"cloneClusterReplicaCount" : (for example, 3)
```

By default, a cloned DB cluster consists of a single primary instance. The `cloneClusterReplicaCount` parameter lets you specify how many additional read-replica instances should be created.

`cloneClusterEnableAuditLogs` field in `params`

(Optional). Default: false.

If the `cloneCluster` parameter is present and set to true, you can use the `cloneClusterEnableAuditLogs` parameter to enable or disable audit logs in the cloned cluster.

By default, audit logging is disabled.

```
"cloneClusterEnableAuditLogs" : true
```

`clusterId` field in `params`

(Optional).

The `clusterId` parameter specifies the ID of a DB cluster to use:

```
"clusterId" : "(the ID of your DB cluster)"
```

If you use the `clusterId` parameter, the export process uses all available instances in that DB cluster to extract data.

> **Note**
>
> The `endpoint`, `endpoints`, and `clusterId` parameters are mutually exclusive. Use one and only one of them.

`endpoint` field in `params`

(Optional).

Use `endpoint` to specify an endpoint of a Neptune instance in your DB cluster that the export process can query to extract data (see Endpoint Connections). This is the DNS name only, and does not include the protocol or port:

```
"endpoint" : "(a DNS endpoint of your DB cluster)"
```

Use a cluster or instance endpoint, but not the main reader endpoint.

> **Note**
>
> The `endpoint`, `endpoints`, and `clusterId` parameters are mutually exclusive. Use one and only one of them.

`endpoints` field in `params`

(Optional).

Use `endpoints` to specify a JSON array of endpoints in your DB cluster that the export process can query to extract data (see Endpoint Connections). These are DNS names only, and do not include the protocol or port:

```
"endpoints": [
  "(one endpoint in your DB cluster)",
  "(another endpoint in your DB cluster)",
  "(a third endpoint in your DB cluster)"
  ]
```

If you have multiple instances in your cluster (a primary and one or more read replicas), you can improve export performance by using the `endpoints` parameter to distribute queries across a list of those endpoints.

> **Note**
>
> The `endpoint`, `endpoints`, and `clusterId` parameters are mutually exclusive. Use one and only one of them.

`profile` field in `params`

(Required to export training data for Neptune ML, unless the `neptune_ml` field is present in the `additionalParams` field).

The `profile` parameter provides sets of pre-configured parameters for specific workloads. At present, the export process only supports the `neptune_ml` profile

If you are exporting training data for Neptune ML, add the following parameter to the `params` object:

```
"profile" : "neptune_ml"
```

`useIamAuth` field in `params`

(Optional). Default: `false`.

If the database from which you are exporting data has IAM authentication enabled, you must include the `useIamAuth` parameter set to `true`:

```
"useIamAuth" : true
```

`includeLastEventId` field in `params`

If you set `includeLastEventId` to true, and the database from which you are exporting data has Neptune Streams enabled, the export process writes a `lastEventId.json` file to your specified export location. This file contains the `commitNum` and `opNum` of the last event in the stream.

```
"includeLastEventId" : true
```

A cloned database created by the export process inherits the streams setting of its parent. If the parent has streams enabled, the clone will likewise have streams enabled. The contents of the stream on the clone will reflect the contents of the parent (including the same event IDs) at the point in time the clone was created.

Fields for property-graph export

`concurrency` field in `params`

(Optional). Default: 4.

The `concurrency` parameter specifies the number of parallel queries that the export process should use:

```
"concurrency" : (for example, 24)
```

A good guideline is to set the concurrency level to twice the number of vCPUs on all the instances from which you are exporting data. An r5.xlarge instance, for example, has 4 vCPUs. If you are exporting from a cluster of 3 r5.xlarge instances, you can set the concurrency level to 24 (= 3 x 2 x 4).

If you are using the Neptune-Export service, the concurrency level is limited by the jobSize setting. A small job, for example, supports a concurrency level of 8. If you try to specify a concurrency level of 24 for a small job using the `concurrency` parameter, the effective level remains at 8.

If you export from a cloned cluster, the export process calculates an appropriate concurrency level based on the size of the cloned instances and the job size.

`edgeLabels` field in `params`

(Optional).

Use `edgeLabels` to export only those edges that have labels that you specify:

```
"edgeLabels" : ["(a label)", "(another label"]
```

Each label in the JSON array must be a single, simple label.

The `scope` parameter takes precedence over the `edgeLabels` parameter, so if the `scope` value does not include edges, the `edgeLabels` parameter has no effect.

`filter` field in `params`

(Optional).

Use `filter` to specify that only nodes and/or edges with specific labels should be exported, and to filter the properties that are exported for each node or edge.

The general structure of a `filter` object, either inline or in a filter-configuration file, is as follows:

```
"filter" : {
  "nodes": [ (array of node label and properties objects) ],
  "edges": [ (array of edge definition an properties objects) ]
}
```

- **`nodes`** – Contains a JSON array of nodes and node properties in the following form:

  ```
  "nodes : [
    {
      "label": "(node label)",
      "properties": [ "(a property name)", "(another property name)", ( ... ) ]
    }
  ]
  ```

  - `label` – The node's property-graph label or labels.

    Takes a single value or, if the node has multiple labels, an array of values.
  - `properties` – Contains an array of the names of the node's properties that you want to export.
- **`edges`** – Contains a JSON array of edge definitions in the following form:

  ```
  "edges" : [
    {
      "label": "(edge label)",
  ```

```
        "properties": [ "(a property name)", "(another property name)", ( ... ) ]
      }
    ]
```

- `label` – The edge's property graph label. Takes a single value.
- `properties` – Contains an array of the names of the edge's properties that you want to export.

`filterConfigFile` field in `params`

(Optional).

Use `filterConfigFile` to specify a JSON file that contains a filter configuration in the same form that the `filter` parameter takes:

```
 "filterConfigFile" : "s3:////(your Amazon S3 bucket)/neptune-export/(the name of the
JSON file)"
```

See filter for the format of the `filterConfigFile` file.

`format` field used for property-graph data in `params`

(Optional). *Default*: `csv` (comma-separated values)

The `format` parameter specifies the output format of the exported property graph data:

```
 "format" : (one of: csv, csvNoHeaders, json, neptuneStreamsJson)
```

- **`csv`** – Comma-separated value (CSV) formatted output, with column headings formatted according to the Gremlin load data format.
- **`csvNoHeaders`** – CSV formatted data with no column headings.
- **`json`** – JSON formatted data.
- **`neptuneStreamsJson`** – JSON formatted data that uses the GREMLIN_JSON change serialization format.

`gremlinFilter` field in `params`

(Optional).

The `gremlinFilter` parameter allows you to supply a Gremlin snippet, such as a `has()` step, that is used to filter both nodes and edges:

```
"gremlinFilter" : (a Gremlin snippet)
```

Field names and string values should be surrounded by escaped double quotes. For dates and times, you can use the datetime method.

The following example exports only those nodes and edges with a date-created property whose value is greater than 2021-10-10:

```
"gremlinFilter" : "has(\"created\", gt(datetime(\"2021-10-10\")))"
```

`gremlinNodeFilter` field in `params`

(Optional).

The `gremlinNodeFilter` parameter allows you to supply a Gremlin snippet, such as a `has()` step, that is used to filter nodes:

```
"gremlinNodeFilter" : (a Gremlin snippet)
```

Field names and string values should be surrounded by escaped double quotes. For dates and times, you can use the datetime method.

The following example exports only those nodes with a `deleted` Boolean property whose value is `true`:

```
"gremlinNodeFilter" : "has(\"deleted\", true)"
```

`gremlinEdgeFilter` field in `params`

(Optional).

The `gremlinEdgeFilter` parameter allows you to supply a Gremlin snippet, such as a `has()` step, that is used to filter edges:

```
"gremlinEdgeFilter" : (a Gremlin snippet)
```

Field names and string values should be surrounded by escaped double quotes. For dates and times, you can use the datetime method.

The following example exports only those edges with a `strength` numerical property whose value is 5:

```
"gremlinEdgeFilter" : "has(\"strength\", 5)"
```

`nodeLabels` field in `params`

(Optional).

Use `nodeLabels` to export only those nodes that have labels you specify:

```
"nodeLabels" : ["(a label)", "(another label"]
```

Each label in the JSON array must be a single, simple label.

The `scope` parameter takes precedence over the `nodeLabels` parameter, so if the `scope` value does not include nodes, the `nodeLabels` parameter has no effect.

`scope` field in `params`

(Optional). Default: `all`.

The `scope` parameter specifies whether to export only nodes, or only edges, or both nodes and edges:

```
"scope" : (one of: nodes, edges, or all)
```

- `nodes` – Export nodes and their properties only.
- `edges` – Export edges and their properties only.
- `all` – Export both nodes and edges and their properties (the default).

Fields for RDF export

`format` field used for RDF data in `params`

(Optional). *Default*: `turtle`

The `format` parameter specifies the output format of the exported RDF data:

```
"format" : (one of: turtle, nquads, ntriples, neptuneStreamsJson)
```

- **`turtle`** – Turtle formatted output.
- **`nquads`** – N-Quads formatted data with no column headings.
- **`ntriples`** – N-Triples formatted data.
- **`neptuneStreamsJson`** – JSON formatted data that uses the SPARQL NQUADS change serialization format.

`rdfExportScope` field in `params`

(Optional). Default: `graph`.

The `rdfExportScope` parameter specifies the scope of the RDF export:

```
"rdfExportScope" : (one of: graph, edges, or query)
```

- `graph` – Export all RDF data.
- `edges` – Export only those triples that represent edges.
- `query` – Export data retrieved by a SPARQL query that issupplied using the `sparql` field.

`sparql` field in `params`

(Optional).

The `sparql` parameter allows you to specify a SPARQL query to retrieve the data to export:

```
"sparql" : (a SPARQL query)
```

If you supply a query using the `sparql` field, you must also set the `rdfExportScope` field to `query`.

`namedGraph` field in `params`

(Optional).

The `namedGraph` parameter allows you to specify an IRI to limit the export to a single named graph:

```
"namedGraph" : (Named graph IRI)
```

The `namedGraph` parameter can only be used with the `rdfExportScope` field set to `graph`.

Examples of filtering what is exported

Here are examples that illustrate ways to filter the data that is exported.

Filtering the export of property-graph data

Example of using `scope` to export only edges

```
{
  "command": "export-pg",
  "params": {
    "endpoint": "(your Neptune endpoint DNS name)",
    "scope": "edges"
  },
  "outputS3Path": "s3://(your Amazon S3 bucket)/neptune-export"
}
```

Example of using `nodeLabels` and `edgeLabels` to export only nodes and edges having specific labels

The `nodeLabels` parameter in the following example specifies that only nodes having a `Person` label or a `Post` label should be exported. The `edgeLabels` parameter specifies that only edges with a `likes` label should be exported:

```
{
  "command": "export-pg",
  "params": {
    "endpoint": "(your Neptune endpoint DNS name)",
    "nodeLabels": ["Person", "Post"],
    "edgeLabels": ["likes"]
  },
  "outputS3Path": "s3://(your Amazon S3 bucket)/neptune-export"
}
```

Example of using `filter` to export only specified nodes, edges and properties

The `filter` object in this example exports `country` nodes with their `type`, `code` and `desc` properties, and also `route` edges with their `dist` property.

```
{
  "command": "export-pg",
  "params": {
```

```
    "endpoint": "(your Neptune endpoint DNS name)",
    "filter": {
      "nodes": [
        {
          "label": "country",
          "properties": [
            "type",
            "code",
            "desc"
          ]
        }
      ],
      "edges": [
        {
          "label": "route",
          "properties": [
            "dist"
          ]
        }
      ]
    }
  },
  "outputS3Path": "s3:///(your Amazon S3 bucket)/neptune-export"
}
```

Example that uses `gremlinFilter`

This example uses `gremlinFilter` to export only those nodes and edges created after 2021-10-10 (that is, with a `created` property whose value is greater than 2021-10-10):

```
{
  "command": "export-pg",
  "params": {
    "endpoint": "(your Neptune endpoint DNS name)",
    "gremlinFilter" : "has(\"created\", gt(datetime(\"2021-10-10\")))"
  },
  "outputS3Path": "s3:///(your Amazon S3 bucket)/neptune-export"
}
```

Example that uses `gremlinNodeFilter`

This example uses `gremlinNodeFilter` to export only deleted nodes (nodes with a Boolean `deleted` property whose value is `true`):

```
{
  "command": "export-pg",
  "params": {
    "endpoint": "(your Neptune endpoint DNS name)",
    "gremlinNodeFilter" : "has(\"deleted\", true)"
  },
  "outputS3Path": "s3:///(your Amazon S3 bucket)/neptune-export"
}
```

Example that uses `gremlinEdgeFilter`

This example uses `gremlinEdgeFilter` to export only edges with a `strength` numerical property whose value is 5:

```
{
  "command": "export-pg",
  "params": {
    "endpoint": "(your Neptune endpoint DNS name)",
    "gremlinEdgeFilter" : "has(\"strength\", 5)"
  },
  "outputS3Path": "s3:///(your Amazon S3 bucket)/neptune-export"
}
```

Combining `filter`, `gremlinNodeFilter`, `nodeLabels`, `edgeLabels` and `scope`

The `filter` object in this example exports:

- `country` nodes with their `type`, `code` and `desc` properties
- `airport` nodes with their `code`, `icao` and `runways` properties
- `route` edges with their `dist` property

The `gremlinNodeFilter` parameter filters the nodes so that only nodes with a `code` property whose value begins with A are exported.

The `nodeLabels` and `edgeLabels` parameters further restrict the output so that only `airport` nodes and `route` edges are exported.

Finally, the `scope` parameter eliminates edges from the export, which leaves only the designated `airport` nodes in the output.

```
{
  "command": "export-pg",
  "params": {
    "endpoint": "(your Neptune endpoint DNS name)",
    "filter": {
      "nodes": [
        {
          "label": "airport",
          "properties": [
            "code",
            "icao",
            "runways"
          ]
        },
        {
          "label": "country",
          "properties": [
            "type",
            "code",
            "desc"
          ]
        }
      ],
      "edges": [
        {
          "label": "route",
          "properties": [
            "dist"
          ]
        }
      ]
    },
    "gremlinNodeFilter": "has(\"code\", startingWith(\"A\"))",
    "nodeLabels": [
      "airport"
    ],
    "edgeLabels": [
      "route"
    ],
    "scope": "nodes"
  },
  "outputS3Path": "s3:///(your Amazon S3 bucket)/neptune-export"
}
```

Filtering the export of RDF data

Using `rdfExportScope` and `sparql` to export specific edges

This example exports triples whose predicate is <http://kelvinlawrence.net/air-routes/objectProperty/route> and whose object is not a literal:

```
{
  "command": "export-rdf",
  "params": {
    "endpoint": "(your Neptune endpoint DNS name)",
    "rdfExportScope": "query",
    "sparql": "CONSTRUCT { ?s <http://kelvinlawrence.net/air-routes/objectProperty/
route> ?o } WHERE { ?s ?p ?o . FILTER(!isLiteral(?o)) }"
  },
  "outputS3Path": "s3:///(your Amazon S3 bucket)/neptune-export"
}
```

Using `namedGraph` to export a single named graph

This example exports triples belonging to the named graph <http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph>:

```
{
  "command": "export-rdf",
  "params": {
    "endpoint": "(your Neptune endpoint DNS name)",
    "rdfExportScope": "graph",
    "namedGraph": "http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph"
  },
  "outputS3Path": "s3://(your Amazon S3 bucket)/neptune-export"
}
```

Troubleshooting the Neptune export process

The Amazon Neptune export process uses AWS Batch to provision the compute and storage resources necessary to export your Neptune data. When an export is running, you can use the link in the `logs` field to access the CloudWatch logs for the export job.

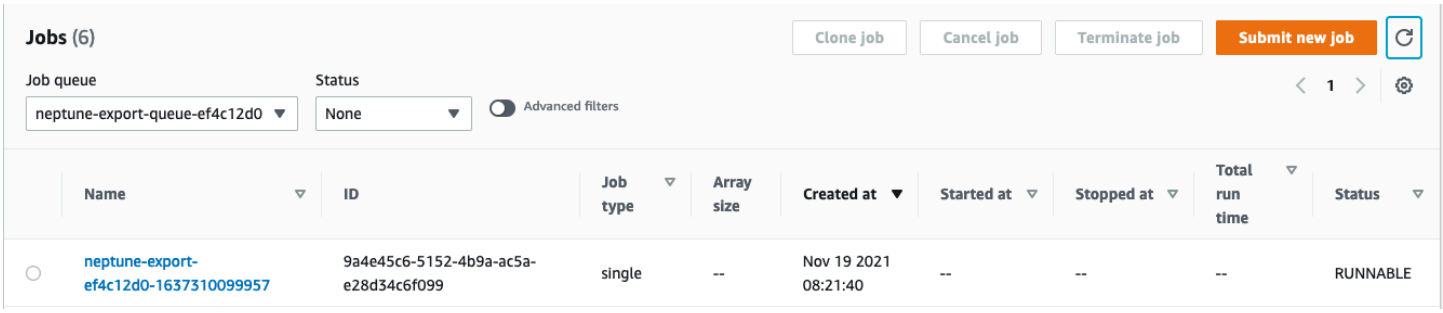
However, the CloudWatch logs for the AWS Batch job that performs the export are only available when the AWS Batch job is running. If Neptune export reports that an export is in a pending state,

there won't be a logs link through which you can access CloudWatch logs. If an export job remains in the `pending` state for more than a few minutes, there may be a problem provisioning the underlying AWS Batch resources.

When the export job leaves the pending state, you can check its status as follows:

To check the status of a AWS Batch job

1. Open the AWS Batch console at https://eusc-de-east-1.console.amazonaws-eusc.eu/batch/.
2. Select the neptune-export job queue.
3. Look for the job whose name matches the `jobName` returned by Neptune export when you started the export.

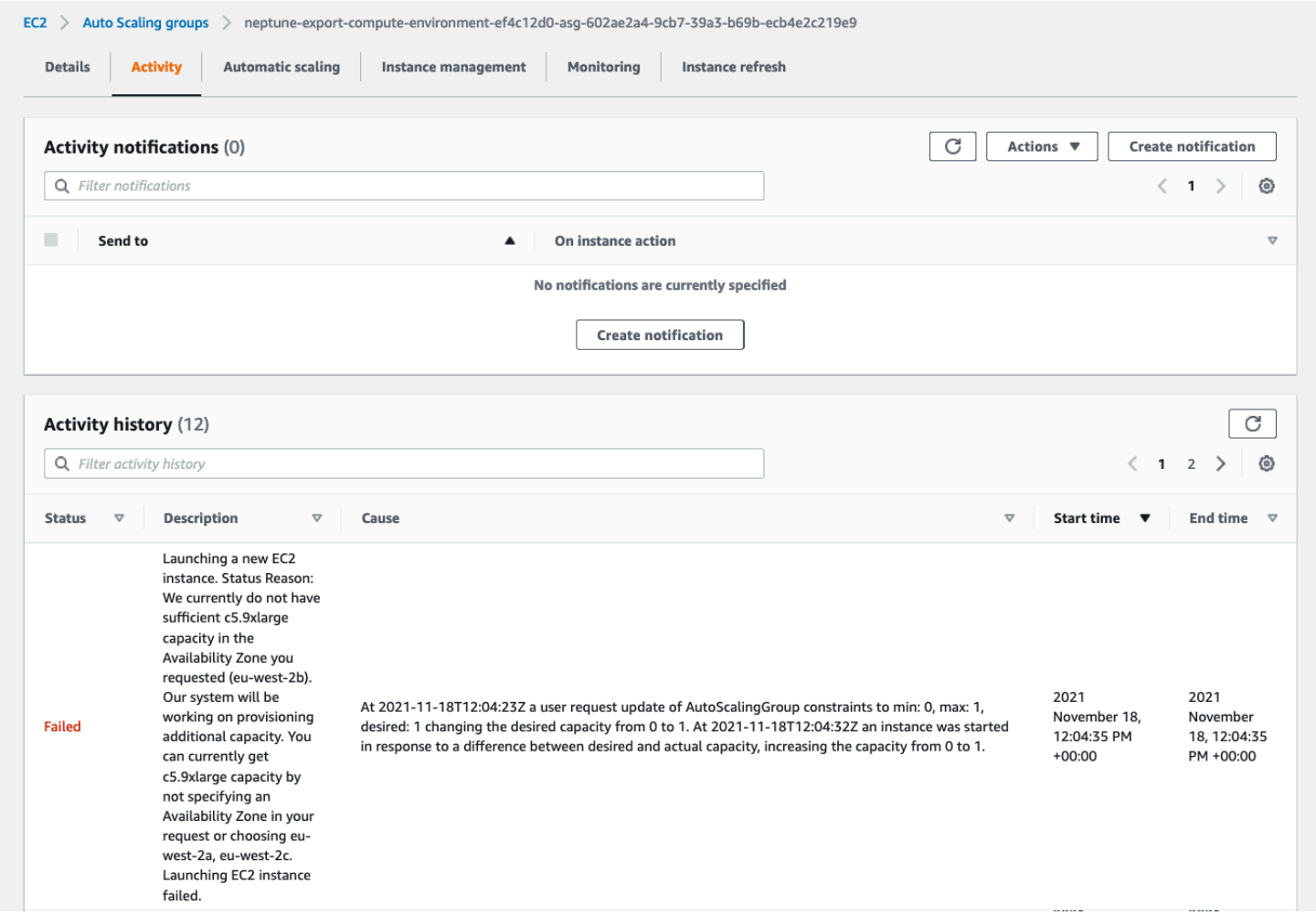


If the job remains stuck in a `RUNNABLE` state, it may be because networking or security issues are preventing the container instance from joining the underlying Amazon Elastic Container Service (Amazon ECS) cluster. See the section about verifying network and security settings of the compute environment in this support article.

Another thing you can check is for problems with auto-scaling:

To check the Amazon EC2 auto-scaling group for the AWS Batch compute environment

1. Open the Amazon EC2 console at https://eusc-de-east-1.console.amazonaws-eusc.eu/ec2/.
2. Select the **Auto Scaling** group for the neptune-export compute environment.
3. Open the **Activity** tab and check the activity history for unsuccessful events.



Neptune Export common errors

`org.eclipse.rdf4j.query.QueryEvaluationException: Tag mismatch!`

If an `export-rdf` job is regularly failing with a `Tag mismatch! QueryEvaluationException`, the Neptune instance is undersized for the large, long-running queries used by Neptune Export.

You can avoid getting this error by scaling up to a larger Neptune instance or by configuring the job to export from a large cloned cluster, like this:

```
'{
  "command": "export-rdf",
  "outputS3Path": "s3:////(your Amazon S3 bucket)/neptune-export",
  "params": {
    "endpoint": "(your Neptune endpoint DNS name)",
    "cloneCluster": True,
    "cloneClusterInstanceType" : "r5.24xlarge"
```

```
  }
}'
```

Managing Your Amazon Neptune Database

This section shows how to manage and maintain your Neptune DB cluster using the AWS Management Console and the AWS CLI.

Neptune operates on clusters of database servers that are connected in a replication topology. Thus, managing Neptune often involves deploying changes to multiple servers and making sure that all Neptune replicas are keeping up with the primary server.

Because Neptune transparently scales the underlying storage as your data grows, managing Neptune requires relatively little management of disk storage. Likewise, because Neptune automatically performs continuous backups, a Neptune cluster does not require extensive planning or downtime for performing backups.

Topics

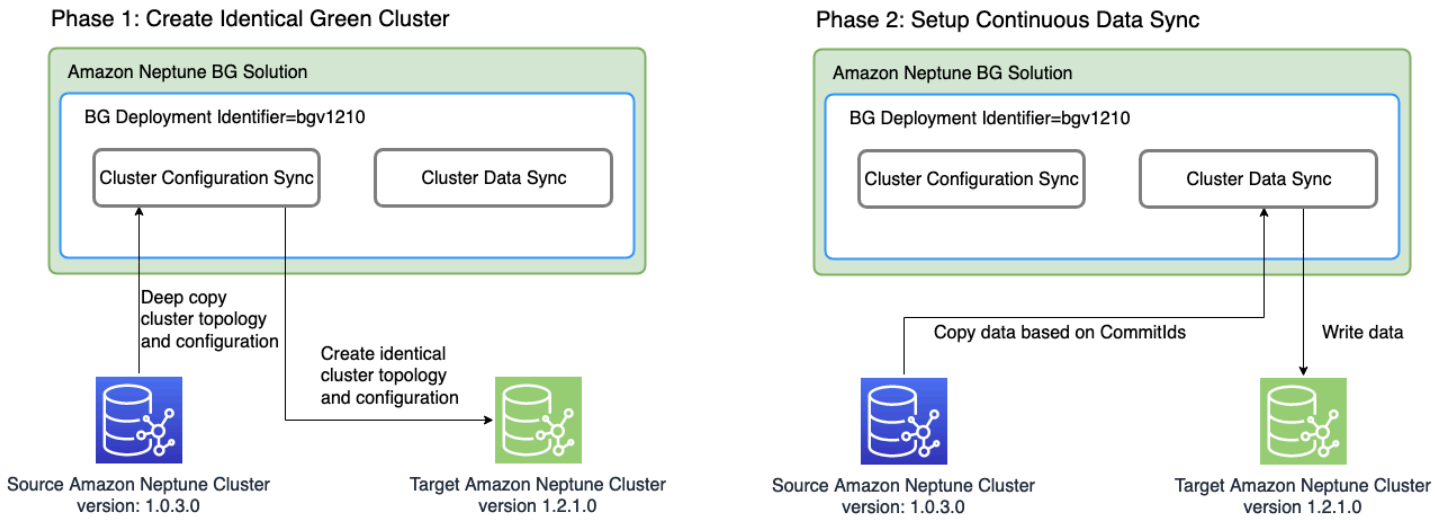
- Using the Neptune Blue/Green solution to perform blue-green updates
- Creating an IAM user with permissions for Neptune
- Amazon Neptune parameter groups
- Amazon Neptune parameters
- Launching a Neptune DB cluster using the AWS Management Console
- Stopping and starting an Amazon Neptune DB cluster
- Empty an Amazon Neptune DB cluster using the fast reset API
- Adding Neptune reader instances to a DB Cluster
- Creating a Neptune reader instance using the console
- Modifying a Neptune DB Cluster Using the Console
- Performance and Scaling in Amazon Neptune
- Auto-scaling the number of replicas in an Amazon Neptune DB cluster
- Maintaining your Amazon Neptune DB Cluster
- Using a CloudFormation template to update the engine version of your Neptune DB Cluster
- Database Cloning in Neptune
- Managing Amazon Neptune Instances

Using the Neptune Blue/Green solution to perform blue-green updates

Amazon Neptune engine upgrades can require application downtime because the database is unavailable while the updates are being installed and verified. This is true whether they are initiated manually or automatically.

Neptune provides a Blue/Green deployment solution that you can run using an CloudFormation stack and that greatly reduces such downtime. It creates a green staging environment that is synchronized with your blue production environment. You can then update that staging environment to perform a minor or major engine version upgrade, a graph data model change, or an operating-system update, and test the result. Finally, you can switch it over quickly to become your production environment, with very little downtime.

The Neptune Blue/Green solution goes through two phases, as illustrated in this diagram:



Phase 1 creates a Green DB cluster identical to your production cluster

The solution creates a DB cluster with a unique blue/green deployment identifier and with the same cluster topology as your production cluster. That is, it has the same number and sizes of DB instances, the same parameter groups and all the same configurations as the production (blue) DB cluster except that it has been upgraded to the target engine version that you specified, which must be higher than your current (blue) engine version. You can specify a minor and major engine version for the target. If necessary, the solution will perform any intermediate upgrades required to reach the specified target engine version. This new cluster becomes the green staging environment.

Phase 2 sets up continuous data synchronization

After the green environment has been fully prepared, the solution sets up continuous replication between the source (blue) cluster and the target (green) cluster using Neptune streams. When the replication difference between them reaches zero, the staging environment is ready for testing. At that point you must pause writing to the blue cluster to avoid any further replication lag.

Your target engine version may have new features or dependencies that affect your applications. Check the target engine release page and intervening engine release pages under Engine releases to see what has changed since your current engine version. It's best to run integration tests or verify your applications manually on the green cluster before promoting it to the production environment.

After you have tested and qualified the changes in the green cluster, just switch the database endpoint in your applications from the blue to the green cluster.

After switchover, the Neptune Blue/Green solution does not delete the old blue production environment. You will still have access to it for additional validation and testing if needed. Standard billing charges do apply to its instances until you delete them. The Blue/Green solution also uses other AWS services, the costs for which are billed at normal prices. Details on deleting the solution when you're done with it are covered in the clean up section.

Prerequisites for running the Neptune Blue/Green stack

Before launching the Neptune Blue/Green stack:

- Be sure to enable Neptune streams on your production (blue) cluster.
- All the instances in your blue cluster must be in the **available** state. You can check instance states in the Neptune console or by using the describe-db-instances API.
- All instances must also be in sync with the DB cluster parameter group.
- The Neptune Blue/Green solution requires a DynamoDB VPC endpoint in the VPC where your blue cluster is located. See Using Amazon VPC endpoints to access DynamoDB.
- Choose at time to run the solution when the write workload on your blue production DB cluster will be as light as possible. Avoid, for example, running the solution when a bulk load will be taking place, or when there's likely to be a large number of write operations for any other reason.

Using an CloudFormation template to run the Neptune Blue/Green solution

You can use AWS CloudFormation to deploy the Neptune Blue/Green solution. The CloudFormation template creates an Amazon EC2 instance in the same VPC as your blue source Neptune database, installs the solution there, and runs it. You can monitor its progress in CloudWatch logs, as explained in Monitoring progress.

You can use these links to review the solution template, or select the **Launch Stack** button to launch it in the CloudFormation console:

View	View in Designer	Launch Stack

In the console, choose the AWS region where you want to run the solution from the dropdown at the upper right of the window.

Set the stack parameters as follows:

- **`DeploymentID`** – An identifier that is unique to each Neptune Blue/Green deployment.

  It is used as the green DB cluster identifier, and as a prefix for naming new resources created during the deployment.
- **`NeptuneSourceClusterId`** – The identifier of the blue DB cluster that you want to upgrade.
- **`NeptuneTargetClusterVersion:`** – The Neptune engine version that you want to upgrade the blue DB cluster to.

  This must be higher than the current blue DB cluster's engine version.
- **`DeploymentMode`** – Indicates whether this is a new deployment or an attempt to resume a previous deployment. When you are using same `DeploymentID` as a previous deployment, set `DeploymentMode` to `resume`.

  Valid values are: new (the default), and `resume`.
- **`GraphQueryType`** – The graph data type for your database.

  Valid values are: `propertygraph` (the default), and `rdf`.

- **`SubnetId`** – A subnet ID from the same VPC that your blue DB cluster is located in. (see Connecting to a Neptune DB Cluster from an Amazon EC2 instance in the same VPC).

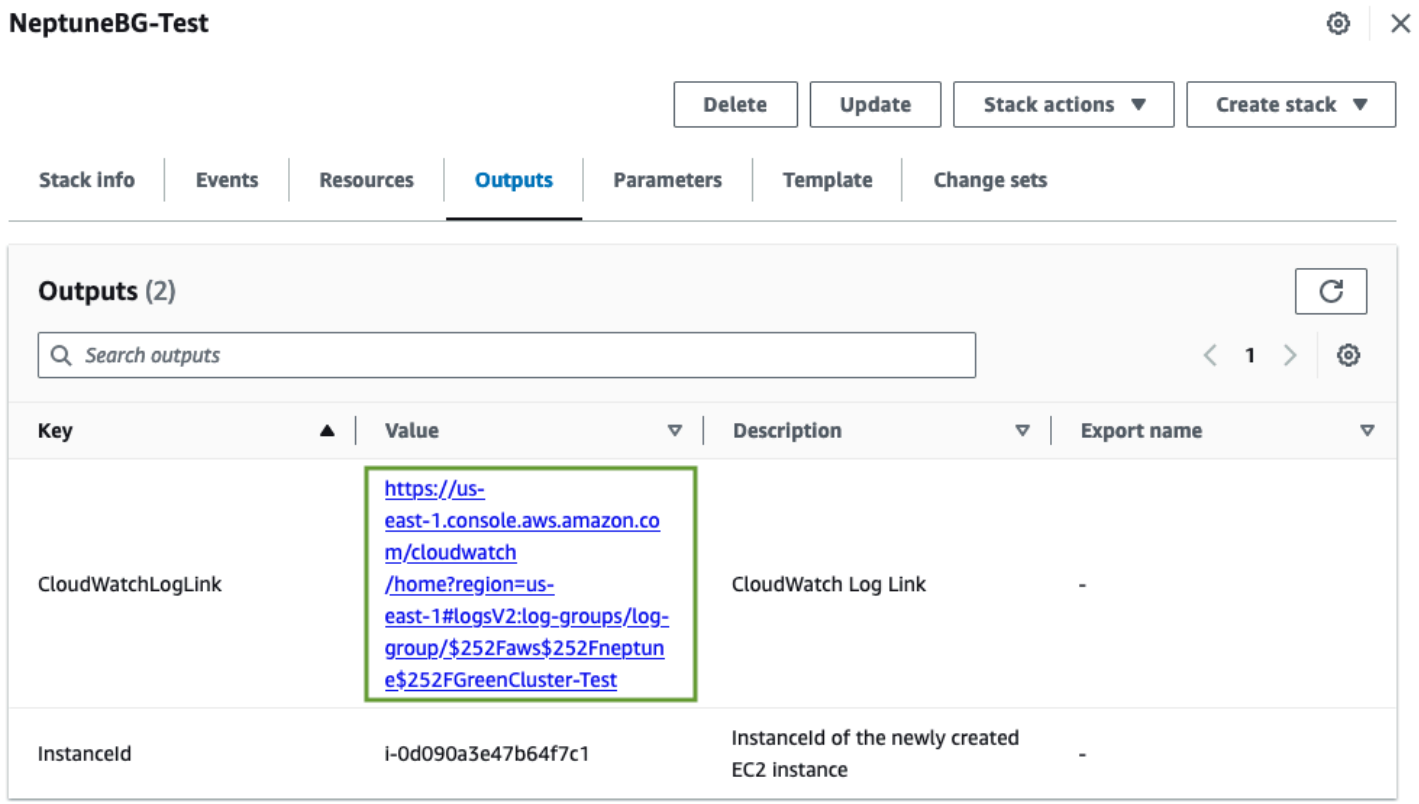
  Provide the ID of a public subnet if you want to SSH to the instance through EC2 Connect.
- **`InstanceSecurityGroup`** – A security group for your Amazon EC2 instance.

  The security group must have access to your blue DB cluster, and you must be able to SSH to the instance. See Create a security group using the VPC console.

Wait until the stack is complete. As soon as it's done the solution is started. You can then monitor deployment process using CloudWatch logs as described in the next section.

Monitoring the progress of a Neptune Blue/Green deployment

You can monitor the progress of the Neptune Blue/Green solution by going to the CloudWatch console and looking at logs in the `/aws/neptune/`*`(Neptune Blue/Green deployment ID)* CloudWatch log group. You can find a link to the CloudWatch logs in the outputs of the solution's CloudFormation stack:



If you provided a public subnet as a stack parameter, you can also SSH to your Amazon EC2 instance created as part of the stack and refer to the log in `/var/log/cloud-init-output.log`.

The log shows the actions taken by the Neptune Blue/Green solution, as shown in this screenshot:

```
============================================================

Neptune Blue Green Deployment Solution Version: 0.1.06012023

============================================================

Checking whether cluster with id = bg-06-01-14-20-29test-bg1-bgInt already exists.

BlueGreen deployment_mode = new

Didn't find any cluster with id bg-06-01-14-20-29test-bg1-bgInt

Cloned_cluster_id: bg-06-01-14-20-29test-bg1-bgInt

Replication_stack_name: bg-06-01-14-20-29test-bg1-bgInt-replication

DescribeDbClusters response for test-bg1-bgIntegTest-06-01-14-20-29: {'AllocatedStorage': 1,
'AvailabilityZones': ['us-east-1b', 'us-east-1c', 'us-east-1f'], 'BackupRetentionPeriod': 1,
'DBClusterIdentifier': 'test-bg1-bgintegtest-06-01-14-20-29', 'DBClusterParameterGroup': 'green-     -blue-
green-deployment-test-123456789012345-pg-tes710', 'DBSubnetGroup': 'default', 'Status': 'available',
'EarliestRestorableTime': datetime.datetime(2023, 6, 1, 8, 51, 23, 394000, tzinfo=tzlocal()), 'Endpoint':
'test-bg1-bgintegtest-06-01-14-20-29.cluster-critvszpmydm.us-east-1.neptune.amazonaws.com', 'ReaderEndpoint':
'test-bg1-bgintegtest-06-01-14-20-29.cluster-ro-critvszpmydm.us-east-1.neptune.amazonaws.com', 'MultiAZ':
False, 'Engine': 'neptune', 'EngineVersion': '1.2.0.0', 'LatestRestorableTime': datetime.datetime(2023, 6, 1,
8, 51, 23, 394000, tzinfo=tzlocal()), 'Port': 8182, 'MasterUsername': 'admin', 'PreferredBackupWindow':
'06:33-07:03', 'PreferredMaintenanceWindow': 'fri:09:44-fri:10:14', 'ReadReplicaIdentifiers': [],
'DBClusterMembers': [{'DBInstanceIdentifier': 'test-bg1-bgintegtest-06-01-14-20-29i-1', 'IsClusterWriter':
True, 'DBClusterParameterGroupStatus': 'in-sync', 'PromotionTier': 1}], 'VpcSecurityGroups':
```

Log messages show the sync status between the blue and green clusters:

```
DDB checkpoint {'S': '1'}, {'S': '6'}

DDB Checkpoint: {'checkpointSubSequenceNumber': {'S': '6'}, 'lastUpdateTime': {'N': '1685611142127'},
'leaseOwner': {'S': 'nobody'}, 'checkpoint': {'S': '1'}, 'leaseKey': {'S': 'bg-anl   -234567899-replication'}}

Time difference for last checkpoint and last stream event: 5841351

Stream eventId difference for last replication checkpoint and last stream event on the Source cluster: 0:0

Found region : us-east-1

Cloudwatch Log Url for blue green solution is https://us-east-1.console.aws.amazon.com/cloudwatch
/home?region=us-east-1#logsV2:log-groups/log-group/aws/neptune/bg

Cloudwatch dashboard url for replication is https://console.aws.amazon.com/cloudwatch/home?region=us-
east-1#dashboards:name=neptune-stream-poller-bg-an   -234567899-replication

Replication poller lambda arn is arn:aws:lambda:us-east-1:451235071234:function:bg-an   -234567899-replic-
NeptuneStreamPollerLambd-B6V1ytULgmSP. Look for CW log the poller lambda for more troubleshooting.

Stream Last EventId {'commitNum': 1, 'opNum': 6} on cluster : database-d61852469-t        -experiment.cluster-
critvszpmydm.us-east-1.neptune.amazonaws.com:8182

DDB checkpoint {'S': '1'}, {'S': '6'}

DDB Checkpoint: {'checkpointSubSequenceNumber': {'S': '6'}, 'lastUpdateTime': {'N': '1685611207245'},
'leaseOwner': {'S': 'nobody'}, 'checkpoint': {'S': '1'}, 'leaseKey': {'S': 'bg-ankig-234567899-replication'}}
```

The sync process checks the replication lag by computing the difference between the latest stream `eventID` on the blue cluster and the replication checkpoint present in the DynamoDB checkpoint table created by the Neptune-to-Neptune replication stack. Using these messages, you can monitor the current replication difference.

Cutting over from the production blue cluster to the updated green cluster

Before promoting the green cluster to production, ensure that the commit difference between the blue and green clusters is zero and then disable all write traffic to the blue cluster. Continuing to write to the blue cluster while switching the database endpoint to the green cluster can result in data corruption caused by writing partial data to both clusters. You may not need to disable read traffic yet.

If you have enabled IAM authentication on the source (blue) cluster, be sure to update any IAM policies used in your applications to point to the green cluster (for an example of such a policy, see this unrestricted access policy).

After disabling write traffic, wait for replication to finish and then enable write traffic on the green cluster (but not on the blue cluster). Switch read traffic from the blue to the green cluster as well.

Cleaning up after the Neptune Blue/Green solution has completed

After you have promoted the staging (green) cluster to production, clean up the resources created by the Neptune Blue/Green solution:

- Delete the Amazon EC2 instance that was created to run the solution.
- Delete the CloudFormation templates for the Neptune streams-based replication that kept the green cluster in sync with the blue cluster. The main one has the stack name that you provided earlier, and one is composed of the deployment ID followd by "-replication": that is, `(DeploymentID)-replication`.

Deleting CloudFormation templates doesn't delete the clusters themselves. Once you have verified that the green cluster is working as expected, you can optionally take a snapshot before manually deleting the blue cluster.

Neptune Blue/Green solution best practices

- Before switching your green cluster over to production, it is worth thoroughly verifying that it is functioning properly. Check the consistency of the data and the configuration of the database. It is possible that some of the new engine versions require client upgrades as well. Check the engine release notes before you upgrade. It is worth testing all this in development, testing, and pre-production environments before starting a blue/green upgrade in production.
- It is best to perform the switch-over from the blue to the green server during your maintence window.
- To ensure that everything is working properly after upgrading and synchronizing, it's worth keeping your original cluster for some period of time before deleting it. It could prove useful if an unforseen issue arises.
- Avoid heavy write operations such as bulk loads when running the Neptune Blue/Green solution, because they can cause replication lag that introduces significant downtime. Ideally, the time between turning off writes to your blue cluster and turning them on for your green cluster is just a few moments.

Troubleshooting the Neptune Blue/Green solution

The following information highlights issues that can arise during the Blue/Green solution deployment process, such as conflicts with existing clusters, the need to enable Neptune streams, ongoing bulk load operations, and version compatibility requirements. By addressing these potential problems, you can ensure a smooth and successful deployment of the Neptune Blue/Green solution.

Errors raised by the Neptune Blue/Green solution

- **`Cluster with id = (blue_green_deployment_id) already exists`** – There is an existing cluster with identifier *`(blue_green_deployment_id)`*.

  Provide a new deployment ID or set the deployment mode to `resume` if the cluster was created in a previous Neptune Blue/Green run.
- **`Streams should be enabled on the source Cluster for Blue Green Deployment`** – Enable Neptune streams on the blue (source) cluster.
- **`No Bulkload should be in progress on source cluster: (cluster_id)`** – The Neptune Blue/Green solution terminates if it identifies an ongoing bulk load.

  This is to ensure that the sync process is able to catch up with writes being made. Avoid or cancel any ongoing bulk load job before starting the Neptune Blue/Green solution.
- **`Blue Green deployment requires instances to be in sync with db cluster parameter group`** – Any changes to cluster parameter group should be in sync throughout the DB cluster. See Amazon Neptune parameter groups.
- **`Invalid target engine version for Blue Green Deployment`** – The target engine version must be listed as active in Engine releases for Amazon Neptune, and must be higher than the current engine release of the source (blue) cluster.

Creating an IAM user with permissions for Neptune

To access the Neptune console to create and manage a Neptune DB cluster, you need to create an IAM user with all the necessary permissions.

The first step is to create a service-linked role policy for Neptune:

Create a service-linked role policy for Amazon Neptune

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane on the left, choose **Policies**.
3. On the **Policies** page, select **Create Policy**.
4. On the **Create policy** page, select the **JSON** tab and copy in the following service-linked role policy:

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Action": "iam:CreateServiceLinkedRole",
      "Effect": "Allow",
      "Resource": "arn:aws-eusc:iam::*:role/aws-service-role/
rds.amazonaws.com/AWSServiceRoleForRDS",
      "Condition": {
        "StringLike": {
            "iam:AWSServiceName":"rds.amazonaws.com"
        }
      }
    }
  ]
}
```

5. Select **Next: Tags**, and on the **Add tags** page select **Next: Review**.
6. On the **Review policy** page, name the new policy "NeptuneServiceLinked".

For more information about service-linked roles, see Using service-linked roles for Amazon Neptune.

Create a new IAM user with all necessary permissions

Next, create the new IAM user with the appropriate managed policies attached that will grant the permissions you'll need, along with the service-linked role policy that you have created (here named `NeptuneServiceLinked`):

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane on the left, choose **Users**, and on the **Users** page, choose **Add users**.
3. On the **Add user** page, enter a name for the new IAM user, choose **Access key - Programatic access** for the AWS credential type, and choose **Next: Permissions**.
4. On the **Set permissions** page, in the **Filter policies** box, type "Neptune". Now select the following from the policies that are listed:
   - **NeptuneFullAccess**
   - **NeptuneConsoleFullAccess**
   - **NeptuneServiceLinked** (assuming that is what you named the service-linked role policy that you created earlier).
5. Next type "VPC" in the **Filter policies** box in place of "Neptune". Select **AmazonVPCFullAccess** from the policies that are listed.
6. Select **Next: Tags**, and in the **Add tags** page, select **Next: Review**.
7. In the **Review** page, check that all of the following policies are now attached to your new user:
   - **NeptuneFullAccess**
   - **NeptuneConsoleFullAccess**
   - **NeptuneServiceLinked**
   - **AmazonVPCFullAccess**

   Then, select **Create User**.
8. Finally, download and save the new user's access key ID and secret access key.

To interoperate on other services such as Amazon Simple Storage Service (Amazon S3), you will need to add more permissions and trust relationships.

Amazon Neptune parameter groups

You manage your database configuration in Amazon Neptune by using parameters in a parameter group. Parameter groups act as a *container* for engine configuration values that are applied to one or more DB instances.

There are two types of parameter group, namely DB cluster parameter groups and DB parameter groups:

- *DB parameter groups* apply at the instance level and generally are associated with settings for the Neptune graph engine, such as the `neptune_query_timeout` parameter.
- *DB cluster parameter groups* apply to every instance in the cluster and generally have broader settings. Every Neptune cluster is associated with a DB cluster parameter group. Every DB instance within that cluster inherits the engine configuration values contained in the DB cluster parameter group.

Any configuration values that you modify in the DB cluster parameter group override default values in the DB parameter group. If you edit the corresponding values in the DB parameter group, those values override the settings in the DB cluster parameter group.

A default DB parameter group is used if you create a DB instance without specifying a custom DB parameter group. You can't modify the parameter settings of the default DB parameter group. Instead, to change the default parameter settings you must create a new DB parameter group. Not all DB engine parameters can be changed in a DB parameter group that you create.

Parameter groups are created in families that are compatible with specific Neptune engine versions. When you upgrade to a new major or minor engine version, you may need to recreate your custom parameter groups using the corresponding parameter group family for that version.

The parameter group family naming follows the pattern `neptuneX.Y`, where `X.Y` matches the engine version. For example:

- `neptune1` – for engine versions prior to 1.2.0.0
- `neptune1.2` – for engine versions 1.2.x
- `neptune1.3` – for engine versions 1.3.x
- `neptune1.4` – for engine versions 1.4.x

When upgrading your Neptune cluster, check the release notes for your target engine version to determine if a new parameter group family is required. If so, you must recreate all custom parameter groups in the new family before upgrading.

Some Neptune parameters are static, and others are dynamic. The differences are as follows:

Static parameters

- A static parameter is one that takes effect only after a DB instance is rebooted. In other words, when you change a static parameter and save the instance DB parameter group, you must manually reboot the DB instance for the parameter change to take effect. Currently, all the Neptune instance-level parameters (in a DB parameter group rather than a DB cluster parameter group) are static.
- When you change a cluster-level static parameter and save the DB cluster parameter group, the parameter change takes effect after you manually reboot every DB instance in the cluster.

Dynamic parameters

- A dynamic parameter is one that takes effect almost immediately after the parameter is updated in its parameter group. In other words, there is no need to reboot a DB instance after updating a dynamic parameter for the parameter change to take effect.
- Expect some minor delay for a dynamic cluster parameter change to be applied across all DB instances.
- An updated dynamic parameter value is not applied to currently running requests, but only to ones submitted after the change took place.
- When you change a dynamic cluster-level parameter, by default the parameter change is applied to your DB cluster immediately, without requiring any reboot. To defer the parameter change until after the DB instances in the cluster are rebooted, you can use the AWS CLI to set the `ApplyMethod` to `pending-reboot` for the parameter change.

Currently all parameters are static except for the following new cluster parameters:

- `neptune_enable_slow_query_log` (cluster-level)
- `neptune_slow_query_log_threshold` (cluster-level)

Here are some important points you should know about working with parameters in a DB parameter group:

- Improperly setting parameters in a DB parameter group can have unintended adverse effects, including degraded performance and system instability. Always exercise caution when modifying database parameters, and back up your data before modifying a DB parameter group. Try out your parameter group setting changes on a test DB instance before applying those changes to a production DB instance.
- When you change the DB parameter group associated with a DB instance, you must manually reboot the instance before the new DB parameter group is used by the DB instance.

> **Note**
>
> Before Release: 1.2.0.0 (2022-07-21), all the read-replica instances in a DB cluster were automatically rebooted whenever the primary (writer) instance restarted.
> From Release: 1.2.0.0 (2022-07-21) going forwards, restarting the primary instance does not cause any of the replica instances to restart. This means that if you are changing a cluster-level parameter, you must restart each instance separately to pick up the parameter change.

Editing a DB Cluster Parameter Group or DB Parameter Group

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. Choose **Parameter groups** in the navigation pane.
3. Choose the **Name** link for the DB parameter group that you want to edit.

   (Optional) Choose **Create parameter group** to create a new cluster parameter group and create the new group. Then choose the **Name** of the new parameter group.

   > **Important**
   >
   > This step is *required* if you only have the default DB cluster parameter group because the default DB cluster parameter group can't be modified.

4. Search for the parameter and click on the **Value** field next to the **Name** column.

5. Enter the allowed value and choose the check beside the value field.
6. Choose **Save changes**.
7. Reboot every DB instance in the Neptune cluster if you are changing a DB cluster parameter, or one or more specific instances if you are changing a DB instance parameter.

Creating a DB parameter group or DB cluster parameter group

You can easily use the Neptune console to create a new parameter group:

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. Choose **Parameter groups** in the left navigation pane.
3. Choose **Create DB parameter group**.

   The **Create DB parameter group** page appears.
4. In the **Parameter group family** list, choose the family that matches your target Neptune engine version (for example, **neptune1.2**, **neptune1.3**, or **neptune1.4**).
5. In the **Type** list, choose **DB Parameter Group** or **DB Cluster Parameter Group**.
6. In the **Group name** box, type the name of the new DB parameter group.
7. In the **Description** box, type a description for the new DB parameter group.
8. Choose **Create**.

You can also create a new parameter group using the AWS CLI:

```
aws neptune create-db-parameter-group \
  --db-parameter-group-name (a name for the new DB parameter group) \
  --db-parameter-group-family (the family matching your engine version, such as
 neptune1.2, neptune1.3, or neptune1.4) \
  --description (a description for the new DB parameter group)
```

Amazon Neptune parameters

You manage your database configuration in Amazon Neptune by using parameters in parameter groups. The following parameters are available for configuring your Neptune database:

Cluster-level parameters

- neptune_enable_audit_log
- neptune_enable_slow_query_log
- neptune_slow_query_log_threshold
- neptune_lab_mode
- neptune_query_timeout
- neptune_streams
- neptune_streams_expiry_days
- neptune_lookup_cache
- neptune_autoscaling_config
- neptune_ml_iam_role
- neptune_ml_endpoint
- neptune_enable_inline_server_generated_edge_id

Instance-level parameters

- neptune_dfe_query_engine
- neptune_query_timeout
- neptune_result_cache

Deprecated parameters

- neptune_enforce_ssl

`neptune_enable_audit_log` (cluster-level parameter)

This parameter toggles audit logging for Neptune.

Allowed values are `0` (disabled) and `1` (enabled). The default value is `0`.

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

You can publish audit logs to Amazon CloudWatch, as described in Using the CLI to publish Neptune audit logs to CloudWatch Logs.

`neptune_enable_slow_query_log` (cluster-level parameter)

Use this parameter to enable or disable Neptune's slow-query logging feature.

This is a dynamic parameter, meaning that changing its value does not require or cause a restart of your DB cluster.

Allowed values are:

- **`info`** – Enables slow-query logging and logs selected attributes that might be contributing to the slow performance.
- **`debug`** – Enables slow-query logging and logs all available attributes of the query run.
- **`disabled`** – Disables slow-query logging.

The default value is `disabled`.

You can publish slow-query logs to Amazon CloudWatch, as described in Using the CLI to publish Neptune slow-query logs to CloudWatch Logs.

`neptune_slow_query_log_threshold` (cluster-level parameter)

This parameter specifies the execution time threshold, in milliseconds, after which a query is considered a slow query. If slow-query logging is enabled, queries that run longer than this threshold will be logged together with some of their attributes.

The default value is 5000 milliseconds (5 seconds).

This is a dynamic parameter, meaning that changing its value does not require or cause a restart of your DB cluster.

`neptune_lab_mode` (cluster-level parameter)

When set, this parameter enables specific experimental features of Neptune. See Neptune Lab Mode for the experimental features currently available.

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

Lab mode features are typically disabled by default. To enable or explicitly disable an experimental feature, include `(feature name)`=`enabled` or `(feature name)`=`disabled` in this parameter. You can enable or disable multiple features by separating them with commas, like this:

`(feature #1 name)`=`enabled`, `(feature #2 name)`=`enabled`

`neptune_query_timeout` (cluster-level parameter)

Specifies a specific timeout duration for graph queries, in milliseconds.

Allowed values range from `10` to `2,147,483,647` (2^{31} - 1). The default value is `120,000` (2 minutes).

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

When multiple timeout settings are configured (cluster-level, instance-level, and per-query), the following table shows which timeout value takes precedence:

Cluster PG	Instance PG	Query Hint	Outcome
Default	Default	none	Cluster
Custom	Default	none	Cluster
Custom	Custom	none	Instance
Default	Custom	none	Instance

Cluster PG	Instance PG	Query Hint	Outcome
Any	Any	lowest	Query
Default	Custom	not lowest	Instance
Custom	Default	not lowest	Cluster
Custom	Custom	not lowest	Instance

> **Note**
>
> It is possible to incur unexpected costs if you set the query timeout value too high, particularly on a serverless instance. Without a reasonable timeout setting, you may inadvertently issue a query that keeps running much longer than you expected, incurring costs you never anticipated. This is particularly true on a serverless instance that could scale up to a large, expensive instance type while running the query.
> You can avoid unexpected expenses of this kind by using a query timeout value that accomodates most of your queries and only causes unexpectedly long-running ones to time out.

`neptune_streams` (cluster-level parameter)

Enables or disables Neptune streams.

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

Allowed values are `0` (disabled, which is the default), and `1` (enabled).

`neptune_streams_expiry_days` (cluster-level parameter)

Specifies how many days elapse before the server deletes stream records.

Allowed values are from `1` to `90`, inclusive. The default is `7`.

This parameter was introduced in engine version 1.2.0.0.

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

neptune_lookup_cache (cluster-level parameter)

Disables or re-enables the Neptune lookup cache. on R5d instances.

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

Allowed values are 1 (enabled) and 0 (disabled). The default value is 0, but whenever an R5d instance is created in the DB cluster, the neptune_lookup_cache parameter is automatically set to 1 and a lookup cache is created on that instance.

neptune_autoscaling_config (cluster-level parameter)

Sets configuration parameters for the read-replica instances that Neptune auto-scaling creates and manages.

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

Using a JSON string that you set as the value of the neptune_autoscaling_config parameter, you can specify:

- The instance type that Neptune auto-scaling uses for all the new read-replica instances that it creates.
- The maintenance windows assigned to those read-replicas.
- Tags to be associated with all the new read-replicas.

The JSON string has a structure like this:

```
"{
  \"tags\": [
    { \"key\" : \"reader tag-0 key\", \"value\" : \"reader tag-0 value\" },
    { \"key\" : \"reader tag-1 key\", \"value\" : \"reader tag-1 value\" },
  ],
  \"maintenanceWindow\" : \"wed:12:03-wed:12:33\",
  \"dbInstanceClass\" : \"db.r5.xlarge\"
```

```
}"
```

Note that the quotation marks within the string must all be escaped with a backslash character (\).

Any of the three configuration settings not specified in the `neptune_autoscaling_config` parameter are copied from the configuration of the DB cluster's primary writer instance.

`neptune_ml_iam_role` (cluster-level parameter)

Specifies the IAM role ARN used in Neptune ML. The value can be any valid IAM role ARN.

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

You can specify the default IAM role ARN for machine learning on graphs.

`neptune_ml_endpoint` (cluster-level parameter)

Specifies the endpoint used for Neptune ML. The value can be any valid SageMaker AI endpoint name.

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

You can specify the default SageMaker AI endpoint for machine learning on graphs.

`neptune_enable_inline_server_generated_edge_id` (cluster-level parameter)

Enable or disable the Neptune inline server-generated Edge ID feature.

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

Allowed values are 1 (enabled), and `0` (disabled). The default value is `0`.

`neptune_dfe_query_engine` (instance-level parameter)

This DB instance parameter controls how the DFE query engine is used. Allowed values are as follows:

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

- **`enabled`** – Causes the DFE engine to be used wherever possible, except where the `useDFE` query hint is present and set to `false`.
- **`viaQueryHint`** (the default) – Causes the DFE engine to be used only for queries that explicitly include the `useDFE` query hint set to `true`.

If this parameter has not been explicitly set, the default value, `viaQueryHint`, is used when the instance is started.

> **Note**
>
> All openCypher queries are executed by the DFE engine regardless of how this parameter is set.

`neptune_query_timeout` (instance-level parameter)

This DB instance parameter specifies a timeout duration for graph queries, in milliseconds, for one instance.

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

Allowed values range from `10` to `2,147,483,647` (2^{31} - 1). The default value is `120,000` (2 minutes).

> **Note**
>
> It is possible to incur unexpected costs if you set the query timeout value too high, particularly on a serverless instance. Without a reasonable timeout setting, you may inadvertently issue a query that keeps running much longer than you expected, incurring costs you never anticipated. This is particularly true on a serverless instance that could scale up to a large, expensive instance type while running the query.
> You can avoid unexpected expenses of this kind by using a query timeout value that accomodates most of your queries and only causes unexpectedly long-running ones to time out.

`neptune_result_cache` (instance-level parameter)

`neptune_result_cache` – This DB instance parameter enables or disables Caching query results.

This parameter is static, meaning that changes to it do not take effect on any instance until it has been rebooted.

Allowed values are `0`, (disabled, which is the default), and 1 (enabled).

`neptune_enforce_ssl` (DEPRECATED cluster-level parameter)

(**Deprecated**) There used to be regions that permitted HTTP connections to Neptune, and this parameter was used to force all connections to use HTTPS when it was set to 1. This parameter is no longer relevant, however, since Neptune now only accepts HTTPS connections in all regions.

Launching a Neptune DB cluster using the AWS Management Console

The easiest way to launch a new Neptune DB cluster is to use an CloudFormation template that creates all the required resources for you, as explained in Create Neptune cluster.

If you prefer, you can also use the Neptune console to launch a new DB cluster manually, as explained here.

> **Note**
>
> Before you can access the Neptune console to create a Neptune cluster, you must have a user with sufficent permissions. If your current user does not have these permissions then you can create an IAM user with the necessary permissions to do so, as explained in Creating an IAM user with permissions for Neptune.

Once you have verified that your user has the correct permissions, or you have created a user with the correct permissions, log into the AWS Management Console as that IAM user and follow the steps below to create a new DB cluster:

To launch a Neptune DB cluster using the console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. Navigate to the **Clusters** page under **Databases** and choose **Create database**, which opens the **Create database** page.
3. Under **Settings**, enter a name for your new DB cluster or accept the default name that is supplied there. This name is used in the endpoint address of the instance, and must satisfy the following constraints:
   - It must contain from 1 to 63 alphanumeric characters or hyphens.
   - Its first character must be a letter.
   - It cannot end with a hyphen or contain two consecutive hyphens.
   - It must be unique across all DB instances in your AWS account in a given AWS Region.
4. Under **Templates**, choose either **Production** or **Development and Testing**.

5. Under **DB instance size**, choose an instance size. This will determine the processing and memory capacity of the primary write instance of your new DB cluster.

   If you selected the **Production** template, you can only choose from among the available memory-optimized classes listed, but if you selected the **Development and testing**, you can also choose from among the more economical burstable classes (see T3 Burstable Instances for a discussion of burstable classes).

   > **Note**
   >
   > Neptune no longer supports R4 instance types.

6. Under **Availability and durability**, you can choose whether or not to enable multi-availability-zone (multi-AZ) deployment. The production template enables multi-AZ deployment by default, whereas the development and testing template does not. If multi-AZ deployment is enabled, Neptune locates read-replica instances that you create in different availability zones (AZs) to improve availability.

7. Under **Connectivity**, select the virtual private cloud (VPC) that will host your new DB cluster from among the available choices. Here you can choose **Create new VPC** if you want Neptune to create the VPC for you. You must create an Amazon EC2 instance in this same VPC to access the Neptune instance (for more information, see Securing your Amazon Neptune database with Amazon VPC). Note that you can't change the VPC after the DB cluster has been created.

   If you need to, you can further configure connectivity for your cluster under **Additional connectivity configuration**:

   a. Under **Subnet group**, you can choose the Neptune DB subnet group to use for the new DB cluster. If your VPC does not yet have any subnet groups, Neptune creates a DB subnet group for you (see Securing your Amazon Neptune database with Amazon VPC).

   b. Under **VPC security groups**, choose one or more existing VPC security groups to secure network access to the new DB cluster, or choose **Create new** if you want Neptune to create one for you, and then supply a name for the new VPC security group (see Create a security group using the VPC console).

   c. Under **Database port**, enter the TCP/IP port that the database will use for application connections. Neptune uses port number 8182 as the default.

d. Under **Network type**, choose **IPv4** to allow only IPv4 addresses to communicate with the DB cluster, or choose **Dual-stack** to allow both IPv4 and IPv6 addresses to communicate with the DB cluster. For more information, see Dual-stack mode.

8. Under **Tags**, you can associate up to 50 tags with your new DB cluster.

9. Under **Additional configuration**, there are more settings that you can make for your new DB cluster (in many cases, you can skip them and accept default values for now):

Option	What you can do
DB instance identifier	You can provide a name for the writer instance of the cluster. If you don't, a default identifier based on the cluster name is used. If you do, specify a name that is unique for all DB instances owned by your AWS account in the current region. The DB instance identifier is case insensitive, but stored as all lower-case.
DB cluster parameter group	Select a DB cluster parameter group to define the default configuration for all DB instances in the cluster. Unless you choose otherwise, Neptune uses a default DB cluster parameter group. For more informati on about parameter groups, see Amazon Neptune parameter groups.
DB parameter group	Select a DB parameter group to define the configuration of the primary DB instance in the cluster. Unless you choose otherwise , Neptune uses a default parameter group. For more information about parameter groups, see Parameter groups.
IAM DB authentication	If you check **Enable IAM DB authentic ation**, all access to your database will be authenticated using AWS Identity and Access Management (IAM).

Option	What you can do
	Important This requires that you sign all requests with AWS Signature Version 4 signing. For more information, see Authenticating your Amazon Neptune database with AWS Identity and Access Management.
Failover priority	Choose `No preference` or a priority tier for failover. If you choose a tier and there is contention within it, the replica that is the same size as the primary instance is selected.
Backup retention period	Choose the length of time, from 1 to 35 days, that Neptune should retain automatic backups of this DB instance. You can only perform a point-in-time restore (PITR) to a time within the backup retention period.
Copy tags to snapshots	(*Enabled by default*) This option causes all tags associated with your DB cluster to be copied to any snapshots of it.

Option	What you can do
Enable encryption	(*Enabled by default*) This option causes the data in your DB cluster to be encrypted at rest. If you do, choose the master key used to protect the key that is used to encrypt this database volume. You can select the default `aws/rds` key, or choose from master keys in your account, or enter the ARN of a key from a different account. You can create a new master encryption key on the **Encryption Keys** tab of the IAM console. For more information, see Encrypting Neptune resources at rest.
Audit log	Check this if you want audit logs from your DB cluster published to CloudWatch Logs.
Enable auto minor version upgrade	(*Enabled by default*) This option causes your DB cluster to be automatically upgraded to new minor engine versions after they are released. The automatic upgrades occur during the maintenance window for the database. See Using AutoMinorVersionUpgrade.
Maintenance window	You can select a specific period during which you want pending modifications to your DB cluster to happen, such as a change to a DB instance class or an automatic engine patch. Any such maintenance operation s are started and completed within the selected period. If you do not select a period, Neptune assigns a maintenance period arbitrarily.

Option	What you can do
Enable deletion protection	(*Enabled by default*) Deletion protection blocks your DB cluster from being deleted. You must explicitly disable it in order to delete the DB cluster.

10. Choose **Create database** to launch your new Neptune DB cluster and its primary instance.

    On the Amazon Neptune console, the new DB cluster appears in the list of Databases. The DB cluster has a status of **Creating** until it is created and ready for use. When the state changes to **Available**, you can connect to the primary instance for your DB cluster. Depending on the DB instance class and store allocated, it can take several minutes for the new instances to be available.

    To view the newly created cluster, choose the **Databases** view in the Neptune console.

    > **Note**
    >
    > If you delete all Neptune DB instances in a DB cluster using the AWS Management Console, the console automatically deletes the DB cluster itself. If you are using the AWS CLI or SDK, you must delete the DB cluster manually after you delete its last instance.

    Make note of the **Cluster endpoint** value. You need this to connect to your Neptune DB cluster.

Stopping and starting an Amazon Neptune DB cluster

Stopping and starting Amazon Neptune clusters helps you manage costs for development and test environments. You can temporarily stop all the DB instances in your cluster, instead of setting up and tearing down all the DB instances each time that you use the cluster.

Topics

- Overview of stopping and starting a Neptune DB cluster
- Stopping a Neptune DB cluster
- Starting a stopped Neptune DB cluster

Overview of stopping and starting a Neptune DB cluster

During periods where you don't need a Neptune cluster, you can stop all instances in that cluster at once. You can start the cluster again anytime you need to use it. Starting and stopping simplifies the setup and teardown processes for clusters used for development, testing, or similar activities that don't require continuous availability. You can accomplish this in the AWS Management Console with a single action, regardless of how many instances there are in the cluster.

While your DB cluster is stopped, you are charged only for cluster storage, manual snapshots, and automated backup storage within your specified retention window. You aren't charged for any DB instance hours.

After seven days, Neptune automatically starts your DB cluster again to make sure that it doesn't fall behind any required maintenance updates.

To minimize charges for a lightly loaded Neptune cluster, you can stop the cluster instead of deleting all its read replicas. For clusters with more than one or two instances, frequently deleting and recreating the DB instances is only practical using the AWS CLI or Neptune API, and deletions can also be difficult to perform in the right order. For example, you must delete all read replicas before deleting the primary instance to avoid activating the failover mechanism.

Don't use starting and stopping if you need to keep your DB cluster running but you want to reduce capacity. If your cluster is too costly or not very busy, you can delete one or more DB instances or change your DB instances to use a smaller instance class, but you can't stop an individual DB instance.

Stopping a Neptune DB cluster

When you won't be using it for a while, you can stop a running Neptune DB cluster, and then start it again when you need it. While the cluster is stopped you are charged for cluster storage, manual snapshots, and automated backup storage within your specified retention window, but not for DB instance hours.

The stop operation stops all the cluster's read replica instances before stopping the primary instance, to avoid activating the failover mechanism.

Stopping a DB cluster using the AWS Management Console

To use the AWS Management Console to stop a Neptune cluster

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Databases**, and then choose a cluster. You can perform the stop operation from this page, or navigate to the details page for the DB cluster that you want to stop.
3. In **Actions**, choose **Stop**.

Stopping a DB cluster using the AWS CLI

To stop a DB instance by using the AWS CLI, call the stop-db-cluster command, using the `--db-cluster-identifier` parameter to identify the DB cluster you want to stop.

Example

```
aws neptune stop-db-cluster --db-cluster-identifier mydbcluster
```

Stopping a DB cluster using the Neptune management API

To stop a DB instance by using the Neptune management API, call the StopDBCluster API and use the `DBClusterIdentifier` parameter to identify the DB cluster you want to stop.

What can happen while a DB cluster is stopped

- You **can** restore it from a snapshot (see Restoring from a DB Cluster Snapshot).
- You **can't** modify the configuration of the DB cluster or any of its DB instances.

- You **can't** add or remove DB instances from the cluster.
- You **can't** delete the cluster if it still has any associated DB instances.
- In general, you must re-start a stopped DB cluster to perform most administrative actions.
- Neptune applies any scheduled maintenance to your stopped cluster as soon as it is started again. Remember that after seven days, Neptune automatically re-starts a stopped cluster so that it doesn't fall too far behind in maintenance status.
- Neptune does not perform any automated backups of a stopped DB cluster, because the underlying data cannot change while the cluster is stopped.
- Neptune does not extend the backup retention period for the DB cluster while it is stopped.

Starting a stopped Neptune DB cluster

You can only start a Neptune DB cluster that is in the stopped state. When you start the cluster, all its DB instances become available again. The cluster retains its configuration settings, such as endpoints, parameter groups and VPC security groups.

Starting a stopped DB cluster using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Databases**, and then choose a cluster. You can perform the start operation from this page, or navigate to the details page for that DB cluster and start from there.
3. In **Actions**, choose **Start**.

Starting a stopped DB cluster using the AWS CLI

To start a stopped DB cluster using the AWS CLI, call the start-db-cluster command using the `--db-cluster-identifier` parameter to specify the stopped DB cluster that you want to start. Provide either the cluster name that you chose when creating the DB cluster, or use a DB instance name that you chose with `-cluster` appended to the end of it.

Example

```
aws neptune start-db-cluster --db-cluster-identifier mydbcluster
```

Starting a stopped DB cluster using the Neptune management API

To start a Neptune DB cluster by using the Neptune management API, call the StartDBCluster API using the `DBCluster` parameter to specify the stopped DB cluster that you want to start. Provide either the cluster name that you chose when creating the DB cluster, or use a DB instance name that you chose, with `-cluster` appended to the end of it.

Empty an Amazon Neptune DB cluster using the fast reset API

The Neptune fast reset REST API lets you reset a Neptune graph quicky and easily, removing all of its data.

- In most cases, a fast reset operation completes within couple of minutes. The duration can vary somewhat depending on the load on the cluster when the operation is initiated.
- A fast reset operation does not result in additional I/Os.
- Storage volume size does not shrink after a fast reset. Instead, the storage is reused as new data is inserted. This means that the volume sizes of snapshots taken before and after a fast reset operation will be the same. Volume sizes of restored clusters using the snapshots created before and after a fast reset operation will also be the same
- As part of the reset operation, all instances in the DB cluster are restarted.

> **Note**
>
> Under rare conditions, these server restarts may also result in failover of the cluster.

> **Important**
>
> Using fast reset may break the integration of your Neptune DB cluster with other services. For example:
>
> - Fast reset deletes all stream data from your database and completely resets streams. This means that your stream consumers may no longer work without new configuration.
> - Fast reset removes all metadata about SageMaker AI resources being used by Neptune ML, including jobs and endpoints. They continue to exist in SageMaker AI, and you can continue to use existing SageMaker AI endpoints for Neptune ML inference queries, but the Neptune ML management APIs no longer work with them.
> - Integrations such as the full-text-search integration with ElasticSearch are also wiped out by fast reset, and must be re-established manually before they can be used again.

To delete all data from a Neptune DB cluster using the API

1. First, you generate a token that you can then use to perform the database reset. This step is intended to help prevent anyone from accidentally resetting a database.

   You do this by sending an `HTTP POST` request to the `/system` endpoint on the writer instance of your DB cluster to specify the `initiateDatabaseReset` action.

   The `curl` command using the JSON content-type would be:

   ```
   curl -X POST \
     -H 'Content-Type: application/json' \
         https://your_writer_instance_endpoint:8182/system \
     -d '{ "action" : "initiateDatabaseReset" }'
   ```

   Or, using the `x-www-form-urlencoded` content type:

   ```
   curl -X POST \
     -H 'Content-Type: application/x-www-form-urlencoded' \
         https://your_writer_instance_endpoint:8182/system \
     -d 'action=initiateDatabaseReset '
   ```

   The `initiateDatabaseReset` request returns the reset token in its JSON response, like this:

   ```
   {
     "status" : "200 OK",
     "payload" : {
       "token" : "new_token_guid"
     }
   }
   ```

   The token remains valid for one hour (60 minutes) after it is issued.

   If you send the request to a reader instance or to the status endpoint, Neptune will throw a `ReadOnlyViolationException`.

   If you send multiple `initiateDatabaseReset` requests, only the latest token generated will be valid for the second step, where you actually perform the reset.

If the server restarts right after your `initiateDatabaseReset` request, the generated token becomes invalid, and you need to send a new request to get a new token.

2. Next, you send a `performDatabaseReset` request with the token that you got back from `initiateDatabaseReset` to the `/system` endpoint on the writer instance of your DB cluster. This deletes all data from your DB cluster.

   The `curl` command using the JSON content-type is:

```
curl -X POST \
  -H 'Content-Type: application/json' \
      https://your_writer_instance_endpoint:8182/system \
  -d '{
        "action" : "performDatabaseReset",
        "token" : "token_guid"
      }'
```

   Or, using the `x-www-form-urlencoded` content type:

```
curl -X POST \
  -H 'Content-Type: application/x-www-form-urlencoded' \
      https://your_writer_instance_endpoint:8182/system \
  -d 'action=performDatabaseReset&token=token_guid'
```

   The request returns a JSON response. If the request is accepted, the response is:

```
{
  "status" : "200 OK"
}
```

   If the token you sent doesn't match the one that was issued, the response looks like this:

```
{
  "code" : "InvalidParameterException",
  "requestId":"token_guid",
  "detailedMessage" : "System command parameter 'token' : 'token_guid' does not
 match database reset token"
}
```

If the request is accepted and the reset begins, the server restarts and deletes the data. You cannot send any other requests to the DB cluster while it is resetting.

Using the fast reset API with IAM-Auth

If you have IAM-Auth enabled on your DB cluster, you can use awscurl to send fast reset commands that are authenticated using IAM-Auth:

Using awscurl to send fast-reset requests with IAM-Auth

1. Set the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables correctly (and also `AWS_SECURITY_TOKEN` if you are using temporary credential).
2. An `initiateDatabaseReset` request looks like this:

```
awscurl -X POST --service neptune-db "$SYSTEM_ENDPOINT" \
  -H 'Content-Type: application/json' --region us-west-2 \
  -d '{ "action" : "initiateDatabaseReset" }'
```

3. A `performDatabaseReset` request looks like this:

```
awscurl -X POST --service neptune-db "$SYSTEM_ENDPOINT" \
  -H 'Content-Type: application/json' --region us-west-2 \
  -d '{ "action" : "performDatabaseReset" }'
```

Common error codes for fast reset operations

Neptune error code	HTTP status	Message	Example
`InvalidParameterException`	400	System command parameter '*action*' has unsupported value '*XXX*'	Invalid parameter
`InvalidParameterException`	400	Too many values supplied for: *action*	A fast reset request with more than one action sent with header 'Content-

Neptune error code	HTTP status	Message	Example
			type:application/x-www-form-urlencoded'
`InvalidParameterException`	400	Duplicate field 'action'	A fast reset request with more than one action sent with header 'Content-Type: application/json'
`MethodNotAllowedException`	400	Bad route: /*bad_endpoint*	Request sent to an incorrect endpoint
`MissingParameterException`	400	Missing required parameters: [action]	A fast reset request doesn't contain the required 'action' parameter
`ReadOnlyViolationException`	400	Writes are not permitted on a read replica instance	A fast reset request was sent to a reader or status endpoint
`AccessDeniedException`	403	Missing Authentication Token	A fast reset request was sent without correct signatures to a DB endpoint with IAM-Auth enabled
`ServerShutdownException`	500	Database reset is in progress. Please retry the query after the cluster is available.	When fast reset begins, existing and incoming Gremlin/Sparql queries fail.

Adding Neptune reader instances to a DB Cluster

In Neptune DB clusters, there is one primary DB instance and up to 15 Neptune reader instances. The primary DB instance supports read and write operations, and performs all of the data modifications to the cluster volume. Neptune reader instances connect to the same storage volume as the primary DB instance and support only read operations.

Use reader instances to offload read workloads from the primary DB instance.

We recommend that you distribute the primary instance and Neptune readers in your DB cluster over multiple Availability Zones to improve the availability of your DB cluster.

The following section describes how to create a reader instance in your DB cluster.

Creating a Neptune reader instance using the console

After creating the primary instance for your Neptune DB cluster, you can add additional Neptune reader instances using the Neptune console.

To create a Neptune reader instance using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Databases**.
3. Select the DB cluster where you want to create the reader instance.
4. Choose **Actions**, and then choose **Add reader**.
5. On the **Create replica DB instance** page, specify options for your Neptune replica. The following table shows settings for a Neptune read replica.

For This Option...	Do This
DB instance class	Choose a DB instance class that defines the processin g and memory requirements for the Neptune replica. For a current listing of the DB instance classes that Neptune offers in different regions, see the Neptune pricing page.
Availability zone	Specify an Availability Zone. Choose a different zone than the primary DB instance. The list includes only those Availability Zones that are mapped by the DB subnet group for the DB cluster.
Encryption	Enable or disable encryption.
Read replica source	Choose the identifier of the primary instance to create a Neptune replica for.
DB instance identifier	Enter a name for the instance that is unique for your account in the Region that you selected. You might choose to add some intelligence to the name, such as

For This Option...	Do This
	including the Availability Zone selected, for example `neptune-us-east-1c` .
Database port	Port number on which the database accepts connections.
DB parameter group	The parameter group for this instance.
Log exports	Choose the logs you want to publish, if any.
Auto Minor Version Upgrade	Choose **Yes** if you want to enable your Neptune replica to receive minor Neptune DB engine version upgrades automatically when they become available. The **Auto Minor Version Upgrade** option applies only to minor upgrades. It does not apply to engine maintenance patches, which are always applied automatically to maintain system stability.

6. Choose **Create read replica** to create the Neptune replica instance.

To remove a Neptune reader instance from a DB cluster, follow the instructions in Deleting a DB instance in Amazon Neptune.

Modifying a Neptune DB Cluster Using the Console

When you modify a DB instance using the AWS Management Console, you can choose to apply the changes right away by selecting **Apply Immediately**. If you choose to apply changes immediately, your new changes and any changes in the pending modifications queue are applied at once.

If you don't choose to apply changes immediately, the changes are put into the pending modifications queue. During the next maintenance window, any pending changes in the queue are applied.

> **Important**
>
> If any pending modifications require downtime, choosing to apply changes immediately can cause unexpected downtime for the DB instance in question. There is no downtime for the other DB instances in the DB cluster.

> **Note**
>
> When you modify a DB cluster in Neptune, the **Apply Immediately** setting only affects changes to the **DB cluster identifier**, **IAM DB authentication**. All other modifications are applied immediately, regardless of the value of the **Apply Immediately** setting.

To modify a DB cluster using the console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Clusters**, and then choose the DB cluster that you want to modify.
3. Choose **Actions**, and then choose **Modify cluster**. The **Modify DB cluster** page appears.
4. Change any of the settings that you want.

   For example, you can change the **Network type** to **Dual-stack** to enable IPv6 support for the DB cluster. For more information, see Dual-stack mode.

> **Note**
>
> On the console, some instance level changes only apply to the current DB instance, whereas others apply to the entire DB cluster. To change a setting that modifies the entire DB cluster at the instance level on the console, follow the instructions in Modifying a DB Instance in a DB Cluster.

5. When all the changes are as you want them, choose **Continue** and check the summary.
6. To apply the changes immediately, select **Apply immediately**.
7. On the confirmation page, review your changes. If they are correct, choose **Modify cluster** to save your changes.

   To edit your changes, choose **Back**, or to cancel your changes, choose **Cancel**.

Modifying a DB Instance in a DB Cluster

To modify a DB Instance in a DB cluster using the console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Instances**, and then choose the DB instance that you want to modify.
3. Choose **Instance actions**, and then choose **Modify**. The **Modify DB Instance** page appears.
4. Change any of the settings that you want.

> **Note**
>
> Some settings apply to the entire DB cluster and must be changed at the cluster level. To change those settings, follow the instructions in Modifying a Neptune DB Cluster Using the Console.
> In the AWS Management Console, some instance-level changes apply only to the current DB instance, whereas others apply to the entire DB cluster.

5. When all the changes are as you want them, choose **Continue** and check the summary.
6. To apply the changes immediately, select **Apply immediately**.

7. On the confirmation page, review your changes. If they are correct, choose **Modify DB Instance** to save your changes.

   To edit your changes, choose **Back**, or to cancel your changes, choose **Cancel**.

Performance and Scaling in Amazon Neptune

Neptune DB clusters and instances scale at three different levels:

- Storage Scaling
- Instance Scaling;
- Read Scaling

Storage Scaling in Neptune

Neptune storage automatically scales with the data in your cluster volume. As your data grows, your cluster volume storage grows, up to 128 TiB in all supported regions except China and GovCloud, where it is limited to 64 TiB.

The size of your cluster volume is checked on an hourly basis to determine your storage costs.

Storage consumed by your Neptune database is billed in per GB-month increments and I/Os consumed are billed in per million request increments. You pay only for the storage and I/Os that your Neptune database consumes, and you don't need to provision in advance.

For pricing information, see the Neptune product page.

Instance Scaling in Neptune

You can scale your Neptune DB cluster as needed by modifying the DB instance class for each DB instance in the DB cluster. Neptune supports several optimized DB instance classes.

Read Scaling in Neptune

You can achieve read scaling for your Neptune DB cluster by creating up to 15 Neptune replicas in the DB cluster. Each Neptune replica returns the same data from the cluster volume with minimal replica lag (often considerably less than 100 milliseconds after the primary instance has written an update). As your read traffic increases, you can create additional Neptune replicas and connect to them directly to distribute the read load for your DB cluster. Neptune replicas don't have to be of the same DB instance class as the primary instance.

For information about adding Neptune replicas to a DB cluster, see Adding reader instances.

Auto-scaling the number of replicas in an Amazon Neptune DB cluster

You can use Neptune auto-scaling to automatically adjust the number of Neptune replicas in a DB cluster to meet your connectivity and workload requirements. Auto-scaling lets your Neptune DB cluster handle increases in workload, and then, when the workload decreases, auto-scaling removes unnecessary replicas so you aren't paying for unused capacity.

You can only use auto-scaling with a Neptune DB cluster that already has one primary writer instance and at least one read-replica instance (see Amazon Neptune DB Clusters and Instances). Also, all read-replica instances in the cluster must be in an available state. If any read-replica is in a state other than available, Neptune autoscaling does nothing until every read-replica in the cluster is available.

See Create Neptune cluster if you need to create a new cluster.

Using the AWS CLI, you define and apply a scaling policy to the DB cluster. You can also use the AWS CLI to edit or delete your auto-scaling policy. The policy specifies the following auto-scaling parameters:

- The minimum and maximum number of replicas to have in the cluster.
- A `ScaleOutCooldown` interval between replica(s)-addition scaling activity, and a `ScaleInCooldown` interval between replica(s)-deletion scaling activity.
- The CloudWatch metric and the metric trigger value for scaling up or down.

The frequency of Neptune auto-scaling actions is damped down in several ways:

- Initially, for auto-scaling to add or delete a reader, the `CPUUtilization` high alarm has to be breached for at least 3 minutes or the low alarm has to be breached for at least 15 minutes.
- After that first addition or deletion, the frequency of subsequent Neptune auto-scaling actions is limited by the `ScaleOutCooldown` and `ScaleInCooldown` settings in the autoscaling policy.

If the CloudWatch metric you're using reaches the high threshold you specified in your policy, and if the `ScaleOutCooldown` interval has elapsed since the last auto-scaling action, and if your DB cluster doesn't already have the maximum number of replicas that you set, Neptune auto-scaling creates a new replica using the same instance type as the DB cluster's primary instance.

Similarly, if the metric reaches the low threshold you specified and if the `ScaleInCooldown` interval has elapsed since the last auto-scaling action, and if your DB cluster has more than the minimum number of replicas that you specified, Neptune auto-scaling deletes one of the replicas.

> **Note**
>
> Neptune auto-scaling only removes replicas that it created. It does not remove pre-existing replicas.

> **Note**
>
> Read-replica auto-scaling removes readers through the Neptune API, so cluster deletion protection does not prevent a scale-in activity from removing a read replica. Deletion protection blocks deletion of the DB cluster itself, not the removal of individual read replicas. The `min-capacity` of your scaling policy determines the minimum number of readers that auto-scaling maintains. For more information about deletion protection, see How deletion protection affects DB instance deletion.

Using the neptune_autoscaling_config DB cluster parameter, you can also specify the instance type of the new read-replicas that Neptune auto-scaling creates, the maintenance windows for those read-replicas, and tags to be associated with each of the new read-replicas. You provide these configuration settings in a JSON string as the value of the `neptune_autoscaling_config` parameter, like this:

```
"{
  \"tags\": [
    { \"key\" : \"reader tag-0 key\", \"value\" : \"reader tag-0 value\" },
    { \"key\" : \"reader tag-1 key\", \"value\" : \"reader tag-1 value\" },
  ],
  \"maintenanceWindow\" : \"wed:12:03-wed:12:33\",
  \"dbInstanceClass\" : \"db.r5.xlarge\"
}"
```

Note that the quotation marks in the JSON string must all be escaped with a backslash character (\). All whitespace in the string is optional, as usual.

Any of the three configuration settings not specified in the `neptune_autoscaling_config` parameter are copied from the configuration of the DB cluster's primary writer instance.

When auto-scaling adds a new read-replica instance, it prefixes the DB instance ID with `autoscaled-reader` (for example, `autoscaled-reader-7r7t7z3lbd-20210828`). It also adds a tag to every read-replica that it creates with the key `autoscaled-reader` and a value of TRUE. You can see this tag on the **Tags** tab of the DB instance detail page in the AWS Management Console.

```
"key" : "autoscaled-reader",  "value" : "TRUE"
```

The promotion tier of all the read-replica instances created by auto-scaling is the lowest priority, which is 15 by default. This means that during a failover, any replica a higher priority, such as one that was created manually, would be promoted first. See Fault tolerance for a Neptune DB cluster.

Neptune auto-scaling is implemented using Application Auto Scaling with a target tracking scaling policy that uses a Neptune `CPUUtilization` CloudWatch metric as a predefined metric.

Using auto-scaling in a Neptune serverless DB cluster

Neptune Serverless responds much more rapidly than Neptune auto-scaling when demand exceeds an instance's capacity, and scales the instance up instead of adding another instance. Where auto-scaling is designed to match relatively stable increases or decreases in workload, serverless excels at handling rapid spikes and jitters in demand.

Understanding their strengths, you can combine auto-scaling and serverless to create a flexible infrastructure that will handle changes in your workload efficiently and meet demand while minimizing cost.

To allow auto-scaling to work effectively together with serverless, it's important to configure your serverless cluster's `maxNCU` setting high enough to accomodate spikes and brief changes in demand. Otherwise, transient changes don't trigger serverless scaling, which can cause auto-scaling to spin up many unnecessary additional instances. If `maxNCU` is set high enough, serverless scaling can handle those changes faster and less expensively.

How to enable auto-scaling for Amazon Neptune

Auto-scaling can only be enabled for a Neptune DB cluster using the AWS CLI. You cannot enable auto-scaling using the AWS Management Console.

Neptune autoscaling is supported in the following AWS Regions:

- US East (N. Virginia) — `us-east-1`
- US East (Ohio) — `us-east-2`
- US West (N. California) — `us-west-1`
- US West (Oregon) — `us-west-2`
- Asia Pacific (Hong Kong) — `ap-east-1`
- Asia Pacific (Tokyo) — `ap-northeast-1`
- Asia Pacific (Seoul) — `ap-northeast-2`
- Asia Pacific (Mumbai) — `ap-south-1`
- Asia Pacific (Singapore) — `ap-southeast-1`
- Asia Pacific (Sydney) — `ap-southeast-2`
- Canada (Central) — `ca-central-1`
- China (Beijing) — `cn-north-1`
- China (Ningxia) — `cn-northwest-1`
- Europe (Frankfurt) — `eu-central-1`
- Europe (Stockholm) — `eu-north-1`
- Europe (Ireland) — `eu-west-1`
- Europe (London) — `eu-west-2`
- Europe (Paris) — `eu-west-3`
- Middle East (Bahrain) — `me-south-1`
- South America (São Paulo) — `sa-east-1`

Enabling auto-scaling for a Neptune DB cluster involves three steps:

1. Register your DB cluster with Application Auto Scaling

The first step in enabling auto-scaling for a Neptune DB cluster is to register the cluster with Application Auto Scaling, using the AWS CLI or one of the Application Auto Scaling SDKs. The cluster must already have one primary instance and at least one read-replica instance:

For example, to register a cluster to be auto-scaled with from one to eight additional replicas, you could use the AWS CLI `register-scalable-target` command as follows:

```
aws application-autoscaling register-scalable-target \
  --service-namespace neptune \
  --resource-id cluster:(your DB cluster name) \
  --scalable-dimension neptune:cluster:ReadReplicaCount \
  --min-capacity 1 \
  --max-capacity 8
```

This is equivalent to using the the `RegisterScalableTarget` Application Auto Scaling API operation.

The AWS CLI `register-scalable-target` command takes the following parameters:

- **`service-namespace`** – Set to `neptune`.

  This parameter is equivalent to the `ServiceNamespace` parameter in the Application Auto Scaling API.

- **`resource-id`** – Set this to the resource identifier for your Neptune DB cluster. The resource type is `cluster`, which is followed by a colon (`':'`), and then the name of your DB cluster.

  This parameter is equivalent to the `ResourceID` parameter in the Application Auto Scaling API.

- **`scalable-dimension`** – The scalable dimension in this case is the number of replica instances in the DB cluster, so you set this parameter to `neptune:cluster:ReadReplicaCount`.

  This parameter is equivalent to the `ScalableDimension` parameter in the Application Auto Scaling API.

- **`min-capacity`** – The minimum number of reader DB replica instances to be managed by Application Auto Scaling. This value should be set in the range from 0 to 15, and must be equal to or less than the value specified for the maximum number of Neptune Replicas in `max-capacity`. There must be at least one reader in the DB cluster for auto-scaling to work.

  This parameter is equivalent to the `MinCapacity` parameter in the Application Auto Scaling API.

- **`max-capacity`** – The maximum number of reader DB replica instances in the DB cluster, including pre-existing instances and new instances managed by Application Auto Scaling. This value must be set in the range from 0 to 15, and must be equal to or greater than the value specified for the minimum number of Neptune Replicas in `min-capacity`.

The `max-capacity` AWS CLI parameter is equivalent to the `MaxCapacity` parameter in the Application Auto Scaling API.

When you register your DB cluster, Application Auto Scaling creates an `AWSServiceRoleForApplicationAutoScaling_NeptuneCluster` service-linked role. For more information, see Service-linked roles for Application auto-scaling in the *Application Auto Scaling User Guide*.

2. Define an autoscaling policy to use with your DB cluster

A target-tracking scaling policy is defined as a JSON text object that can also be saved in a text file. For Neptune this policy currently can only use the Neptune `CPUUtilization` CloudWatch metric as a predefined metric named `NeptuneReaderAverageCPUUtilization`.

Here is an example target tracking scaling configuration policy for Neptune:

```
{
  "PredefinedMetricSpecification": { "PredefinedMetricType":
 "NeptuneReaderAverageCPUUtilization" },
  "TargetValue": 60.0,
  "ScaleOutCooldown" : 600,
  "ScaleInCooldown" : 600
}
```

The **`TargetValue`** element here contains the percentage of CPU utilization above which auto-scaling *scales out* (that is, adds more replicas) and below which it *scales in* (that is, deletes replicas). In this case, the target percentage that triggers scaling is `60.0`%.

The **`ScaleInCooldown`** element specifies the amount of time, in seconds, after a scale-in activity completes before another scale-in can start. The default is 300 seconds. Here, the value of 600 specifies that at least ten minutes must elapse between the completion of one replica deletion and the start of another one.

The **`ScaleOutCooldown`** element specifies the amount of time, in seconds, after a scale-out activity completes before another scale-out can start. The default is 300 seconds. Here, the value of 600 specifies that at least ten minutes must elapse between the completion of one replica addition and the start of another one.

The **`DisableScaleIn`** element is a Boolean that if present and set to `true` disables scale-in entirely, meaning that auto-scaling may add replicas but will never remove any. By default, scale-in is enabled, and `DisableScaleIn` is `false`.

After registering your Neptune DB cluster with Application Auto Scaling and defining a JSON scaling policy in a text file, next apply the scaling policy to the registered DB cluster. You can use the AWS CLI `put-scaling-policy` command to do this, with parameters like the following:

```
aws application-autoscaling put-scaling-policy \
  --policy-name (name of the scaling policy) \
  --policy-type TargetTrackingScaling \
  --resource-id cluster:(name of your Neptune DB cluster) \
  --service-namespace neptune \
  --scalable-dimension neptune:cluster:ReadReplicaCount \
  --target-tracking-scaling-policy-configuration file://(path to the JSON configuration
 file)
```

When you have applied the auto-scaling policy, auto-scaling is enabled on your DB cluster.

You can also use the AWS CLI `put-scaling-policy` command to update an existing auto-scaling policy.

See also PutScalingPolicy in the *Application Auto Scaling API Reference*.

Removing auto-scaling from a Neptune DB cluster

To remove auto-scaling from a Neptune DB cluster, use the AWS CLI delete-scaling-policy and deregister-scalable-target commands.

Maintaining your Amazon Neptune DB Cluster

Neptune performs maintenance periodically on all the resources it uses, including:

- **Replacing the underlying hardware as necessary.** This happens in the background, without your having to take any action, and generally has no affect on your operations.
- **Updating the underlying operating system.** Operating system upgrades of the instances in your DB cluster are undertaken to improve performance and security, so you should generally complete them as soon as possible. Typically, the updates take about 10 minutes. Operating system updates don't change the DB engine version or DB instance class of a DB instance.

  It's generally best to update the reader instances in a DB cluster first, and then the writer instance. Updating the readers and the writer at the same time can cause downtime in the event of a failover. Note that DB instances are not automatically backed up before an operating-system update, so be sure to make manual backups before you apply an operating-system update.
- **Updating the Neptune database engine.** Neptune regularly releases a variety of engine updates to introduce new features and improvements and to fix bugs.

Engine version numbers

Version numbering before engine release 1.3.0.0

Before November 2019, Neptune only supported one engine version at a time, and engine version numbers all took the form, `1.0.1.0.200`*`<xxx>`*, where `xxx` was the patch number. All new engine versions were released as patches to earlier versions.

In November 2019, Neptune started supporting multiple versions, allowing customers better control over their upgrade paths. As a result, engine release numbering changed.

From November 2019 up until engine release 1.3.0.0, engine version numbers have 5 parts. Take version number `1.0.2.0.R2` as an example:

- The first part was always 1.
- The second part, `0` in `1.0.2.0.R2`), was the database major version number.
- The third and fourth parts, `2.0` in `1.0.2.0.R2`) were both minor version numbers.
- The fifth part (`R2` in `1.0.2.0.R2`) was the patch number.

Most updates were patch updates, and the distinction between patches and minor version updates was not always clear.

Version numbering from engine release 1.3.0.0 on

Starting with engine release 1.3.0.0, Neptune changed the way engine updates are numbered and managed.

Engine version numbers now have four parts, each of which corresponds to a type of release, as follows:

```
product-version.major-version.minor-version.patch-version
```

Non-breaking changes of the sort that were previously released as patches are now released as minor versions that you can manage using the `AutoMinorVersionUpgrade` instance setting.

This means that if you want you can receive a notification every time a new minor version is released, by subscribing to the `RDS-EVENT-0156` event (see Subscribing to Neptune event notification).

Patch releases are now reserved for urgent targeted fixes, and are numbered using the last part of the version number (`*.*.*.1`, `*.*.*.2`, and so forth).

Different types of engine release in Amazon Neptune

The four types of engine release that correspond to the four parts of an engine version number are as follows:

- **Product version** – This only changes if the product undergoes sweeping, fundamental changes in functionality or interface. The current Neptune product version is 1.
- **Major version** – Major versions introduce important new features and breaking changes, and generally have a useful lifespan of at least two years.
- **Minor version** – Minor versions can contain new features, improvements, and bug fixes but do not contain any breaking changes. You can choose whether or not to have them applied them automatically during the next maintenance window, and you can also choose to be notified whenever one is released.
- **Patch version** – Patch versions are released only to address urgent bug fixes or critical security updates. They seldom contain breaking changes, and they are automatically applied during the next maintenance window following their release.

Amazon Neptune major version updates

A major version update generally introduces one or more important new features and often contains breaking changes. It usually has a support lifetime of around two years. Neptune major versions are listed in Engine releases, along with the date they were released and their estimated end of life.

Major version updates are entirely optional until the major version that you're using reaches its end of life. If you do choose to upgrade to a new major version, you must install the new version yourself using the AWS CLI or the Neptune console as described in Major version upgrades.

If the major version that you're using reaches its end of life, however, you'll be notified that you are required to upgrade to a more recent major version. Then, if you don't upgrade within a grace period after the notification, an upgrade to the most recent major version is automatically scheduled to occur during the next maintenance window. See Engine version life-spans for more information.

Amazon Neptune minor version updates

Most Neptune engine updates are minor version updates. They happen quite frequently and do not contain breaking changes.

If you have the `AutoMinorVersionUpgrade` field set to `true` in the writer (primary) instance of your DB cluster, minor version updates are applied automatically to all instances in your DB cluster during the next maintenance window after they are released.

If you have the `AutoMinorVersionUpgrade` field set to `false` in the writer instance of your DB cluster, they are applied only if you explicitly install them.

> **Note**
>
> Minor version updates are self-contained (they don't depend on previous minor version updates to the same major version), and cumulative (they contain all the features and fixes introduced in previous minor version updates). This means that you can install any given minor version update whether or not you have installed previous ones.

It's easy to keep track of minor-version releases by subscribing to the `RDS-EVENT-0156` event (see Subscribing to Neptune event notification). You will then be notified every time a new minor version is released.

Also, whether or not you subscribe to notifications, you can always check to see what updates are pending.

Amazon Neptune patch version updates

In the case of security issues or other serious defects that affect instance reliability, Neptune deploys mandatory patches. They are applied to all instances in your DB cluster during your next maintenance window without any intervention on your part.

A patch release is only deployed when the risks of not deploying it outweigh any risks and downtime associated with deploying it. Patch releases happen infrequently (typically once every few months) and seldom require more than a fraction of your maintenance window to apply.

Planning for Amazon Neptune major engine version life-span

Neptune engine versions almost always reach their end of life at the end of a calendar quarter. Exceptions occur only when important security or availability issues arise.

When an engine version reaches its end of life, you will be required to upgrade your Neptune database to a newer version.

In general, Neptune engine versions continue to be available as follows:

- **Minor engine versions:** Minor engine versions remain available for at least 6 months following their release.
- **Major engine versions:** Major engine versions remain available for at least 12 months following their release.

At least 3 months before an engine version reaches its end of life, AWS will send an automated email notification to the email address associated with your AWS account and post the same message to your AWS Health Dashboard. This will give you time to plan and prepare to upgrade.

When an engine version reaches its end of life, you will no longer be able to create new clusters or instances using that version, nor will autoscaling be able to create instances using that version.

An engine version that actually reaches its end of life will automatically be upgraded during a maintenance window. The message sent to you 3 months before the engine version's end of life will contain details about what this automatic update would involve, including the version to

which you would be automatically upgraded, the impact on your DB clusters, and actions that we recommend.

> **Important**
>
> You are responsible for keeping your database engine versions current. AWS urges all customers to upgrade their databases to the latest engine version in order to benefit from the most current security, privacy, and availability safeguards. If you operate your database on an unsupported engine or software past the deprecation date ("Legacy Engine"), you face a greater likelihood of security, privacy, and operational risks, including downtime events.
>
> Operation of your database on any engine is subject to the Agreement governing your use of the AWS Services. Legacy Engines are not Generally Available. AWS no longer provides support for the Legacy Engine, and AWS may place limits on the access to or use of any Legacy Engine at any time, if AWS determines the Legacy Engine poses a security or liability risk, or a risk of harm, to the Services, AWS, its Affiliates, or any third party. Your decision to continue running Your Content in a Legacy Engine could result in Your Content becoming unavailable, corrupted, or unrecoverable. Databases running on a Legacy Engine are subject to Service Level Agreement (SLA) Exceptions.
>
> DATABASES AND RELATED SOFTWARE RUNNING ON A LEGACY ENGINE CONTAIN BUGS, ERRORS, DEFECTS, AND/OR HARMFUL COMPONENTS. ACCORDINGLY, AND NOTWITHSTANDING ANYTHING TO THE CONTRARY IN THE AGREEMENT OR THE SERVICE TERMS, AWS IS PROVIDING THE LEGACY ENGINE "AS IS."

Managing engine updates to your Neptune DB cluster

> **Note**
>
> Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you experience downtime ranging from 20 or 30 seconds to several minutes, after which you can resume using the DB cluster. On rare occasions a Multi-AZ failover might be required for a maintenance update on an instance to complete.
>
> For major version upgrades that can take longer to apply, you can use a blue-green deployment strategy to minimize downtime.

Determining which engine version you are currently using

You can use the AWS CLI `get-engine-status` command to check which engine release version your DB cluster is currently using:

```
aws neptunedata get-engine-status
```

The JSON output includes a `"dbEngineVersion"` field like this:

```
  "dbEngineVersion": "1.3.0.0",
```

Check to see what updates are pending and available

You can check pending updates to your DB cluster using the Neptune console. Select **Databases** in the left column and then select your DB cluster in the databases pane. Pending updates are listed in the **Maintenance** column. If you select **Actions** and then **Maintenance**, you have three choices about what to do:

- Upgrade now.
- Upgrade at next window.
- Defer upgrade.

You can list pending engine updates using the AWS CLI as follows:

```
aws neptune describe-pending-maintenance-actions \
  --resource-identifier (ARN of your DB cluster)
  --region (your region) \
  --engine neptune
```

You can also list available engine updates using the AWS CLI as follows:

```
aws neptune describe-db-engine-versions \
  --region (your region) \
  --engine neptune
```

The list of available engine releases includes only those releases that have a version number higher than the current one and for which an upgrade path is defined.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. A minor upgrade could introduce new features or behavior that would affect your code even without any breaking change.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to use the Neptune Blue-Green deployment solution. That way you can run applications and queries on the new version without affecting your production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

Neptune Maintenance Window

The weekly maintenance window is a 30-minute period during which scheduled engine updates and other system changes are applied. Most maintenance events complete during the 30-minute window, although larger maintenance events might sometimes longer to complete.

Every DB cluster has a weekly 30-minute maintenance window. If you don't specify a preferred time for it when you create the DB cluster, Neptune randomly picks a day of the week and then randomly assigns a 30-minute period within it from an 8-hour block of time that varies with the region.

Here, for example, are the 8-hour time blocks for maintence windows used in several AWS regions:

Region	Time Block
US West (Oregon) Region	06:00–14:00 UTC
US West (N. California) Region	06:00–14:00 UTC
US East (Ohio) Region	03:00–11:00 UTC
Europe (Ireland) Region	22:00–06:00 UTC

The maintenance window determines when pending operations start, and most maintenance operations complete within the window, but larger maintenance tasks can continue beyond the window's end time.

Moving your DB cluster maintenance window

Ideally, the your maintenance window should fall at a time when you cluster is at its lowest usage. If that isn't true of your current window, you can move it to a better time, like this:

To change your DB cluster maintenance window

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **databases**.
3. Choose the DB cluster for which you want to change the maintenance window.
4. Choose **Modify**.
5. Choose **Show more** at the bottom of the **Modify cluster** page.
6. In the **Preferred maintenance window** section, set the day, time and duration of the maintenance window as you prefer.
7. Choose **Next**.

   On the confirmation page, review your changes.
8. To apply the changes to the maintenance window immediately, select **Apply immediately**.
9. Choose **Submit** to apply your changes.

   To edit your changes, choose **Previous**, or to cancel your changes, choose **Cancel**.

Using AutoMinorVersionUpgrade to control automatic minor version updates

> **Important**
>
> `AutoMinorVersionUpgrade` is only effective for minor version upgrades above engine release 1.3.0.0.
>
> Automatic minor version upgrades don't apply to Neptune clusters that are part of a global database. You can set this option on a DB instance in a global database cluster, but it has no effect.

If you have the `AutoMinorVersionUpgrade` field set to `true` in the writer (primary) instance of your DB cluster, minor version updates are applied automatically to all instances in your DB cluster during the next maintenance window after they are released.

If you have the `AutoMinorVersionUpgrade` field set to `false` in the writer instance of your DB cluster, they are applied only if you explicitly install them.

> **Note**
>
> Patch releases (`*.*.*.1`, `*.*.*.2`, etc.) are always installed automatically during your next maintenance window, regardless of how the `AutoMinorVersionUpgrade` parameter is set.

You can set `AutoMinorVersionUpgrade` using the AWS Management Console as follows:

To set `AutoMinorVersionUpgrade` using the Neptune console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Databases**.
3. Choose the primary (writer) instance of the DB cluster for which you want to set `AutoMinorVersionUpgrade`.
4. Choose **Modify**.
5. Choose **Show more** at the bottom of the **Modify cluster** page.
6. At the bottom of the expanded page, choose either **Turn on auto minor version upgrade** or **Turn off auto minor version upgrade**.

7. Choose **Next**.

   On the confirmation page, review your changes.
8. To apply the changes to auto minor version upgrade, select **Apply immediately**.
9. Choose **Submit** to apply your changes.

   To edit your changes, choose **Previous**, or to cancel your changes, choose **Cancel**.

You can also use the AWS CLI to set the `AutoMinorVersionUpgrade` field. For example, to set it to `true`, you can use a command like this:

```
aws neptune modify-db-instance \
  --db-instance-identifier (the ID of your cluster's writer instance) \
  --auto-minor-version-upgrade \
  --apply-immediately
```

Similarly, to set it to `false`, use a command like this:

```
aws neptune modify-db-instance \
  --db-instance-identifier (the ID of your cluster's writer instance) \
  --no-auto-minor-version-upgrade \
  --apply-immediately
```

Installing updates to your Neptune engine manually

Installing a major version engine upgrade

Major engine releases must always be installed manually. To minimize downtime and provide for plenty of time for testing and validation, the best way to install a new major version is generally to use the Neptune Blue-Green deployment solution.

In some cases you can also use the CloudFormation template with which you created your DB cluster to install a major version upgrade (see Using a CloudFormation template to update the engine version of your Neptune DB Cluster).

If you want to install a major version update immediately, you can use a CLI command like the following:

```
aws neptune modify-db-cluster \
  --db-cluster-identifier (identifier for your neptune cluster) \
```

```
  --engine neptune \
  --engine-version (the new engine version) \
  --apply-immediately
```

Be sure to specify the engine version to which you want to upgrade. If you don't, your engine may be upgraded to a version that is not the most recent one or the one you expect.

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`.

If your cluster uses a custom cluster parameter group, be sure to specify using this paramater:

```
  --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to specify it using this parameter:

```
  ---db-instance-parameter-group-name (name of the custom instance parameter group)
```

Installing a minor version engine upgrade using the AWS Management Console

To perform a minor version upgrade using the Neptune console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Databases**, and then choose the DB cluster that you want to modify.
3. Choose **Modify**.
4. Under **Instance specifications**, choose the new version to which you want to upgrade.
5. Choose **Next**.
6. If you want to apply the changes immediately, choose **Apply immediately**.
7. Choose **Submit** to update your DB cluster.

Installing a minor version engine upgrade using the AWS CLI

You can use a command like the following to perform a minor version upgrade without waiting for the next maintenance window:

```
aws neptune modify-db-cluster \
```

```
  --db-cluster-identifier (your-neptune-cluster) \
  --engine-version (new-engine-version) \
  --apply-immediately
```

If you are manually upgrading using the AWS CLI, be sure to include the engine version to which you want to upgrade. If you do not, your engine may be upgraded to a version that is not the most recent one or the one you expect.

Upgrading to engine version 1.2.0.0 or above from a version earlier than 1.2.0.0

Engine release 1.2.0.0 introduced several significant changes that can make upgrading from an earlier version more complicated than usual:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 introduced a new format for undo logs. Consequently, if you are upgrading to version 1.2.0.0 or higher from a version earlier than 1.2.0.0, the `UndoLogListSize` metric must be below a certain threshold. Otherwise, the patch will roll back and fail. The thresholds are based on instance type: the default limit is 40k for 4xlarge or larger instances, and 10k for instances smaller than 4xlarge. If the `UndoLogListSize` exceeds the limit when you attempt to upgrade, the patch process will roll back, the upgrade will be canceled, and an event with the reason will be visible on the cluster event page. These limits can change for operational reasons without prior warning.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can help bring down the `UndoLogListSize` below the applicable threshold. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

  If the `UndoLogListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down below the required limit.
- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a

resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Using a CloudFormation template to update the engine version of your Neptune DB Cluster

You can re-use the Neptune CloudFormation template that you used to create your Neptune DB Cluster to update its engine version.

Neptune engine version upgrades can be minor or major. Using an CloudFormation template can help with major version upgrades, which often contain significant changes. Since major version upgrades can contain database changes that are not backward-compatible with existing applications, you may also need to make changes to your applications when upgrading. Always test before upgrading, and we strongly recommend that you always create a manual snapshot of your DB cluster before upgrading.

Note that you have to do a separate engine upgrade for each major version. You can't skip a major version and upgrade directly to the major version following.

Prior to May 17, 2023, if you used the Neptune CloudFormation stack to upgrade your engine version, it simply created a new, empty DB cluster in place your current one. As of May 17, 2023, however, the Neptune CloudFormation stack now supports in-place engine upgrades that preserve your existing data.

> **Note**
>
> If you are using the AWS Cloud Development Kit (AWS CDK), ensure that the AWS CDK version being used is 2.82.0 or later. Versions prior to 2.82.0 do not support in-place Neptune engine upgrades.

For a major version upgrade, your template should set the following properties in `DBCluster`:

- `DBClusterParameterGroup` (Custom or Default)
- `DBInstanceParameterGroupName`
- `EngineVersion`

Similarly, for DBInstances attached to DBCluster you should set:

- `DBParameterGroup` (Custom/Default)

Make sure that all your parameter groups are defined in the template, whether they are default or custom.

In the case of a custom parameter group, make sure that the family of your existing custom parameter group is compatible with the new engine version. Engine versions earlier than 1.2.0.0 used parameter group family `neptune1`, whereas engine releases from 1.2.0.0 forward require parameter group family `neptune1.2`. See Amazon Neptune parameter groups for more information.

For major engine version upgrades, specify a parameter group with the appropriate family in the `DBCluster` `DBInstanceParameterGroupName` field.

A default parameter group should be upgraded to one that is compatible with the new engine version.

Note that Neptune automatically reboots DB instances after an engine upgrade.

Topics

- Example: Minor engine upgrade from 1.2.0.1 to 1.2.0.2
- Example: Major version upgrade from 1.1.1.0 to 1.2.0.2 with default parameter groups
- Example: Major version upgrade from 1.1.1.0 to 1.2.0.2 with custom parameter groups
- Example: Major version upgrade from 1.1.1.0 to 1.2.0.2 with a mix of default and custom parameter groups

Example: Minor engine upgrade from 1.2.0.1 to 1.2.0.2

Find the DB cluster that you want to upgrade, and the template you used to create it. For example:

```
Description: Base Template to create Neptune Stack with Engine Version 1.2.0.1 using
 custom Parameter Groups
Parameters:
  DbInstanceType:
    Description: Neptune DB instance type
    Type: String
    Default: db.r5.large
Resources:
  NeptuneDBClusterParameterGroup:
    Type: 'AWS::Neptune::DBClusterParameterGroup'
    Properties:
```

```
      Family: neptune1.2
      Description: test-cfn-neptune-db-cluster-parameter-group-description
      Parameters:
        neptune_enable_audit_log: 0
  NeptuneDBParameterGroup:
    Type: 'AWS::Neptune::DBParameterGroup'
    Properties:
      Family: neptune1.2
      Description: test-cfn-neptune-db-parameter-group-description
      Parameters:
        neptune_query_timeout: 20000
  NeptuneDBCluster:
    Type: 'AWS::Neptune::DBCluster'
    Properties:
      EngineVersion: 1.2.0.1
      DBClusterParameterGroupName:
        Ref: NeptuneDBClusterParameterGroup
    DependsOn:
      - NeptuneDBClusterParameterGroup
  NeptuneDBInstance:
    Type: 'AWS::Neptune::DBInstance'
    Properties:
      DBClusterIdentifier:
        Ref: NeptuneDBCluster
      DBInstanceClass:
        Ref: DbInstanceType
      DBParameterGroupName:
        Ref: NeptuneDBParameterGroup
    DependsOn:
      - NeptuneDBCluster
      - NeptuneDBParameterGroup
Outputs:
  DBClusterId:
    Description: Neptune Cluster Identifier
    Value:
      Ref: NeptuneDBCluster
```

Update the `EngineVersion` property from `1.2.0.1` to `1.2.0.2`:

```
Description: Template to upgrade minor engine version to 1.2.0.2
Parameters:
  DbInstanceType:
    Description: Neptune DB instance type
```

```
    Type: String
    Default: db.r5.large
Resources:
  NeptuneDBClusterParameterGroup:
    Type: 'AWS::Neptune::DBClusterParameterGroup'
    Properties:
      Family: neptune1.2
      Description: test-cfn-neptune-db-cluster-parameter-group-description
      Parameters:
        neptune_enable_audit_log: 0
  NeptuneDBParameterGroup:
    Type: 'AWS::Neptune::DBParameterGroup'
    Properties:
      Family: neptune1.2
      Description: test-cfn-neptune-db-parameter-group-description
      Parameters:
        neptune_query_timeout: 20000
  NeptuneDBCluster:
    Type: 'AWS::Neptune::DBCluster'
    Properties:
      EngineVersion: 1.2.0.2
      DBClusterParameterGroupName:
        Ref: NeptuneDBClusterParameterGroup
    DependsOn:
      - NeptuneDBClusterParameterGroup
  NeptuneDBInstance:
    Type: 'AWS::Neptune::DBInstance'
    Properties:
      DBClusterIdentifier:
        Ref: NeptuneDBCluster
      DBInstanceClass:
        Ref: DbInstanceType
      DBParameterGroupName:
        Ref: NeptuneDBParameterGroup
    DependsOn:
      - NeptuneDBCluster
      - NeptuneDBParameterGroup
Outputs:
  DBClusterId:
    Description: Neptune Cluster Identifier
    Value:
      Ref: NeptuneDBCluster
```

Now use CloudFormation to run the revised template.

Example: Major version upgrade from 1.1.1.0 to 1.2.0.2 with default parameter groups

Find the `DBCluster` that you want to upgrade, and the template you used to create it. For example:

```
Description: Base Template to create Neptune Stack with Engine Version 1.1.1.0 using
 default Parameter Groups
Parameters:
  DbInstanceType:
    Description: Neptune DB instance type
    Type: String
    Default: db.r5.large
Resources:
  NeptuneDBCluster:
    Type: 'AWS::Neptune::DBCluster'
    Properties:
      EngineVersion: 1.1.1.0
  NeptuneDBInstance:
    Type: 'AWS::Neptune::DBInstance'
    Properties:
      DBClusterIdentifier:
        Ref: NeptuneDBCluster
      DBInstanceClass:
        Ref: DbInstanceType
    DependsOn:
      - NeptuneDBCluster
Outputs:
  DBClusterId:
    Description: Neptune Cluster Identifier
    Value:
      Ref: NeptuneDBCluster
```

- Update the default `DBClusterParameterGroup` to the one in the parameter group family used by the new engine version (here `default.neptune1.2`).
- For each `DBInstance` attached to the `DBCluster`, update the default `DBParameterGroup` to the one in the family used by new engine version (here `default.neptune1.2`).
- Set the `DBInstanceParameterGroupName` property to the default parameter group in that family (here `default.neptune1.2`).

- Update the `EngineVersion` property from `1.1.0.0` to `1.2.0.2`.

The template should look like this:

```
Description: Template to upgrade major engine version to 1.2.0.2 by using upgraded
 default parameter groups
Parameters:
  DbInstanceType:
    Description: Neptune DB instance type
    Type: String
    Default: db.r5.large
Resources:
  NeptuneDBCluster:
    Type: 'AWS::Neptune::DBCluster'
    Properties:
      EngineVersion: 1.2.0.2
      DBClusterParameterGroupName: default.neptune1.2
      DBInstanceParameterGroupName: default.neptune1.2
  NeptuneDBInstance:
    Type: 'AWS::Neptune::DBInstance'
    Properties:
      DBClusterIdentifier:
        Ref: NeptuneDBCluster
      DBInstanceClass:
        Ref: DbInstanceType
      DBParameterGroupName: default.neptune1.2
    DependsOn:
      - NeptuneDBCluster
Outputs:
  DBClusterId:
    Description: Neptune Cluster Identifier
    Value:
```

Now use CloudFormation to run the revised template.

Example: Major version upgrade from 1.1.1.0 to 1.2.0.2 with custom parameter groups

Find the `DBCluster` that you want to upgrade, and the template you used to create it. For example:

```
Description: Base Template to create Neptune Stack with Engine Version 1.1.1.0 using
 custom Parameter Groups
Parameters:
  DbInstanceType:
    Description: Neptune DB instance type
    Type: String
    Default: db.r5.large
Resources:
  NeptuneDBClusterParameterGroup:
    Type: 'AWS::Neptune::DBClusterParameterGroup'
    Properties:
      Name: engineupgradetestcpg
      Family: neptune1
      Description: 'NeptuneDBClusterParameterGroup with family neptune1'
      Parameters:
        neptune_enable_audit_log: 0
  NeptuneDBParameterGroup:
    Type: 'AWS::Neptune::DBParameterGroup'
    Properties:
      Name: engineupgradetestpg
      Family: neptune1
      Description: 'NeptuneDBParameterGroup1 with family neptune1'
      Parameters:
        neptune_query_timeout: 20000
  NeptuneDBCluster:
    Type: 'AWS::Neptune::DBCluster'
    Properties:
      EngineVersion: 1.1.1.0
      DBClusterParameterGroupName:
        Ref: NeptuneDBClusterParameterGroup
    DependsOn:
      - NeptuneDBClusterParameterGroup
  NeptuneDBInstance:
    Type: 'AWS::Neptune::DBInstance'
    Properties:
      DBClusterIdentifier:
        Ref: NeptuneDBCluster
      DBInstanceClass:
        Ref: DbInstanceType
      DBParameterGroupName:
        Ref: NeptuneDBParameterGroup
    DependsOn:
      - NeptuneDBCluster
```

```
      - NeptuneDBParameterGroup
Outputs:
  DBClusterId:
    Description: Neptune Cluster Identifier
    Value:
      Ref: NeptuneDBCluster
```

- Update the custom `DBClusterParameterGroup` family to the one used by the new engine version here `default.neptune1.2`).
- For each `DBInstance` attached to the `DBCluster`, update the custom `DBParameterGroup` family to the one used by the new engine version (here `default.neptune1.2`).
- Set the `DBInstanceParameterGroupName` property to the parameter group in that family (here `default.neptune1.2`).
- Update the `EngineVersion` property from `1.1.0.0` to `1.2.0.2`.

The template should look like this:

```
Description: Template to upgrade major engine version to 1.2.0.2 by modifying existing
 custom parameter groups
Parameters:
  DbInstanceType:
    Description: Neptune DB instance type
    Type: String
    Default: db.r5.large
Resources:
  NeptuneDBClusterParameterGroup:
    Type: 'AWS::Neptune::DBClusterParameterGroup'
    Properties:
      Name: engineupgradetestcpgnew
      Family: neptune1.2
      Description: 'NeptuneDBClusterParameterGroup with family neptune1.2'
      Parameters:
        neptune_enable_audit_log: 0
  NeptuneDBParameterGroup:
    Type: 'AWS::Neptune::DBParameterGroup'
    Properties:
      Name: engineupgradetestpgnew
      Family: neptune1.2
      Description: 'NeptuneDBParameterGroup1 with family neptune1.2'
      Parameters:
```

```
        neptune_query_timeout: 20000
  NeptuneDBCluster:
    Type: 'AWS::Neptune::DBCluster'
    Properties:
      EngineVersion: 1.2.0.2
      DBClusterParameterGroupName:
        Ref: NeptuneDBClusterParameterGroup
      DBInstanceParameterGroupName:
        Ref: NeptuneDBParameterGroup
    DependsOn:
      - NeptuneDBClusterParameterGroup
  NeptuneDBInstance:
    Type: 'AWS::Neptune::DBInstance'
    Properties:
      DBClusterIdentifier:
        Ref: NeptuneDBCluster
      DBInstanceClass:
        Ref: DbInstanceType
      DBParameterGroupName:
        Ref: NeptuneDBParameterGroup
    DependsOn:
      - NeptuneDBCluster
      - NeptuneDBParameterGroup
Outputs:
  DBClusterId:
    Description: Neptune Cluster Identifier
    Value:
      Ref: NeptuneDBCluster
```

Now use CloudFormation to run the revised template.

Example: Major version upgrade from 1.1.1.0 to 1.2.0.2 with a mix of default and custom parameter groups

Find the `DBCluster` that you want to upgrade, and the template you used to create it. For example:

```
Description: Base Template to create Neptune Stack with Engine Version 1.1.1.0 using
 custom Parameter Groups
Parameters:
  DbInstanceType:
    Description: Neptune DB instance type
```

```
    Type: String
    Default: db.r5.large
Resources:
  NeptuneDBClusterParameterGroup:
    Type: 'AWS::Neptune::DBClusterParameterGroup'
    Properties:
      Family: neptune1
      Description: 'NeptuneDBClusterParameterGroup with family neptune1'
      Parameters:
        neptune_enable_audit_log: 0
  NeptuneDBParameterGroup:
    Type: 'AWS::Neptune::DBParameterGroup'
    Properties:
      Family: neptune1
      Description: 'NeptuneDBParameterGroup with family neptune1'
      Parameters:
        neptune_query_timeout: 20000
  NeptuneDBCluster:
    Type: 'AWS::Neptune::DBCluster'
    Properties:
      EngineVersion: 1.1.1.0
      DBClusterParameterGroupName:
        Ref: NeptuneDBClusterParameterGroup
    DependsOn:
      - NeptuneDBClusterParameterGroup
  CustomNeptuneDBInstance:
    Type: 'AWS::Neptune::DBInstance'
    Properties:
      DBClusterIdentifier:
        Ref: NeptuneDBCluster
      DBInstanceClass:
        Ref: DbInstanceType
      DBParameterGroupName:
        Ref: NeptuneDBParameterGroup
    DependsOn:
      - NeptuneDBCluster
      - NeptuneDBParameterGroup
  DefaultNeptuneDBInstance:
    Type: 'AWS::Neptune::DBInstance'
    Properties:
      DBClusterIdentifier:
        Ref: NeptuneDBCluster
      DBInstanceClass:
        Ref: DbInstanceType
```

```
    DependsOn:
      - NeptuneDBCluster
Outputs:
  DBClusterId:
    Description: Neptune Cluster Identifier
    Value:
      Ref: NeptuneDBCluster
```

- For a custom cluster parameter group, update the `DBClusterParameterGroup` family to the one corresponding to new engine version, namely `neptune1.2`.
- For a default cluster parameter group, update the `DBClusterParameterGroup` to the default corresponding to new engine version, namely `default.neptune1.2`.
- For each `DBInstance` attached to the `DBCluster`, update a default `DBParameterGroup` to the one in the family used by new engine version (here `default.neptune1.2`), and a custom parameter group to one that uses the family supported by the new engine version (here `neptune1.2`).
- Set the `DBInstanceParameterGroupName` property to the parameter group in the family supported by the new engine version.

The template should look like this:

```
Description: Template to update Neptune Stack to Engine Version 1.2.0.1 using custom
 and default Parameter Groups
Parameters:
  DbInstanceType:
    Description: Neptune DB instance type
    Type: String
    Default: db.r5.large
Resources:
  NeptuneDBClusterParameterGroup:
    Type: 'AWS::Neptune::DBClusterParameterGroup'
    Properties:
      Family: neptune1.2
      Description: 'NeptuneDBClusterParameterGroup with family neptune1.2'
      Parameters:
        neptune_enable_audit_log: 0
  NeptuneDBParameterGroup:
    Type: 'AWS::Neptune::DBParameterGroup'
    Properties:
      Family: neptune1.2
```

```
      Description: 'NeptuneDBParameterGroup1 with family neptune1.2'
      Parameters:
        neptune_query_timeout: 20000
  NeptuneDBCluster:
    Type: 'AWS::Neptune::DBCluster'
    Properties:
      EngineVersion: 1.2.0.2
      DBClusterParameterGroupName:
        Ref: NeptuneDBClusterParameterGroup
      DBInstanceParameterGroupName: default.neptune1.2
    DependsOn:
      - NeptuneDBClusterParameterGroup
  CustomNeptuneDBInstance:
    Type: 'AWS::Neptune::DBInstance'
    Properties:
      DBClusterIdentifier:
        Ref: NeptuneDBCluster
      DBInstanceClass:
        Ref: DbInstanceType
      DBParameterGroupName:
        Ref: NeptuneDBParameterGroup
    DependsOn:
      - NeptuneDBCluster
      - NeptuneDBParameterGroup
  DefaultNeptuneDBInstance:
    Type: 'AWS::Neptune::DBInstance'
    Properties:
      DBClusterIdentifier:
        Ref: NeptuneDBCluster
      DBInstanceClass:
        Ref: DbInstanceType
      DBParameterGroupName: default.neptune1.2
    DependsOn:
      - NeptuneDBCluster
Outputs:
  DBClusterId:
    Description: Neptune Cluster Identifier
    Value:
      Ref: NeptuneDBCluster
```

Now use CloudFormation to run the revised template.

Database Cloning in Neptune

Using DB cloning, you can quickly and cost-effectively create clones of all your databases in Amazon Neptune. The clone databases require only minimal additional space when they are first created. Database cloning uses a *copy-on-write protocol*. Data is copied at the time that it changes, either on the source databases or the clone databases. You can make multiple clones from the same DB cluster. You can also create additional clones from other clones. For more information about how the copy-on-write protocol works in the context of Neptune storage, see Copy-on-Write Protocol.

You can use DB cloning in a variety of use cases, especially where you don't want to have an impact on your production environment, such as the following:

- Experiment with and assess the impact of changes, such as schema changes or parameter group changes.
- Perform workload-intensive operations, such as exporting data or running analytical queries.
- Create a copy of a production DB cluster in a non-production environment for development or testing.

To create a clone of a DB cluster using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Instances**. Choose the primary instance for the DB cluster that you want to create a clone of.
3. Choose **Instance actions**, and then choose **Create clone**.
4. On the **Create Clone** page, enter a name for the primary instance of the clone DB cluster as the **DB instance identifier**.

   If you want to, configure any other settings for the clone DB cluster. For information about the different DB cluster settings, see Launch using the console.
5. Choose **Create Clone** to launch the clone DB cluster.

To create a clone of a DB cluster using the AWS CLI

- Call the Neptune restore-db-cluster-to-point-in-time AWS CLI command and supply the following values:
  - `--source-db-cluster-identifier` – The name of the source DB cluster to create a clone of.
  - `--db-cluster-identifier` – The name of the clone DB cluster.
  - `--restore-type copy-on-write` – The `copy-on-write` value indicates that a clone DB cluster should be created.
  - `--use-latest-restorable-time` – This specifies that the latest restorable backup time should be used.

> **Note**
>
> The restore-db-cluster-to-point-in-time AWS CLI command only clones the DB cluster, not the DB instances for that DB cluster.

The following Linux/UNIX example creates a clone from the `source-db-cluster-id` DB cluster and names the clone `db-clone-cluster-id`.

```
aws neptune restore-db-cluster-to-point-in-time \
  --region us-east-1 \
  --source-db-cluster-identifier source-db-cluster-id \
  --db-cluster-identifier db-clone-cluster-id \
  --restore-type copy-on-write \
  --use-latest-restorable-time
```

The same example works on Windows if the \ line-end escape character is replaced by the Windows ^ equivalent:

```
aws neptune restore-db-cluster-to-point-in-time ^
  --region us-east-1 ^
  --source-db-cluster-identifier source-db-cluster-id ^
  --db-cluster-identifier db-clone-cluster-id ^
  --restore-type copy-on-write ^
```

```
--use-latest-restorable-time
```

Limitations

DB cloning in Neptune has the following limitations:

- You can't create clone databases across AWS Regions. The clone databases must be created in the same Region as the source databases.
- A cloned database always uses the most recent patch of the Neptune engine version being used by the database it was cloned from. This is true even if the source database has not yet been upgraded to that patch version. The engine version itself does not change, however.
- Currently, you are limited to no more than 15 clones per copy of your Neptune DB cluster, including clones based on other clones. After reaching that limit, you must make another copy of your database rather than cloning it. However, if you make a new copy, it can also have up to 15 clones.
- Cross-account DB cloning is not currently supported.
- You can provide a different virtual private cloud (VPC) for your clone. However, the subnets in those VPCs must map to the same set of Availability Zones.

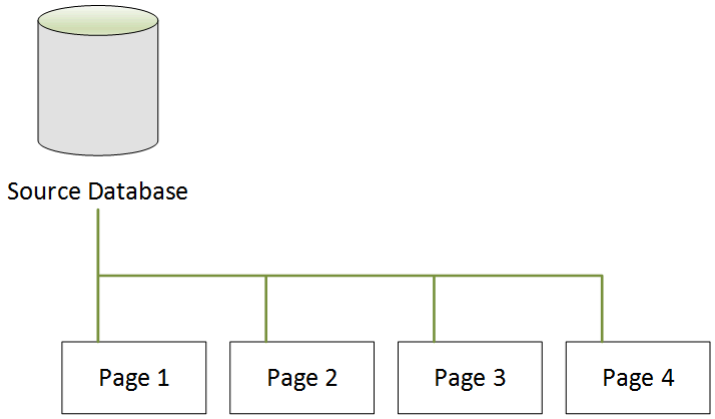
Copy-on-Write Protocol for DB Cloning

The following scenarios illustrate how the copy-on-write protocol works.

- Neptune Database Before Cloning
- Neptune Database After Cloning
- When a Change Is Made to the Source Database
- When a Change Is Made to the Clone Database

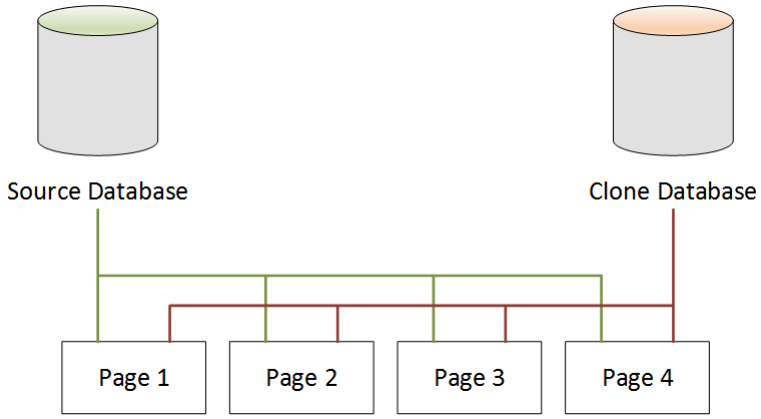
Neptune Database Before Cloning

Data in a source database is stored in pages. In the following diagram, the source database has four pages.



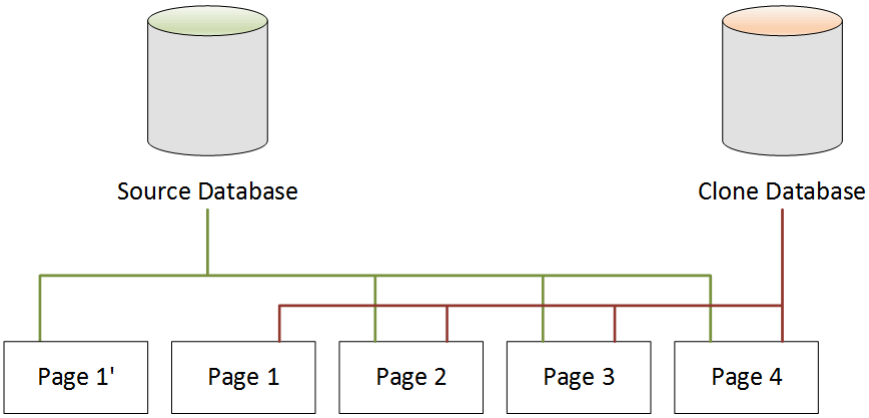
Neptune Database After Cloning

As shown in the following diagram, there are no changes in the source database after DB cloning. Both the source database and the clone database point to the same four pages. No pages have been physically copied, so no additional storage is required.



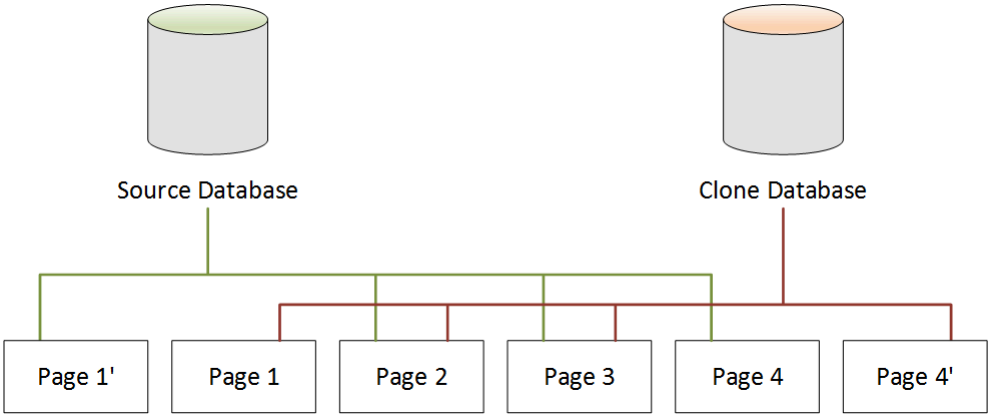
When a Change Is Made to the Source Database

In the following example, the source database makes a change to the data in Page 1. Instead of writing to the original Page 1, it uses additional storage to create a new page, called Page 1'. The source database now points to the new Page 1', and also to Page 2, Page 3, and Page 4. The clone database continues to point to Page 1 through Page 4.



When a Change Is Made to the Clone Database

In the following diagram, the clone database has also changed, this time in Page 4. Instead of writing to the original Page 4, additional storage is used to create a new page, called Page 4'. The source database continues to point to Page 1', and also Page 2 through Page 4, but the clone database now points to Page 1 through Page 3, and also Page 4'.



As shown in the second scenario, after DB cloning, there is no additional storage required at the point of clone creation. However, as changes occur in the source database and clone database, only the changed pages are created, as shown in the third and fourth scenarios. As more changes occur over time in both the source database and clone database, you need incrementally more storage to capture and store the changes.

Deleting a Source Database

Deleting a source database does not affect the clone databases that are associated with it. The clone databases continue to point to the pages that were previously owned by the source database.

Managing Amazon Neptune Instances

The following sections have information on instance-level operations.

Topics

- Neptune T3 Burstable Instance Class
- Modifying a Neptune DB Instance (and Applying Immediately)
- Renaming a Neptune DB Instance
- Rebooting a DB instance in Amazon Neptune
- Deleting a DB Instance in Amazon Neptune

Neptune T3 Burstable Instance Class

In addition to fixed-performance instance classes such as R5 and R6, Amazon Neptune gives you the option of using a burstable-performance T3 instance. While you're developing your graph application, you want your database to be fast and responsive, but you don't need to use it all the time. Neptune's `db.t3.medium` instance class is just what you should use in that situation, at significantly lower cost than the least expensive fixed-performance instance class.

A burstable instance runs at a baseline level of CPU performance until a workload needs more, and then bursts well above the baseline for as long as a workload requires. Its hourly price covers the bursts, provided that the average CPU utilization doesn't exceed the baseline over a 24-hour period. For most development and test situations, that translates to good performance at a low cost.

If you start with a T3 instance class, you can easily switch later to a fixed-performance instance class when you're ready to go into production, using the AWS Management Console, AWS CLI, or one of the AWS SDKs.

T3 Bursting Is Governed by CPU Credits

A CPU credit represents the full utilization of one virtual CPU core (vCPU) for one minute. That can also translate into 50% utilization of a vCPU for two minutes, or 25% utilization of two vCPUs for two minutes, and so on.

A T3 instance accrues CPU credits when it's idle and uses them up when it's active, both measured at millisecond resolution. The `db.t3.medium` instance class has two vCPUs, each of which earns 12 CPU credits per hour when idle. This means that 20% utilization of each vCPU results in a zero CPU credit balance. The 12 CPU credits earned are spent by 20% utilization of the vCPU (since 20% of 60 minutes is also 12). This 20% utilization is thus the *baseline* utilization rate that produces neither a positive nor negative CPU-credit balance.

Idle time (CPU utilization below 20% of the total available) causes CPU credits to be stored in a credit balance bucket, up to the limit for a `db.t3.medium` instance class of 576 (the maximum number of CPU credits that could be accrued in 24 hours, namely 2 x 12 x 24). Over that limit, CPU credits are simply discarded.

When necessary, CPU utilization can burst to as high as 100% for as long as needed by a workload, even after the CPU credit balance falls below zero. If the instance sustains a negative balance continuously for 24 hours, it incurs an additional charge of $0.05 for every -60 CPU credits accrued

over that period. For most development and test workloads, however, bursting is usually covered by idle time before or after the burst.

> **Note**
>
> Neptune's T3 instance class is configured like the Amazon EC2 unlimited mode.

Using the AWS Management Console to Create a T3 Burstable Instance

In the AWS Management Console, you can create a primary DB cluster instance or a read-replica instance that uses the `db.t3.medium` instance class, or you can modify an existing instance to use the `db.t3.medium` instance class.

For example, to create a new DB cluster primary instance in the Neptune console:

- Choose **Create Database**.
- Choose a **DB engine version** equal to or later than `1.0.2.2`.
- Under **Purpose**, choose **Development and Testing**.
- As the **DB instance class**, accept the default: `db.t3.medium — 2 vCPU, 4 GiB RAM`.

Using the AWS CLI to Create a T3 Burstable Instance

You can also use the AWS CLI to do the same thing:

```
aws neptune create-db-cluster \
    --db-cluster-identifier (name for a new DB cluster) \
    --engine neptune \
    --engine-version "1.0.2.2"

aws neptune create-db-instance \
    --db-cluster-identifier (name of the new DB cluster) \
    --db-instance-identifier (name for the primary writer instance in the cluster) \
    --engine neptune \
    --db-instance-class db.t3.medium
```

Modifying a Neptune DB Instance (and Applying Immediately)

You can apply most changes to an Amazon Neptune DB instance immediately or defer them until the next maintenance window. Some modifications, such as parameter group changes, require that you manually reboot your DB instance for the change to take effect.

> **Important**
>
> Modifications result in an outage if Neptune must reboot your DB instance for the change to be applied. Review the impact on your database and applications before modifying DB instance settings.

Common Settings and Downtime Implications

The following table contains details about which settings you can change, when the changes can be applied, and whether the changes cause downtime for the DB instance.

DB instance setting	Downtime notes
DB instance class	An outage occurs during this change, whether it is applied immediately or during the next maintenance window.
DB instance identifier	The DB instance is rebooted and an outage occurs during this change, whether it is applied immediately or during the next maintenance window.
Subnet group	The DB instance is rebooted and an outage occurs during this change, whether it is applied immediately or during the next maintenance window.

DB instance setting	Downtime notes	
Security group	The change is applied asynchronously as soon as possible, regardless of when you specify changes should take place, and no outage results.	–
Certificate Authority	By default, the DB instance is restarted when you assign a new Certificate Authority.	
Database Port	The change always occurs immediately, causing the DB instance to be rebooted, and an outage occurs.	

DB instance setting	Downtime notes	
DB parameter group	Changing this setting doesn't result in an outage. The parameter group name itself is changed immediately, but the actual parameter changes are not applied until you reboot the instance without failover. In this case, the DB instance isn't rebooted automatic ally, and the parameter changes aren't applied during the next maintenan ce window. However, if you modify dynamic parameter s in the newly associate d DB parameter group, these changes are applied immediately without a reboot. For more information, see Rebooting a DB instance in Amazon Neptune.	
DB cluster parameter group	The DB parameter group name is changed immediate ly.	

DB instance setting	Downtime notes	
Backup retention period	If you specify that changes should occur immediate ly, this change does occur immediately. Otherwise, if you change the setting from a nonzero value to another nonzero value, the change is applied asynchronously, as soon as possible. Any other change occurs during the next maintenance window. An outage occurs if you change from zero to a nonzero value, or from a nonzero value to zero.	
Audit log	Select **Audit log** if you want to use audit logging through CloudWatch Logs. You must also set the `neptune_enable_audit_log` parameter in the DB cluster parameter group to `enable` (1) for audit logging to be enabled.	

DB instance setting	Downtime notes	
Auto minor version upgrade	Select **Enable auto minor version upgrade** if you want to enable your Neptune DB cluster to receive minor engine version upgrades automatically when they become available. The *Auto minor version upgrade* option only applies to upgrades to minor engine versions for your Amazon Neptune DB cluster. It doesn't apply to regular patches applied to maintain system stability.	

Renaming a Neptune DB Instance

You can rename an Amazon Neptune DB instance by using the AWS Management Console. Renaming a DB instance can have far-reaching effects. The following is a list of things you should know before you rename a DB instance.

- When you rename a DB instance, the endpoint for the DB instance changes because the URL includes the name you assigned to the DB instance. You should always redirect traffic from the old URL to the new one.
- When you rename a DB instance, the old DNS name that was used by the DB instance is immediately deleted, but it can remain cached for a few minutes. The new DNS name for the renamed DB instance becomes effective after about 10 minutes. The renamed DB instance is not available until the new name becomes effective.
- You can't use an existing DB instance name when you are renaming an instance.
- All read replicas that are associated with a DB instance remain associated with that instance after it is renamed. For example, suppose that you have a DB instance that serves your production database, and the instance has several associated read replicas. If you rename the DB instance and then replace it in the production environment with a DB snapshot, the DB instance that you renamed still has the read replicas associated with it.
- Metrics and events that are associated with the name of a DB instance are maintained if you reuse a DB instance name. For example, if you promote a read replica and rename it to be the name of the previous primary instance, the events and metrics that were associated with the primary instance are then associated with the renamed instance.
- DB instance tags remain with the DB instance, regardless of renaming.
- DB snapshots are retained for a renamed DB instance.

To rename a DB instance using the Neptune console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Databases**.
3. Choose the radio button next to the DB instance that you want to rename.
4. In the **Instance actions** menu, choose **Modify**.
5. Enter a new name in the **DB instance identifier** text box. Select **Apply immediately**, and then choose **Continue**.

6. Choose **Modify DB instance** to complete the change.

Rebooting a DB instance in Amazon Neptune

In some cases, if you modify an Amazon Neptune DB instance, change the DB parameter group that is associated with the instance, or change a static DB parameter in a parameter group that the instance uses, you must reboot the instance to apply the changes.

Rebooting a DB instance restarts the database engine service. A reboot also applies to the DB instance any changes to the associated DB parameter group that were pending. Rebooting a DB instance results in a momentary outage of the instance, during which the DB instance status is set to *rebooting*. If the Neptune instance is configured for Multi-AZ, the reboot might be conducted through a failover. A Neptune event is created when the reboot is completed.

If your DB instance is a Multi-AZ deployment, you can force a failover from one Availability Zone to another when you choose the **Reboot** option. When you force a failover of your DB instance, Neptune automatically switches to a standby replica in another Availability Zone. It then updates the DNS record for the DB instance to point to the standby DB instance. As a result, you must clean up and re-establish any existing connections to your DB instance.

Reboot with failover is beneficial when you want to simulate a failure of a DB instance for testing or restore operations to the original Availability Zone after a failover occurs. For more information, see High Availability (Multi-AZ) in the *Amazon RDS User Guide*. When you reboot a DB cluster, it fails over to the standby replica. Rebooting a Neptune replica does not initiate a failover.

The time required to reboot is a function of the crash recovery process. To improve the reboot time, we recommend that you reduce database activities as much as possible during the reboot process to reduce rollback activity for in-transit transactions.

On the console, the **Reboot** option might be disabled if the DB instance is not in the **Available** state. This can be due to several reasons, such as an in-progress backup, a customer-requested modification, or a maintenance window action.

> **Note**
>
> Before Release: 1.2.0.0 (2022-07-21), all the read-replicas in a DB cluster were automatically rebooted whenever the primary (writer) instance restarted.
> Starting with Release: 1.2.0.0 (2022-07-21), restarting the primary instance does not cause any of the replicas to restart. This means that if you are changing a cluster parameter, you must restart each instance separately to pick up the parameter change (see Parameter groups).

To reboot a DB instance using the Neptune console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Databases**.
3. Choose the DB instance that you want to reboot.
4. Choose **Instance actions**, and then choose **Reboot**.
5. To force a failover from one Availability Zone to another, select **Reboot with failover?** in the **Reboot DB Instance** dialog box.
6. Choose **Reboot**. To cancel the reboot, choose **Cancel** instead.

Deleting a DB Instance in Amazon Neptune

You can delete an Amazon Neptune DB instance in any state and at any time, as long the instance has been started.

> **Warning**
>
> If you delete the last remaining instance in a cluster using the **web console**, it will also delete the underlying cluster storage volume.

How deletion protection affects DB instance deletion

Deletion protection is a *cluster-level* setting. When it is enabled, it prevents the DB cluster, along with its storage volume and data, from being deleted. You set deletion protection on the DB cluster, not on an individual DB instance.

The effect of cluster deletion protection on the deletion of an individual DB instance depends on the method that you use:

- **AWS Management Console** – While deletion protection is enabled on the cluster, you cannot delete the DB instances in that cluster. The console blocks the deletion.
- **API, SDK, or AWS CLI** – You can still delete individual DB instances, including the last (primary) instance, while cluster deletion protection is enabled. Neptune blocks only the `DeleteDBCluster` operation, which fails until you disable deletion protection. If you remove the last instance by using the API, an SDK, or the AWS CLI, the cluster remains with no instances. Its storage volume and data are retained until you disable deletion protection and explicitly delete the cluster.

> **Note**
>
> Deletion protection is not a way to enforce a minimum number of read replicas. Read-replica auto-scaling removes readers through the Neptune API, so cluster deletion protection does not prevent a scale-in activity from removing a read replica. The `min-capacity` of your scaling policy determines the minimum number of readers that auto-scaling maintains. For more information, see Auto-scaling the number of replicas in an Amazon Neptune DB cluster.

To delete a DB cluster that has deletion protection enabled, first disable deletion protection on the cluster. Enabling or disabling deletion protection does not cause an outage. When you create a DB cluster in the AWS Management Console, deletion protection is enabled by default. When you create a DB cluster with the AWS CLI or the API, it is disabled by default.

Taking a Final Snapshot of Your DB Instance Before Deleting It

To delete a DB instance, you must specify the name of the instance and whether you want to have a final DB snapshot taken of the instance. If the DB instance that you're deleting has a status of **Creating**, you can't have a final DB snapshot taken. If the DB instance is in a failure state with a status of **failed**, **incompatible-restore**, or **incompatible-network**, you can only delete the instance when the `SkipFinalSnapshot` parameter is set to `true`.

If you delete all Neptune DB instances in a DB cluster using the AWS Management Console, the entire DB cluster is deleted automatically. If you are using the AWS CLI or SDK, you must delete the DB cluster manually after you delete the last instance.

> **Important**
>
> If you delete an entire DB cluster, all its automated backups are deleted at the same time, and cannot be recovered. This means that unless you choose to create a final DB snapshot manually, you can't restore the DB instance to its final state at a later time. Manual snapshots of an instance are not deleted when the cluster is deleted.

If the DB instance that you want to delete has a read replica, you should either promote the read replica or delete it.

In the following examples, you delete a DB instance both with and without a final DB snapshot.

Deleting a DB Instance with No Final Snapshot

If you want to quickly delete a DB instance, you can skip creating a final DB snapshot. When you delete a DB instance, all automated backups are deleted and cannot be recovered. Manual snapshots are not deleted.

To delete a DB instance with no final DB snapshot using the Neptune console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.

2. In the navigation pane, choose **Databases**.
3. In the **Instances** list, choose the radio button next to the DB instance that you want to delete.
4. Choose **Instance actions**, and then choose **Delete**.
5. Choose **No** in the **Create final snapshot?** box.
6. Choose **Delete**.

Deleting a DB Instance with a Final Snapshot

If you want to be able to restore a deleted DB instance at a later time, you can create a final DB snapshot. All automated backups are also deleted and cannot be recovered. Manual snapshots are not deleted.

To delete a DB instance with a final DB snapshot using the Neptune console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Databases**.
3. In the **Instances** list, choose the radio button next to the DB instance that you want to delete.
4. Choose **Instance actions**, and then choose **Delete**.
5. Choose **Yes** in the **Create final snapshot?** box.
6. In the **Final snapshot name** box, enter the name of your final DB snapshot.
7. Choose **Delete**.

You can check the health of an instance, determine what kind of instance it is, find out which engine release version you currently have installed, and obtain other information about an instance using the instance-status API.

Capturing graph changes in real time using Neptune streams

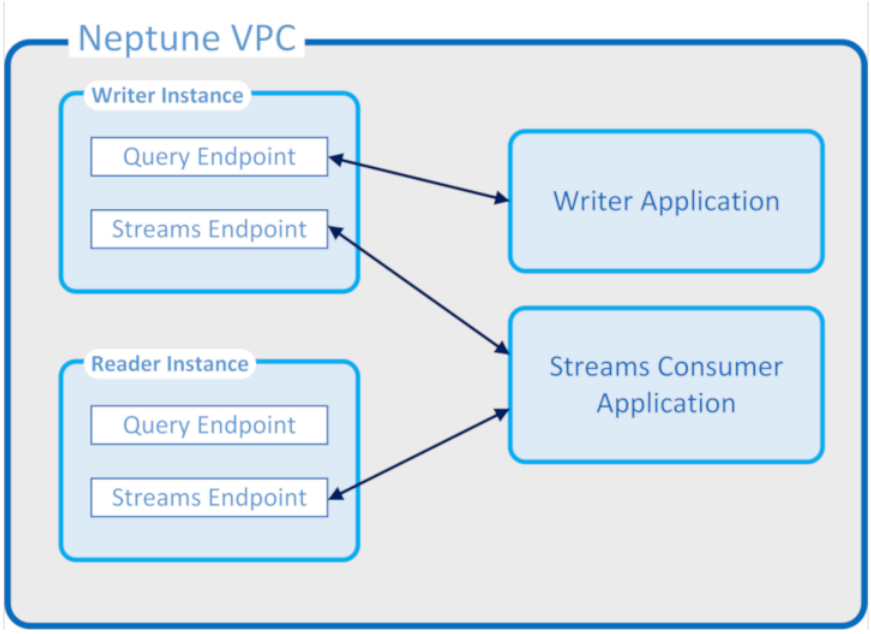
Neptune Streams logs every change to your graph as it happens, in the order that it is made, in a fully managed way. Once you enable Streams, Neptune takes care of availability, backup, security and expiry.

The following are some of the many use cases where you might want to capture changes to a graph as they occur:

- You might want your application to notify people automatically when certain changes are made.
- You might want to maintain a current version of your graph data in another data store also, such as Amazon OpenSearch Service, Amazon ElastiCache, or Amazon Simple Storage Service (Amazon S3).

Neptune uses the same native storage for the change-log stream as for graph data. It writes change log entries synchronously together with the transaction that makes those changes. You retrieve these change records from the log stream using an HTTP REST API. (For information, see Calling the Streams API.)

The following diagram shows how change-log data can be retrieved from Neptune Streams.



Neptune streams guarantees

- Changes made by a transaction are immediately available for reading from both writer and readers as soon as the transaction is complete (aside from any normal replication lag in readers).
- Change records appear strictly sequentially, in the order in which they occurred (this includes the changes made within a transaction).
- The changes streams contain no duplicates. Each change is logged only once.
- The changes streams are complete. No changes are lost or omitted.
- The changes streams contain all the information needed to determine the complete state of the database itself at any point in time, provided that the starting state is known.
- Streams can be turned on or off at any time.

Neptune streams operational properties

- The change-log stream is fully managed.
- Change-log data is written synchronously as part of the same transaction that makes a change.
- When Neptune Streams are enabled, you incur I/O and storage charges associated with the change-log data.
- By default, change records are automatically purged one week after they are created. Starting with engine release 1.2.0.0, this retention period can be changed using the the neptune_streams_expiry_days DB cluster parameter to any number of days between 1 and 90.
- Read performance on the streams scales with instances.
- You can achieve high availability and read throughput using read replicas. There is no limit on the number of stream readers that you can create and use concurrently.
- Change-log data is replicated across multiple Availability Zones, making it highly durable.
- The log data is as secure as your graph data itself. It can be encrypted at rest and in transit. Access can be controlled using IAM, Amazon VPC, and AWS Key Management Service (AWS KMS). Like the graph data, it can be backed up and later restored using point-in-time restores (PITR).
- The synchronous writing of stream data as part of each transaction causes a slight degradation in overall write performance.
- Stream data is not sharded, because Neptune is single-sharded by design.
- The log stream `GetRecords` API uses the same resources as all other Neptune graph operations. This means that clients need to load balance between stream requests and other DB requests.

- When streams are disabled, all log data becomes inaccessible immediately. This means that you must read all log data of interest to you before you disable logging.
- There is currently no native integration with AWS Lambda. The log stream does not generate an event that can trigger a Lambda function.

Topics

- Using Neptune Streams
- Serialization Formats in Neptune Streams
- Neptune Streams Examples
- Using AWS CloudFormation to Set Up Neptune-to-Neptune Replication with the Streams Consumer Application

Using Neptune Streams

With the Neptune Streams feature, you can generate a complete sequence of change-log entries that record every change made to your graph data as it happens. For an overview of this feature, see Capturing graph changes in real time using Neptune streams.

Topics

- Enabling Neptune Streams
- Disabling Neptune Streams
- Calling the Neptune Streams REST API
- Neptune Streams API Response Format
- Reading Neptune stream data
- Neptune Streams API Exceptions

Enabling Neptune Streams

You can enable or disable Neptune Streams at any time by setting the `neptune_streams` DB cluster parameter. Setting the parameter to 1 enables Streams, and setting it to `0` disables Streams.

> **Note**
>
> After changing the `neptune_streams` DB cluster parameter, you must reboot all DB instances in the cluster for the change to take effect.

You can set the neptune_streams_expiry_days DB cluster parameter to control how many days, from 1 to 90, that stream records remain on the server before being deleted. The default is 7.

Neptune Streams was initially introduced as an experimental feature that you enabled or disabled in Lab Mode using the DB Cluster `neptune_lab_mode` parameter (see Neptune Lab Mode). Using Lab Mode to enable Streams is now deprecated and will be disabled in the future.

Disabling Neptune Streams

You can turn Neptune Streams off any time that it is running.

To turn Streams off, update the DB Cluster parameter group so that the value of the `neptune_streams` parameter is set to 0.

> **Important**
>
> As soon as Streams is turned off, you can't access the change-log data any more. Be sure to read what you are interested in *before* turning Streams off.

Calling the Neptune Streams REST API

You access Neptune Streams using a REST API that sends an HTTP GET request to one of the following local endpoints:

- For a SPARQL graph DB: `https://Neptune-DNS:8182/sparql/stream`.
- For a Gremlin or openCypher graph DB: `https://Neptune-DNS:8182/propertygraph/stream` or `https://Neptune-DNS:8182/pg/stream`.

> **Note**
>
> The Gremlin stream endpoint (`https://Neptune-DNS:8182/gremlin/stream`) is deprecated, along with its associated output format (`GREMLIN_JSON`). It is still supported for backward compatibility but may be removed in future releases.

Only an HTTP `GET` operation is allowed.

Neptune supports `gzip` compression of the response, provided that the HTTP request includes an `Accept-Encoding` header that specifies `gzip` as an accepted compression format (that is, `"Accept-Encoding: gzip"`).

Parameters

- `limit` – long, optional. Range: 1–100,000. Default: 10.

  Specifies the maximum number of records to return. There is also a size limit of 10 MB on the response that can't be modified and that takes precedence over the number of records specified in the `limit` parameter. The response does include a threshold-breaching record if the 10 MB limit was reached.

- `iteratorType` – String, optional.

  This parameter can take one of the following values:

  - `AT_SEQUENCE_NUMBER`(default) – Indicates that reading should start from the event sequence number specified jointly by the `commitNum` and `opNum` parameters.
  - `AFTER_SEQUENCE_NUMBER` – Indicates that reading should start right after the event sequence number specified jointly by the `commitNum` and `opNum` parameters.
  - `TRIM_HORIZON` – Indicates that reading should start at the last untrimmed record in the system, which is the oldest unexpired (not yet deleted) record in the change-log stream. This mode is useful during application startup, when you don't have a specific starting event sequence number.
  - `LATEST` – Indicates that reading should start at the most recent record in the system, which is the latest unexpired (not yet deleted) record in the change-log stream. This is useful when there is a need to read records from current top of the streams so as not to process older records, such as during disaster recovery or a zero-downtime upgrade. Note that in this mode, there is at most only one record returned.

- `commitNum` – long, required when iteratorType is `AT_SEQUENCE_NUMBER` or `AFTER_SEQUENCE_NUMBER`.

  The commit number of the starting record to read from the change-log stream.

  This parameter is ignored when `iteratorType` is `TRIM_HORIZON` or `LATEST`.
- `opNum` – long, optional (the default is 1).

  The operation sequence number within the specified commit to start reading from in the change-log stream data.

Operations that change SPARQL graph data generally only generate a single change record per operation. However, operations that change Gremlin graph data can generate multiple change records per operation, as in the following examples:

- `INSERT` – A Gremlin vertex can have multiple labels, and a Gremlin element can have multiple properties. A separate change record is generated for each label and property when an element is inserted.
- `UPDATE` – When a Gremlin element property is changed, two change records are generated: the first for removing the previous value, and the second for inserting the new value.
- `DELETE` – A separate change record is generated for each element property that is deleted. For example, when a Gremlin edge with properties is deleted, one change record is generated for each of the properties, and after that, one is generated for deletion of the edge label.

  When a Gremlin vertex is deleted, all the incoming and outgoing edge properties are deleted first, then the edge labels, then the vertex properties, and finally the vertex labels. Each of these deletions generates a change record.

Neptune Streams API Response Format

A response to a Neptune Streams REST API request has the following fields:

- `lastEventId` – Sequence identifier of the last change in the stream response. An event ID is composed of two fields: A `commitNum` identifies a transaction that changed the graph, and an `opNum` identifies a specific operation within that transaction. This is shown in the following example.

```
  "eventId": {
    "commitNum": 12,
    "opNum": 1
  }
```

- `lastTrxTimestamp` – The time at which the commit for the transaction was requested, in milliseconds from the Unix epoch.
- `format` – Serialization format for the change records being returned. The possible values are `PG_JSON` for Gremlin or openCypher change records, and `NQUADS` for SPARQL change records.
- `records` – An array of serialized change-log stream records included in the response. Each record in the `records` array contains these fields:
  - `commitTimestamp` – The time at which the commit for the transaction was requested, in milliseconds from the Unix epoch.
  - `eventId` – The sequence identifier of the stream change record.
  - `data` – The serialized Gremlin, SPARQL, or OpenCypher change record. The serialization formats of each record are described in more detail in the next section, Serialization Formats in Neptune Streams.
  - op – The operation that created the change. Valid values are:
    - `ADD` – Indicates that a new element (such as a vertex, edge, property, or triple) was added to the graph.
    - `REMOVE` – Indicates that an existing element was removed from the graph.

    Neptune expresses a modification to an existing element as a `REMOVE` record followed by an `ADD` record.
  - `isLastOp` – Only present if this operation is the last one in its transaction. When present, it is set to `true`. Useful for ensuring that an entire transaction is consumed.
- `totalRecords` – The total number of records in the response.

For example, the following response returns Gremlin change data, for a transaction that contains more than one operation:

```
{
  "lastEventId": {
    "commitNum": 12,
    "opNum": 1
```

```
  },
  "lastTrxTimestamp": 1560011610678,
  "format": "PG_JSON",
  "records": [
    {
      "commitTimestamp": 1560011610678,
      "eventId": {
        "commitNum": 1,
        "opNum": 1
      },
      "data": {
        "id": "d2b59bf8-0d0f-218b-f68b-2aa7b0b1904a",
        "type": "vl",
        "key": "label",
        "value": {
          "value": "vertex",
          "dataType": "String"
        }
      },
      "op": "ADD"
    }
  ],
  "totalRecords": 1
}
```

The following response returns SPARQL change data for the last operation in a transaction (the operation identified by `EventId(97, 1)` in transaction number 97).

```
{
  "lastEventId": {
    "commitNum": 97,
    "opNum": 1
  },
  "lastTrxTimestamp": 1561489355102,
  "format": "NQUADS",
  "records": [
    {
      "commitTimestamp": 1561489355102,
      "eventId": {
        "commitNum": 97,
        "opNum": 1
      },
      "data": {
```

```
        "stmt": "<https://test.com/s> <https://test.com/p> <https://test.com/o> .\n"
      },
      "op": "ADD",
      "isLastOp": true
    }
  ],
  "totalRecords": 1
}
```

Reading Neptune stream data

The following examples show how to read records from the property graph stream endpoint. You can use the AWS CLI, the AWS SDK for Python (Boto3), **awscurl**, or **curl**.

AWS CLI

```
aws neptunedata get-propertygraph-stream \
  --endpoint-url https://your-neptune-endpoint:port \
  --iterator-type TRIM_HORIZON \
  --limit 10
```

To read from a specific position in the stream, use the `--commit-num` and `--iterator-type` parameters:

```
aws neptunedata get-propertygraph-stream \
  --endpoint-url https://your-neptune-endpoint:port \
  --limit 10 \
  --commit-num 1 \
  --iterator-type AT_SEQUENCE_NUMBER
```

For more information, see get-propertygraph-stream in the AWS CLI Command Reference.

SDK (Python)

```
import boto3
import json
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    region_name='us-east-1',
```

```
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=65)
)

response = client.get_propertygraph_stream(
    limit=10,
    iteratorType='TRIM_HORIZON'
)

print(json.dumps(response['records'], indent=2, default=str))
```

To paginate through the stream, use the `commitNum` and `opNum` from the `lastEventId` in the response:

```
last_event = response['lastEventId']
next_response = client.get_propertygraph_stream(
    limit=10,
    commitNum=last_event['commitNum'],
    opNum=last_event['opNum'],
    iteratorType='AFTER_SEQUENCE_NUMBER'
)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/propertygraph/stream?limit=10 \
  --region us-east-1 \
  --service neptune-db
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

For more information about using **awscurl** with IAM authentication, see Using `awscurl` with temporary credentials to securely connect to a DB cluster with IAM authentication enabled.

curl

```
curl https://your-neptune-endpoint:port/propertygraph/stream?limit=10
```

> **Note**
>
> Using **curl** without SigV4 signing only works if IAM authentication is disabled on your Neptune cluster. If IAM authentication is enabled, use **awscurl** or the AWS CLI instead.

For SPARQL graphs, replace `/propertygraph/stream` with `/sparql/stream` in the preceding examples. When using the AWS CLI or SDK, use `get-sparql-stream` or `client.get_sparql_stream()` instead.

Viewing stream data in a graph notebook

If you use Neptune graph notebooks, the `%stream_viewer` line magic provides a visual interface for browsing stream records interactively without writing code. Run the following in a notebook cell:

```
%stream_viewer
```

You can optionally specify the query language and page size:

```
%stream_viewer sparql --limit 20
```

For more information, see ???.

> **Note**
>
> The `%stream_viewer` magic is fully supported only on engine versions 1.0.5.1 and earlier.

Neptune Streams API Exceptions

The following table describes Neptune Streams exceptions.

Error Code	HTTP Code	OK to Retry?	Message
`InvalidParameterException`	400	No	An invalid or out-of-range value was

Error Code	HTTP Code	OK to Retry?	Message
			supplied as an input parameter.
`ExpiredStreamException`	400	No	All of the requested records exceed the maximum age allowed and have expired.
`ThrottlingException`	500	Yes	Rate of requests exceeds the maximum throughput.
`StreamRecordsNotFoundException`	404	No	The requested resource could not be found. The stream may not be specified correctly.
`MemoryLimitExceededException`	500	Yes	The request processing did not succeed due to lack of memory, but can be retried when the server is less busy.

Serialization Formats in Neptune Streams

Amazon Neptune uses two different formats for serializing graph-changes data to log streams, depending on whether the graph was created using Gremlin or SPARQL.

Both formats share a common record serialization format, as described in Neptune Streams API Response Format, that contains the following fields:

- `commitTimestamp` – The time at which the commit for the transaction was requested, in milliseconds from the Unix epoch.
- `eventId` – The sequence identifier of the stream change record.
- `data` – The serialized Gremlin, SPARQL, or OpenCypher change record. The serialization formats of each record are described in more detail in the next sections.
- `op` – The operation that created the change.

Topics

- PG_JSON Change Serialization Format
- SPARQL NQUADS Change Serialization Format

PG_JSON Change Serialization Format

> **Note**
>
> The Gremlin stream output format (`GREMLIN_JSON`) output by the Gremlin stream endpoint (`https://Neptune-DNS:8182/gremlin/stream`) is deprecated. It is replaced by PG_JSON, which is currently identical to `GREMLIN_JSON`.

A Gremlin or openCypher change record, contained in the `data` field of a log stream response, has the following fields:

- `id` – String, required.

  The ID of the Gremlin or openCypher element.
- `type` – String, required.

  The type of this Gremlin or openCypher element. Must be one of the following:
  - `vl` – Vertex label for Gremlin; node label for openCypher.
  - `vp` – Vertex properties for Gremlin; node properties for openCypher.
  - `e` – Edge and edge label for Gremlin; relationship and relationship type for openCypher.
  - `ep` – Edge properties for Gremlin; relationship properties for openCypher.
- `key` – String, required.

The property name. For element labels, this is "label".

- `value` – `value` object, required.

  This is a JSON object that contains a `value` field for the value itself, and a `datatype` field for the JSON data type of that value.

```
"value": {
  "value": "the new value",
  "dataType": "the JSON datatype of the new value"
}
```

- `from` – String, optional.

  If this is an edge (type="e"), the ID of the corresponding *from* vertex or source node.

- `to` – String, optional.

  If this is an edge (type="e"), the ID of the corresponding *to* vertex or target node.

Gremlin Examples

- The following is an example of a Gremlin vertex label.

```
{
  "id": "an ID string",
  "type": "vl",
  "key": "label",
  "value": {
    "value": "the new value of the vertex label",
    "dataType": "String"
  }
}
```

- The following is an example of a Gremlin vertex property.

```
{
  "id": "an ID string",
  "type": "vp",
  "key": "the property name",
  "value": {
    "value": "the new value of the vertex property",
```

```
    "dataType": "the datatype of the vertex property"
  }
}
```

- The following is an example of a Gremlin edge.

```
{
  "id": "an ID string",
  "type": "e",
  "key": "label",
  "value": {
    "value": "the new value of the edge",
    "dataType": "String"
  },
  "from": "the ID of the corresponding "from" vertex",
  "to": "the ID of the corresponding "to" vertex"
}
```

openCypher Examples

- The following is an example of an openCypher node label.

```
{
  "id": "an ID string",
  "type": "vl",
  "key": "label",
  "value": {
    "value": "the new value of the node label",
    "dataType": "String"
  }
}
```

- The following is an example of an openCypher node property.

```
{
  "id": "an ID string",
  "type": "vp",
  "key": "the property name",
  "value": {
    "value": "the new value of the node property",
    "dataType": "the datatype of the node property"
  }
```

```
}
```

- The following is an example of an openCypher relationship.

```
{
  "id": "an ID string",
  "type": "e",
  "key": "label",
  "value": {
    "value": "the new value of the relationship",
    "dataType": "String"
  },
  "from": "the ID of the corresponding source node",
  "to": "the ID of the corresponding target node"
}
```

SPARQL NQUADS Change Serialization Format

Neptune logs changes to SPARQL quads in the graph using the Resource Description Framework (RDF) `N-QUADS` language defined in the W3C RDF 1.1 N-Quads specification.

The `data` field in the change record simply contains a `stmt` field that holds an N-QUADS statement expressing the changed quad, as in the following example.

```
"stmt" : "<https://test.com/s> <https://test.com/p> <https://test.com/o> .\n"
```

Neptune Streams Examples

The following examples show how to access change-log stream data in Amazon Neptune.

Topics

- AT_SEQUENCE_NUMBER Change Log
- AFTER_SEQUENCE_NUMBER Change Log
- TRIM_HORIZON Change Log
- LATEST Change Log
- Compression Change Log

AT_SEQUENCE_NUMBER Change Log

The following example shows a Gremlin or openCypher AT_SEQUENCE_NUMBER change log.

```
curl -s "https://Neptune-DNS:8182/propertygraph/stream?
limit=1&commitNum=1&opNum=1&iteratorType=AT_SEQUENCE_NUMBER" |jq
{
  "lastEventId": {
    "commitNum": 1,
    "opNum": 1
  },
  "lastTrxTimestamp": 1560011610678,
  "format": "PG_JSON",
  "records": [
    {
      "eventId": {
        "commitNum": 1,
        "opNum": 1
      },
      "commitTimestamp": 1560011610678,
      "data": {
        "id": "d2b59bf8-0d0f-218b-f68b-2aa7b0b1904a",
        "type": "vl",
        "key": "label",
        "value": {
          "value": "vertex",
          "dataType": "String"
        }
      },
      "op": "ADD",
      "isLastOp": true
    }
  ],
  "totalRecords": 1
}
```

This one shows a SPARQL example of an AT_SEQUENCE_NUMBER change log.

```
curl -s "https://localhost:8182/sparql/stream?
limit=1&commitNum=1&opNum=1&iteratorType=AT_SEQUENCE_NUMBER" |jq
{
  "lastEventId": {
    "commitNum": 1,
```

```
    "opNum": 1
  },
  "lastTrxTimestamp": 1571252030566,
  "format": "NQUADS",
  "records": [
    {
      "eventId": {
        "commitNum": 1,
        "opNum": 1
      },
      "commitTimestamp": 1571252030566,
      "data": {
        "stmt": "<https://test.com/s> <https://test.com/p> <https://test.com/o> .\n"
      },
      "op": "ADD",
      "isLastOp": true
    }
  ],
  "totalRecords": 1
}
```

AFTER_SEQUENCE_NUMBER Change Log

The following example shows a Gremlin or openCypher AFTER_SEQUENCE_NUMBER change log.

```
curl -s "https://Neptune-DNS:8182/propertygraph/stream?
limit=1&commitNum=1&opNum=1&iteratorType=AFTER_SEQUENCE_NUMBER" |jq
{
  "lastEventId": {
    "commitNum": 2,
    "opNum": 1
  },
  "lastTrxTimestamp": 1560011633768,
  "format": "PG_JSON",
  "records": [
    {
      "commitTimestamp": 1560011633768,
      "eventId": {
        "commitNum": 2,
        "opNum": 1
      },
      "data": {
        "id": "d2b59bf8-0d0f-218b-f68b-2aa7b0b1904a",
```

```
        "type": "vl",
        "key": "label",
        "value": {
          "value": "vertex",
          "dataType": "String"
        }
      },
      "op": "REMOVE",
      "isLastOp": true
    }
  ],
  "totalRecords": 1
}
```

TRIM_HORIZON Change Log

The following example shows a Gremlin or openCypher TRIM_HORIZON change log.

```
curl -s "https://Neptune-DNS:8182/propertygraph/stream?
limit=1&iteratorType=TRIM_HORIZON" |jq
{
  "lastEventId": {
    "commitNum": 1,
    "opNum": 1
  },
  "lastTrxTimestamp": 1560011610678,
  "format": "PG_JSON",
  "records": [
    {
      "commitTimestamp": 1560011610678,
      "eventId": {
        "commitNum": 1,
        "opNum": 1
      },
      "data": {
        "id": "d2b59bf8-0d0f-218b-f68b-2aa7b0b1904a",
        "type": "vl",
        "key": "label",
        "value": {
          "value": "vertex",
          "dataType": "String"
        }
      },
```

```
      "op": "ADD",
      "isLastOp": true
    }
  ],
  "totalRecords": 1
}
```

LATEST Change Log

The following example shows a Gremlin or openCypher LATEST change log. Note that the API parameters `limit`, `commitNum`, and opNum are completely optional.

```
curl -s "https://Neptune-DNS:8182/propertygraph/stream?iteratorType=LATEST" | jq
{
  "lastEventId": {
    "commitNum": 21,
    "opNum": 4
  },
  "lastTrxTimestamp": 1634710497743,
  "format": "PG_JSON",
  "records": [
    {
      "commitTimestamp": 1634710497743,
      "eventId": {
        "commitNum": 21,
        "opNum": 4
      },
      "data": {
        "id": "24be4e2b-53b9-b195-56ba-3f48fa2b60ac",
        "type": "e",
        "key": "label",
        "value": {
          "value": "created",
          "dataType": "String"
        },
        "from": "4",
        "to": "5"
      },
      "op": "REMOVE",
      "isLastOp": true
    }
  ],
  "totalRecords": 1
```

```
}
```

Compression Change Log

The following example shows a Gremlin or openCypher compression change log.

```
curl -sH \
  "Accept-Encoding: gzip" \
  "https://Neptune-DNS:8182/propertygraph/stream?limit=1&commitNum=1" \
  -H "Accept-Encoding: gzip" \
  -v |gunzip -|jq
> GET /propertygraph/stream?limit=1 HTTP/1.1
> Host: localhost:8182
> User-Agent: curl/7.64.0
> Accept: /
> Accept-Encoding: gzip
*> Accept-Encoding: gzip*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=UTF-8
< Connection: keep-alive
*< content-encoding: gzip*
< content-length: 191
<
{ [191 bytes data]
Connection #0 to host localhost left intact
{
  "lastEventId": "1:1",
  "lastTrxTimestamp": 1558942160603,
  "format": "PG_JSON",
  "records": [
    {
      "commitTimestamp": 1558942160603,
      "eventId": "1:1",
      "data": {
        "id": "v1",
        "type": "vl",
        "key": "label",
        "value": {
          "value": "person",
          "dataType": "String"
        }
      },
```

```
      "op": "ADD",
      "isLastOp": true
    }
  ],
  "totalRecords": 1
}
```

Using AWS CloudFormation to Set Up Neptune-to-Neptune Replication with the Streams Consumer Application

You can use an CloudFormation template to set up the Neptune streams consumer application to support Neptune-to-Neptune replication.

Topics

- Choose an CloudFormation template for Your Region
- Add details About the Neptune streams consumer stack you're creating
- Run the CloudFormation Template
- To update the stream poller with the latest Lambda artifacts

Choose an CloudFormation template for Your Region

To launch the appropriate CloudFormation stack on the CloudFormation console, choose one of the **Launch Stack** buttons in the following table, depending on the AWS Region that you want to use.

Region	View	Launch
US East (N. Virginia)	View	Launch Stack
US East (Ohio)	View	Launch Stack
US West (N. California)	View	Launch Stack
US West (Oregon)	View	Launch Stack

Region	View	Launch
Canada (Central)	View	Launch Stack
South America (São Paulo)	View	Launch Stack
Europe (Stockholm)	View	Launch Stack
Europe (Ireland)	View	Launch Stack
Europe (London)	View	Launch Stack
Europe (Paris)	View	Launch Stack
Europe (Frankfurt)	View	Launch Stack
Middle East (Bahrain)	View	Launch Stack
Middle East (UAE)	View	Launch Stack
Israel (Tel Aviv)	View	Launch Stack
Africa (Cape Town)	View	Launch Stack
Asia Pacific (Tokyo)	View	Launch Stack
Asia Pacific (Hong Kong)	View	Launch Stack
Asia Pacific (Seoul)	View	Launch Stack
Asia Pacific (Singapore)	View	Launch Stack

Region	View	Launch
Asia Pacific (Sydney)	View	Launch Stack
Asia Pacific (Mumbai)	View	Launch Stack
China (Beijing)	View	Launch Stack
China (Ningxia)	View	Launch Stack
AWS GovCloud (US-West)	View	Launch Stack
AWS GovCloud (US-East)	View	Launch Stack

On the **Create Stack** page, choose **Next**.

Add details About the Neptune streams consumer stack you're creating

The **Specify Stack Details** page provides properties and parameters that you can use to control the setup of the application:

Stack Name – The name of the new CloudFormation stack that you're creating. You can generally use the default value, `NeptuneStreamPoller`.

Under **Parameters**, provide the following:

Network configuration for the VPC Where the streams consumer runs

- **`VPC`** – Provide the name of the VPC where the polling Lambda function will run.
- **`SubnetIDs`** – The subnets to which a network interface is established. Add subnets corresponding to your Neptune cluster.
- **`SecurityGroupIds`** – Provide the IDs of security groups that grant write inbound access to your source Neptune DB cluster.

- **`RouteTableIds`** – This is needed to create an Amazon DynamoDB endpoint in your Neptune VPC, if you do not already have one. You must provide a comma-separated list of route table IDs associated with the subnets.
- **`CreateDDBVPCEndPoint`** – A Boolean value that defaults to `true`, indicating whether or not it is necessary to create a Dynamo DB VPC endpoint. You only need to change it to `false` if you have already created a DynamoDB endpoint in your VPC.
- **`CreateMonitoringEndPoint`** – A Boolean value that defaults to `true`, indicating whether or not it is necessary to create a monitoring VPC endpoint.. You only need to change it to `false` if you have already created a monitoring endpoint in your VPC.

Stream Poller

- **`ApplicationName`** – You can generally leave this set to the default (`NeptuneStream`). If you use a different name, it must be unique.
- **`LambdaMemorySize`** – Used to set the memory size available to the Lambda poller function. The default value is 2,048 megabytes.
- **`LambdaRuntime`** – The language used in the Lambda function that retrieves items from the Neptune stream. You can set this either to `python3.9` or to `java8`.
- **`LambdaS3Bucket`** – The Amazon S3 bucket that contains Lambda code artifacts. Leave this blank unless you are using a custom Lambda polling function that loads from a different Amazon S3 bucket.
- **`LambdaS3Key`** – The Amazon S3 key that corresponds to your Lambda code artifacts. Leave this blank unless you are using a custom Lambda polling function.
- **`LambdaLoggingLevel`** – In general, leave this set to the default value, which is `INFO`.
- **`ManagedPolicies`** – Lists the managed policies to use for execution of your Lambda function. In general, leave this blank unless you are using a custom Lambda polling function.
- **`StreamRecordsHandler`** – In general, leave this blank unless you are using a custom handler for the records in Neptune streams.
- **`StreamRecordsBatchSize`** – The maximum number of records to be fetched from stream. You can use this parameter to tune performance. The default (`5000`) is a good place to start. The maximum allowable is 10,000. The higher the number, the fewer network calls are needed to read records from the stream, but the more memory is required to process the records. Lower values of this parameter result in lower throughput.

- **`MaxPollingWaitTime`** – The maximum wait time between two polls (in seconds). Determines how frequently the Lambda poller is invoked to poll the Neptune streams. Set this value to 0 for continuous polling. The maximum value is 3,600 seconds (1 hour). The default value (60 seconds) is a good place to start, depending on how fast your graph data changes.
- **`MaxPollingInterval`** – The maximum continuous polling period (in seconds). Use this to set a timeout for the Lambda polling function. The value should be in the range between 5 seconds and 900 seconds. The default value (600 seconds) is a good place to start.
- **`StepFunctionFallbackPeriod`** – The number of units of step-function-fallback-period to wait for the poller, after which the step function is called through Amazon CloudWatch Events to recover from a failure. The default (5 minutes) is a good place to start.
- **`StepFunctionFallbackPeriodUnit`** – The time units used to measure the preceding `StepFunctionFallbackPeriodUnit` (`minutes`, `hours`, or `days`). The default (`minutes`) is generally sufficient.
- **`StartingCheckpoint`** – The starting checkpoint for the stream poller. The default is `0:0`, which signifies starting from the beginning of the Neptune stream.
- **`StreamPollerInitialState`** – The initial state of the poller. The default is `ENABLED`, which means that the stream replication will start as soon as the entire stack creation is complete.

Neptune stream

- **`NeptuneStreamEndpoint`** – (*Required*) The endpoint of the Neptune source stream. This takes one of two forms:
  - **`https://`*`your DB cluster`*`:`*`port`*`/propertygraph/stream`** (or its alias, `https://`*`your DB cluster`*`:`*`port`*`/pg/stream`).
  - **`https://`*`your DB cluster`*`:`*`port`*`/sparql/stream`**.
- **`Neptune Query Engine`** – Choose Gremlin, openCypher, or SPARQL.
- **`IAMAuthEnabledOnSourceStream`** – If your Neptune DB cluster is using IAM authentication, set this parameter to `true`.
- **`StreamDBClusterResourceId`** – If your Neptune DB cluster is using IAM authentication, set this parameter to the cluster resource ID. The resource ID is not the same as the cluster ID. Instead, it takes the form: `cluster-` followed by 28 alpha-numeric characters. It can be found under **Cluster Details** in the Neptune console.

Target Neptune DB cluster

- **`TargetNeptuneClusterEndpoint`** – The cluster endpoint (hostname only) of the target backup cluster.

  Note that if you specify `TargetNeptuneClusterEndpoint`, you cannot also specify `TargetSPARQLUpdateEndpoint`.
- **`TargetNeptuneClusterPort`** – The port number for the target cluster.

  Note that if you specify `TargetSPARQLUpdateEndpoint`, the setting for `TargetNeptuneClusterPort` is ignored.
- **`IAMAuthEnabledOnTargetCluster`** – Set to true if IAM authentication is to be enabled on the target cluster.
- **`TargetAWSRegion`** – The target backup cluster's AWS region, such as `us-east-1`). You must provide this parameter only when the AWS region of the target backup cluster is different from the region of the Neptune source cluster, as in the case of cross-region replication. If the source and target regions are the same, this parameter is optional.

  Note that if the `TargetAWSRegion` value is not a valid AWS region that Neptune supports, the process fails.
- **`TargetNeptuneDBClusterResourceId`** – *Optional*: this is only needed when IAM authentication is enabled on the target DB cluster. Set to the resource ID of the target cluster.
- **`SPARQLTripleOnlyMode`** – Boolean flag that determines whether triple-only mode is enabled. In triple-only mode, there is no named-graph replication. The default value is `false`.
- **`TargetSPARQLUpdateEndpoint`** – URL of the target endpoint for SPARQL update, such as `https://abc.com/xyz`. This endpoint can be any SPARQL store that supports quad or triples.

  Note that if you specify `TargetSPARQLUpdateEndpoint`, you cannot also specify `TargetNeptuneClusterEndpoint`, and the setting of `TargetNeptuneClusterPort` is ignored.
- **`BlockSparqlReplicationOnBlankNode`** – Boolean flag which, if set to `true`, stops replication for BlankNode in SPARQL (RDF) data. The default value is `false`.

Alarm

- **`Required to create Cloud watch Alarm`** – Set this to `true` if you want to create a CloudWatch alarm for the new stack.

- **`SNS Topic ARN for Cloudwatch Alarm Notifications`** – The SNS topic ARN where CloudWatch alarm notifications should be sent (only needed if alarms are enabled).
- **`Email for Alarm Notifications`** – The email address to which alarm notifications should be sent (only needed if alarms are enabled).

For destination of the alarm notification, you can add SNS only, email only, or both SNS and email.

Run the CloudFormation Template

Now you can complete the process of provisioning a Neptune streams consumer application instance as follows:

1. In CloudFormation, on the **Specify Stack Details** page, choose **Next**.
2. On the **Options** page, choose **Next**.
3. On the **Review** page, select the first check box to acknowledge that CloudFormation will create IAM resources. Select the second check box to acknowledge `CAPABILITY_AUTO_EXPAND` for the new stack.

   > **Note**
   >
   > `CAPABILITY_AUTO_EXPAND` explicitly acknowledges that macros will be expanded when creating the stack, without prior review. Users often create a change set from a processed template so that the changes made by macros can be reviewed before actually creating the stack. For more information, see the CloudFormation CreateStack API in the *AWS CloudFormation API Reference*.

   Then choose **Create**.

To update the stream poller with the latest Lambda artifacts

You can update the stream poller with the latest Lambda code artifacts as follows:

1. In the AWS Management Console, navigate to CloudFormation and select the main parent CloudFormation stack.
2. Select the **Update** option for the stack.

3. Select **Replace current template**.
4. For the template source, choose **Amazon S3 URL** and enter the following S3 URL:

```
https://aws-neptune-customer-samples.s3.amazonaws.com/neptune-stream/
neptune_to_neptune.json
```

5. Select **Next** without changing any CloudFormation parameters.
6. Choose **Update Stack**.

The stack will now update the Lambda artifacts with the most recent ones.

Full text search in Amazon Neptune using Amazon OpenSearch Service

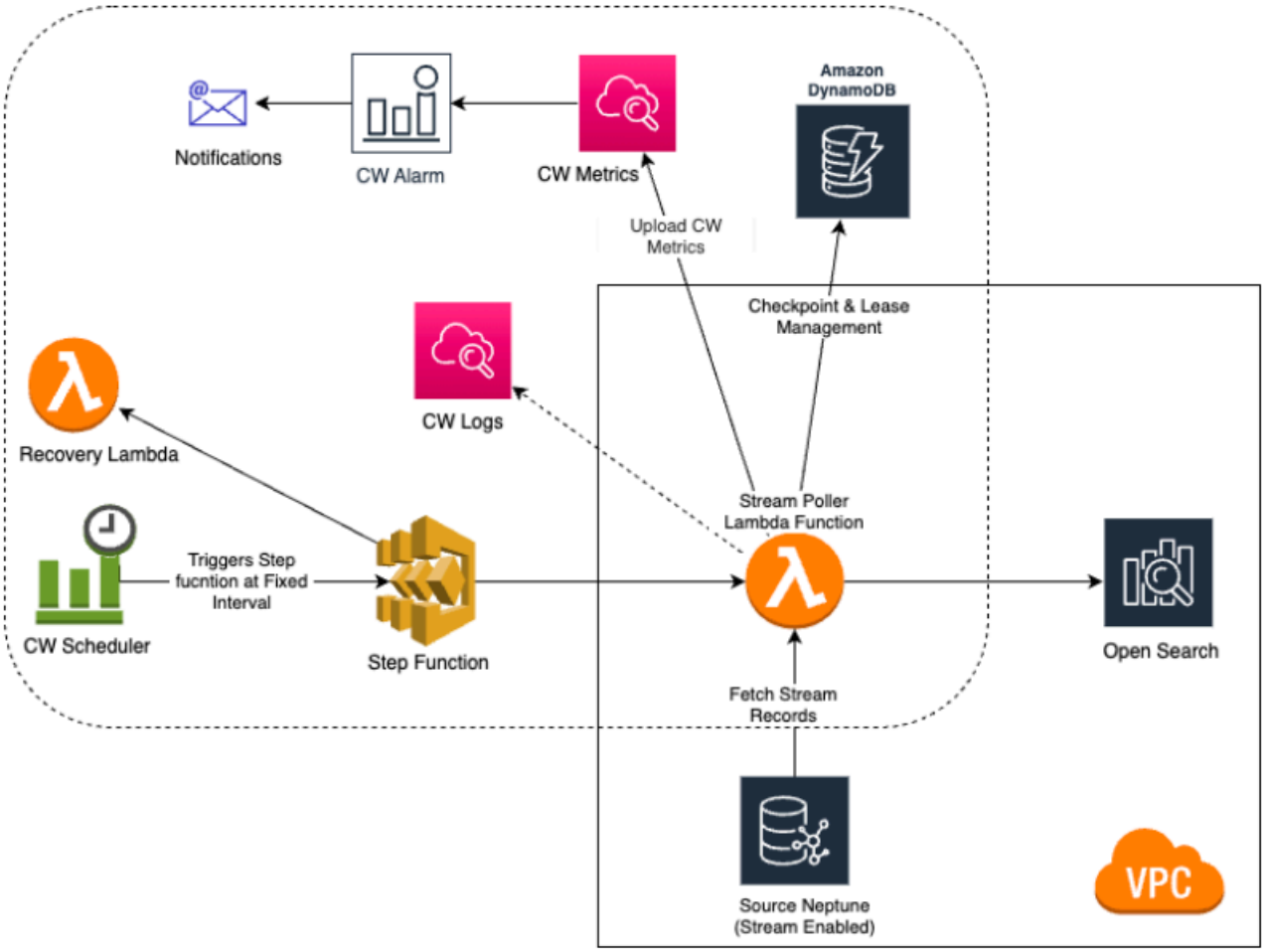
Neptune integrates with Amazon OpenSearch Service (OpenSearch Service) to support full-text search in both Gremlin and SPARQL queries.

Starting with engine release 1.3.0.0, Amazon Neptune supports using Amazon OpenSearch Service Serverless for full-text search in Gremlin and SPARQL queries.

> **Note**
>
> When integrating with Amazon OpenSearch Service, Neptune requires Elasticsearch version 7.1 or higher, and works with OpenSearch 2.3, 2.5 and above. Neptune also works with OpenSearch Serverless.

You can use Neptune with an existing OpenSearch Service cluster that has been populated according to the Neptune data model for OpenSearch data. Or, you can create an OpenSearch Service domain linked with Neptune using an AWS CloudFormation stack.

> **Important**
>
> The Neptune to OpenSearch replication process described here does not replicate blank nodes. This is an important limitation to note.
> Also, if you enable fine-grained access control on your OpenSearch cluster, you need to enable IAM authentication in your Neptune database as well.



Topics

- Amazon Neptune-to-OpenSearch replication
- Replication to OpenSearch Serverless
- Querying from an OpenSearch cluster with Fine-grained access control (FGAC) enabled
- Using Apache Lucene query syntax in Neptune full-text search queries
- Neptune data model for OpenSearch data
- Neptune full-text search parameters
- Non-string OpenSearch indexing in Amazon Neptune
- Full-text-search query execution in Amazon Neptune
- Sample SPARQL queries using full-text search in Neptune
- Using Neptune full-text search in Gremlin queries
- Troubleshooting Neptune full-text search

Amazon Neptune-to-OpenSearch replication

Amazon Neptune supports full-text search in Gremlin and SPARQL queries using Amazon OpenSearch Service (OpenSearch Service). You can use an AWS CloudFormation stack to link an OpenSearch Service domain to Neptune. The CloudFormation template creates a streams-consumer application instance that provides Neptune-to-OpenSearch replication.

Before you begin, you need an existing Neptune DB cluster with streams enabled on it to serve as the source, and an OpenSearch Service domain to serve as the replication target.

If you already have an existing target OpenSearch Service domain that can be accessed by Lambda in the VPC where your Neptune DB cluster is located, the template can use that one. Otherwise, you need to create a new one.

> **Note**
>
> The OpenSearch cluster and Lambda function that you create must be located in the same VPC as your Neptune DB cluster, and the OpenSearch cluster must be configured in VPC mode (not Internet mode).

We recommend that you use a newly created Neptune instance to use with OpenSearch Service. If you use an existing instance that already has data in it, you should perform an OpenSearch Service data sync before making queries or there may be data inconsistencies. This GitHub project provides an example of how to perform the synchronization: Export Neptune to OpenSearch (https://github.com/awslabs/amazon-neptune-tools/tree/master/export-neptune-to-elasticsearch).

> **Important**
>
> When integrating with Amazon OpenSearch Service, Neptune requires Elasticsearch version 7.1 or higher, and works with OpenSearch 2.3, 2.5 and future compatible Opensearch versions.

> **Note**
>
> Starting with engine release 1.3.0.0, Amazon Neptune supports using Amazon OpenSearch Service Serverless for full-text search in Gremlin and SPARQL queries.

Topics

- Using an AWS CloudFormation template to start Neptune-to-OpenSearch replication
- Enabling full text search on existing Neptune databases
- Updating the stream poller
- Disabling and re-enabling the stream poller process

Using an AWS CloudFormation template to start Neptune-to-OpenSearch replication

Launch an CloudFormation stack specific to your region

Each of the CloudFormation templates below creates a streams-consumer application instance in a specific AWS region. To launch the corresponding stack using the CloudFormation console, choose one of the **Launch Stack** buttons in the following table, depending on the AWS Region that you want to use.

Region	View	View in Designer	Launch
US East (N. Virginia)	View	View in Designer	Launch Stack
US East (Ohio)	View	View in Designer	Launch Stack
US West (N. Californi a)	View	View in Designer	Launch Stack
US West (Oregon)	View	View in Designer	Launch Stack
Canada (Central)	View	View in Designer	Launch Stack
South America (São Paulo)	View	View in Designer	Launch Stack
Europe (Stockholm)	View	View in Designer	Launch Stack

Region	View	View in Designer	Launch
Europe (Ireland)	View	View in Designer	Launch Stack
Europe (London)	View	View in Designer	Launch Stack
Europe (Paris)	View	View in Designer	Launch Stack
Europe (Spain)	View	View in Designer	Launch Stack
Europe (Frankfurt)	View	View in Designer	Launch Stack
Middle East (Bahrain)	View	View in Designer	Launch Stack
Middle East (UAE)	View	View in Designer	Launch Stack
Israel (Tel Aviv)	View	View in Designer	Launch Stack
Africa (Cape Town)	View	View in Designer	Launch Stack
Asia Pacific (Hong Kong)	View	View in Designer	Launch Stack
Asia Pacific (Tokyo)	View	View in Designer	Launch Stack
Asia Pacific (Seoul)	View	View in Designer	Launch Stack
Asia Pacific (Singapore)	View	View in Designer	Launch Stack
Asia Pacific (Jakarta)	View	View in Designer	Launch Stack

Region	View	View in Designer	Launch
Asia Pacific (Mumbai)	View	View in Designer	Launch Stack
China (Beijing)	View	View in Designer	Launch Stack
China (Ningxia)	View	View in Designer	Launch Stack
AWS GovCloud (US-West)	View	View in Designer	Launch Stack
AWS GovCloud (US-East)	View	View in Designer	Launch Stack

On the **Create Stack** page, choose **Next**.

Add Details About the new OpenSearch stack you are creating

The **Specify Stack Details** page provides properties and parameters that you can use to control the setup of full-text search:

Stack Name – The name of the new CloudFormation stack that you're creating. You can generally use the default value, `NeptuneStreamPoller`.

Under **Parameters**, provide the following:

Network Configuration for the VPC Where the Streams Consumer Runs

- **`VPC`** – Provide the name of the VPC where the polling Lambda function will run.
- **`List of Subnet IDs`** – The subnets to which a network interface is established. Add subnets corresponding to your Neptune cluster.
- **`List of Security Group Ids`** – Provide the IDs of security groups that grant write inbound access to your source Neptune DB cluster.
- **`List of Route Table Ids`** – This is needed to create an Amazon DynamoDB endpoint in your Neptune VPC, if you do not already have one. You must provide a comma-separated list of route table IDs associated with the subnets.

- **`Require to create Dynamo DB VPC Endpoint`** – A Boolean value that defaults to `true`. You only need to change it to `false` if you have already created a DynamoDB endpoint in your VPC.
- **`Require to create Monitoring VPC Endpoint`** – A Boolean value that defaults to `true`. You only need to change it to `false` if you have already created a monitoring endpoint in your VPC.

Stream Poller

- **`Application Name`** – You can generally leave this set to the default (`NeptuneStream`). If you use a different name, it must be unique.
- **`Memory size for Lambda Poller`** – Used to set the memory size available to the Lambda poller function. The default value is 2,048 megabytes.
- **`Lambda Runtime`** – The language used in the Lambda function that retrieves items from the Neptune stream. You can set this either to `python3.9` or to `java8`.
- **`S3 Bucket having Lambda code artifacts`** – Leave this blank unless you are using a custom Lambda polling function that loads from a different S3 bucket.
- **`S3 Key corresponding to Lambda Code artifacts`** – Leave this blank unless you are using a custom Lambda polling function.
- **`StartingCheckpoint`** – the starting checkpoint for the stream poller. The default is `0:0`, which signifies starting from the beginning of the Neptune stream.
- **`StreamPollerInitialState`** – The initial state of the poller. The default is `ENABLED`, which means that the stream replication will start as soon as the entire stack creation is complete.
- **`Logging level for Lambda`** – In general, leave this set to the default value, `INFO`.
- **`Managed Policies for Lambda Execution`** – In general, leave this blank unless you are using a custom Lambda polling function.
- **`Stream Records Handler`** – In general, leave this blank unless you are using a custom handler for the records in Neptune streams.
- **`Maximum records Fetched from Stream`** – You can use this parameter to tune performance. The default (`100`) is a good place to start. The maximum allowable is 10,000. The higher the number, the fewer network calls are needed to read records from the stream, but the more memory is required to process the records.
- **`Max wait time between two Polls (in Seconds)`** – Determines how frequently the Lambda poller is invoked to poll the Neptune streams. Set this value to 0 for continuous polling.

The maximum value is 3,600 seconds (1 hour). The default value (60 seconds) is a good place to start, depending on how fast your graph data changes.

- **`Maximum Continuous polling period (in Seconds)`** – Used to set a timeout for the Lambda polling function. It should be between 5 seconds and 900 seconds. The default value (600 seconds) is a good place to start.
- **`Step Function Fallback Period`** – The number of step-function-fallback-period units to wait for the poller, after which the step function is called through Amazon CloudWatch Events to recover from a failure. The default (5 minutes) is a good place to start.
- **`Step Function Fallback Period Unit`** – The time units used to measure the preceding `Step Function Fallback Period` (minutes, hours, days). The default (minutes) is generally sufficient.
- **`Enable/disable creation of Elastic Search Mappings by Lambda`** – Flag to enable/disable automatic management of Elastic Search mappings by the stream poller. Elastic Search mappings are critical to conversion between Neptune and Elastic Search data types. This should only be disabled by advanced users who wish to assume direct, manual control over all mappings in their Elastic Search index. The default is `true`.
- **`Data replication scope`** – Determines whether to replicate both nodes and edges, or only nodes to OpenSearch (this applies to Gremlin engine data only). The default value (All) is generally a good place to start.
- **`Ignore OpenSearch missing document error`** – Flag to determine whether a missing document error in OpenSearch can be ignored. Missing document errors occur rarely but need manual intervention if not ignored. The default value (`True`) is generally a good place to start.
- **`Enable Non-String Indexing`** – Flag to enable or disable indexing of fields that do not have string content. If this flag is set to `true`, non-string fields are indexed in OpenSearch, or if `false`, only string fields are indexed. The default is `true`.
- **`Properties to exclude from being inserted into OpenSearch`** – A comma-delimited list of property or predicate keys to exclude from OpenSearch indexing. If this CFN parameter value is left blank, all the property keys are indexed.
- **`Datatypes to exclude from being inserted into OpenSearch`** – A comma-delimited list of property or predicate datatypes to exclude from OpenSearch indexing. If this CFN parameter value is left blank, all the property values that can safely be converted to OpenSearch datatypes are indexed.

Neptune Stream

- **`Endpoint of source Neptune Stream`** – (*Required*) This takes one of two forms:
  - **`https://`*`your DB cluster`*`:`*`port`*`/propertygraph/stream`** (or its alias, `https://`*`your DB cluster`*`:`*`port`*`/pg/stream`).
  - `https://`*`your DB cluster`*`:`*`port`*`/sparql/stream`
- **`Neptune Query Engine`** – Choose Gremlin or SPARQL.
- **`Is IAM Auth Enabled?`** – If your Neptune DB cluster is using IAM authentication, set this parameter to `true`.
- **`Neptune Cluster Resource Id`** – If your Neptune DB cluster is using IAM authentication, set this parameter to the cluster resource ID. The resource ID is not the same as the cluster ID. Instead, it takes the form: `cluster-` followed by 28 alpha-numeric characters. It can be found under **Cluster Details** in the Neptune console.

Target OpenSearch cluster

- **`Endpoint for OpenSearch service`** – (Required) Provide the endpoint for the OpenSearch service in your VPC.
- **`Number of Shards for OpenSearch Index`** – The default value (5) is generally a good place to start.
- **`Number of Replicas for OpenSearch Index`** – The default value (1) is generally a good place to start.
- **`Geo Location Fields for Mapping`** – If you are using geolocation fields, list the property keys here.

Alarm

- **`Require to create Cloud watch Alarm`** – Set this to `true` if you want to create a CloudWatch alarm for the new stack.
- **`SNS Topic ARN for Cloudwatch Alarm Notifications`** – The SNS topic ARN where CloudWatch alarm notifications should be sent (only needed if alarms are enabled).
- **`Email for Alarm Notifications`** – The email address to which alarm notifications should be sent (only needed if alarms are enabled).

For destination of the alarm notification, you can add SNS only, email only, or both SNS and email.

Run the CloudFormation Template

Now you can complete the process of provisioning a Neptune streams consumer application instance as follows:

1. In CloudFormation, on the **Specify Stack Details** page, choose **Next**.
2. On the **Options** page, choose **Next**.
3. On the **Review** page, select the first check box to acknowledge that CloudFormation will create IAM resources. Select the second check box to acknowledge `CAPABILITY_AUTO_EXPAND` for the new stack.

   > **Note**
   >
   > `CAPABILITY_AUTO_EXPAND` explicitly acknowledges that macros will be expanded when creating the stack, without prior review. Users often create a change set from a processed template so that the changes made by macros can be reviewed before actually creating the stack. For more information, see the CloudFormation CreateStack API operation in the *AWS CloudFormation API Reference*.

   Then choose **Create**.

Enabling full text search on existing Neptune databases

These are the established approaches to enabling full text search on existing Amazon Neptune databases. Depending on whether you can pause your write workloads or not, the steps may vary slightly. This guide outlines the recommended steps for both scenarios - when you can pause writes, and when you cannot. It covers enabling Neptune streams, creating a database clone, synchronizing data to an OpenSearch domain, and setting up continuous updates. The guidance leverages AWS services and open-source tools to streamline the process and minimize downtime.

If you can pause your write workloads

The best way to enable full text search on an existing Neptune database is generally as follows, provided you can pause your write workloads. It requires creating a clone, enabling the streams

using a cluster parameter, and restarting all the instances. Creating a clone is a relatively fast operation, so the downtime required is limited.

Here are the steps required:

1. Stop all write workloads on the database.
2. Enable streams on the database (see Enabling Neptune Streams).
3. Create a clone of the database (see Database Cloning in Neptune).
4. Resume the write workloads.
5. Use the export-neptune-to-elasticsearch tool on github to perform a one-time synchronization from the cloned database to the OpenSearch domain.
6. Use the CloudFormation template for your region to start synchronization from your original database with continuous updating (no configuration change is needed in the template).
7. Delete the cloned database and the CloudFormation stack created for the `export-neptune-to-elasticsearch` tool.

> **Note**
>
> If using export-neptune-to-elasticsearch with Opensearch Serverless, the `LambdaExecutionRole` created by the export-neptune-to-elasticsearch Cloudformation stack must be added to the Opensearch Data Access Policy to successfully replicate the data.

If you cannot pause your write workloads

If you can't afford to suspend write workloads on your database, here is an approach that requires even less downtime than the recommended approach above, but it needs to be done carefully:

1. Enable streams on the database (see Enabling Neptune Streams).
2. Create a clone of the database (see Database Cloning in Neptune).
3. Get the latest `eventID` for the streams on the cloned database by executing a command of this kind against the Streams API endpoint (see Calling the Neptune Streams REST API for more information):

```
curl "https://(your neptune endpoint):(port)/(propertygraph or sparql)/stream?
iteratorType=LATEST"
```

Make a note of the values in the `commitNum` and `opNum` fields in the `lastEventId` object in the response.

4. Use the export-neptune-to-elasticsearch tool on github to perform a one-time synchronization from the cloned database to the OpenSearch domain.
5. Use the CloudFormation template for your region to start synchronization from your original database with continuous updating.

   Make the following change while creating the stack: on the stack details page, in the **Parameters** section, set the value of the `StartingCheckpoint` field to *`commitNum`*:*`opnum`* using the the `commitNum` and `opNum` values you recorded above.
6. Delete the cloned database and the CloudFormation stack created for the `export-neptune-to-elasticsearch` tool.

Updating the stream poller

The following information outlines the steps needed to update the stream poller with the latest Lambda artifacts using the AWS management console.

To update the stream poller with the latest Lambda artifacts

You can update the stream poller with the latest Lambda code artifacts as follows:

1. In the AWS Management Console, navigate to CloudFormation and select the main parent CloudFormation stack.
2. Select the **Update** option for the stack.
3. Select **Replace current template**.
4. For the template source, choose **Amazon S3 URL** and enter the following S3 URL:

   ```
   https://aws-neptune-customer-samples.s3.amazonaws.com/neptune-stream/
   neptune_to_elastic_search.json
   ```
5. Select **Next** without changing any CloudFormation parameters.
6. Choose **Update Stack**.

The stack will now update the Lambda artifacts with the most recent ones.

Extending the stream poller to support custom fields

The current stream poller can easily be extended to write custom code for handling custom fields, as is explained in detail in this blog post: Capture graph changes using Neptune Streams.

> **Note**
>
> When adding a custom field in OpenSearch, make sure to add the new field as an inner object of a predicate (see Neptune Full-text search data model).

Disabling and re-enabling the stream poller process

> **Warning**
>
> Be careful when you disable the stream poller process! Data loss can occur if the process is paused for longer than the stream expiry window. The default window is 7 days, but starting with engine version 1.2.0.0, you can set a custom stream expiry window up to a maximum of 90 days.

Disabling (pausing) the stream poller process

1. Sign in to the AWS Management Console and open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, select **Rules**.
3. Select the rule whose name contains the name you supplied as **Application Name** in the CloudFormation template that you used to set up the stream poller.
4. Choose **Disable**.
5. Open the Step Functions console at https://eusc-de-east-1.console.amazonaws-eusc.eu/states/.
6. Select the running step function that corresponds to the stream poller process. Again, the name of that step function contains the name you supplied as **Application Name** in the CloudFormation template that you used to set up the stream poller. You can filter by function execution status to see only **Running** functions.

7. Choose **Stop**.

Re-enabling the stream poller process

1. Sign in to the AWS Management Console and open the Amazon EventBridge console at https://eusc-de-east-1.console.amazonaws-eusc.eu/events/.
2. In the navigation pane, select **Rules**.
3. Select the rule whose name contains the name you supplied as **Application Name** in the CloudFormation template that you used to set up the stream poller.
4. Choose **Enable**. The event rule based on the specified scheduled interval will now trigger a new execution of the step function.

Replication to OpenSearch Serverless

Starting with engine release 1.3.0.0, Amazon Neptune supports using Amazon OpenSearch Service Serverless for full-text search in Gremlin and SPARQL queries. Using OpenSearch Serverless requires you to enable IAM authentication on your Neptune Database cluster. Neptune Database clusters with IAM authentication disabled are not supported with OpenSearch Serverless.

Changes required for poller AWS Lambda function

If you are replicating to OpenSearch Serverless, add the Lambda stream poller execution role to the data access policy for the OpenSearch Serverless collection. The ARN for the Lambda stream poller execution role has this format:

```
arn:aws:iam::(account ID):role/stack-name-NeptuneOSReplication-
NeptuneStreamPollerExecu-(uuid)
```

Changes required for bulk import utility

If you are using export-neptune-to-elasticsearch to synchronize existing data to OpenSearch Serverless, add the `LambdaExecutionRole` from the CloudFormation stack to the data access policy for the OpenSearch Serverless collection. The ARN for the `LambdaExecutionRole` has this format:

```
arn:aws:iam::012345678901:role/stack-name-LambdaExecutionRole-(id)
```

For more information, see Data access control for Amazon OpenSearch Serverless.

Changes needed to the IAM role used to query Neptune

The IAM entity (User or Role) used for connecting to the Neptune database should have permissions both for Neptune and the OpenSearch Serverless collection. This means that your user or role must have an OpenSearch Serverless policy like this attached:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowOpenSearchServerlessAccess",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws-eusc:iam::111122223333:root"
      },
      "Action": "aoss:APIAccessAll",
      "Resource": "arn:aws-eusc:aoss:eusc-de-
east-1:111122223333:collection/collection-id"
    }
  ]
}
```

See Creating custom IAM policy statements to access data in Amazon Neptune for more information.

Querying from an OpenSearch cluster with Fine-grained access control (FGAC) enabled

If you have enabled fine-grained access control on your OpenSearch cluster, you need to enable IAM authentication in your Neptune database as well.

The IAM entity (User or Role) used for connecting to the Neptune database should have permissions both for Neptune and the OpenSearch cluster. This means that your user or role must have an OpenSearch Service policy like this attached:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowFullTextSearchAccess",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws-eusc:iam::111122223333:root"
      },
      "Action": "es:*",
      "Resource": "arn:aws-eusc:es:eusc-de-east-1:111122223333:domain/domain-
name/*"
    }
  ]
}
```

See Creating custom IAM policy statements to access data in Amazon Neptune for more information.

Using Apache Lucene query syntax in Neptune full-text search queries

OpenSearch supports using Apache Lucene syntax for query_string queries. This is particularly useful for passing multiple filters in a query.

Neptune uses a nested structure for storing properties in an OpenSearch document (see Neptune Full-text search data model). When using Lucene syntax, you need to use full paths to the properties in this nexted model.

Here is a Gremlin example:

```
g.withSideEffect("Neptune#fts.endpoint", "es_endpoint")
 .withSideEffect("Neptune#fts.queryType", "query_string")
 .V()
 .has("*", "Neptune#fts predicates.name.value:\"Jane Austin\" AND entity_type:Book")
```

Here is a SPARQL example:

```
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
  neptune-fts:config neptune-fts:endpoint 'http://localhost:9200 (http://
localhost:9200/)' .
  neptune-fts:config neptune-fts:queryType 'query_string' .
  neptune-fts:config neptune-fts:query "predicates.\\*foaf\\*name.value:Ronak AND
 predicates.\\*foaf\\*surname.value:Sh*" .
  neptune-fts:config neptune-fts:field '*' .
  neptune-fts:config neptune-fts:return ?res .
}
```

Neptune data model for OpenSearch data

Amazon Neptune uses a unified JSON document structure for storing both SPARQL and Gremlin data in OpenSearch Service. Each document in OpenSearch corresponds to an entity and stores all the relevant information for that entity. For Gremlin, vertexes and edges are considered entities, so the corresponding OpenSearch documents have information about vertexes, labels, and properties. For SPARQL, subjects can be considered entities, so corresponding OpenSearch documents have information about all the predicate-object pairs in one document.

> **Note**
>
> The Neptune-to-OpenSearch replication implementation only stores string data. However, you can modify it to store other data types.

The unified JSON document structure looks like the following.

```
{
  "entity_id": "Vertex Id/Edge Id/Subject URI",
  "entity_type": [List of Labels/rdf:type object value],
  "document_type": "vertex/edge/rdf-resource"
```

```
  "predicates": {
    "Property name or predicate URI": [
      {
        "value": "Property Value or Object Value",
        "graph": "(Only for Sparql) Named Graph Quad is present"
        "language": "(Only for Sparql) rdf:langString"
      },
      {
        "value": "Property Value 2/ Object Value 2",
      }
    ]
  }
}
```

- `entity_id` – Entity unique ID representing the document.
  - For SPARQL, this is the subject URI.
  - For Gremlin, this is the `Vertex_ID` or `Edge_ID`.
- `entity_type` – Represents one or more labels for a vertex or edge, or zero or more `rdf:type` predicate values for a subject.
- `document_type` – Used to specify whether the current document represents a vertex, edge, or rdf-resource.
- `predicates` – For Gremlin, stores properties and values for a vertex or edge. For SPARQL, it stores predicate-object pairs.

  The property name takes the form `properties.name.value` in OpenSearch. To query it, you have to name it in that form.
- `value` – A property value for Gremlin or an object value for SPARQL.
- `graph` – A named graph for SPARQL.
- `language` – A language tag for a `rdf:langString` literal in SPARQL.

Sample SPARQL OpenSearch document

Data

```
@prefix dt:   <http://example.org/datatype#> .
@prefix ex:   <http://example.org/> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
```

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:simone   rdf:type     ex:Person                   ex:g1
ex:michael  rdf:type     ex:Person                   ex:g1
ex:simone   ex:likes     "spaghetti"                 ex:g1

ex:simone   ex:knows     ex:michael                  ex:g2   # Not stored in ES
ex:simone   ex:likes     "spaghetti"                 ex:g2
ex:simone   ex:status    "La vita è un sogno"@it     ex:g2

ex:simone   ex:age       "40"^^xsd:int               DG      # Not stored in ES
ex:simone   ex:dummy     "testData"^^dt:newDataType  DG      # Not stored in ES
ex:simone   ex:hates     _:bnode                             # Not stored in ES
_:bnode     ex:means     "coding"                    DG      # Not stored in ES
```

Documents

```
{
  "entity_id": "http://example.org/simone",
  "entity_type": ["http://example.org/Person"],
  "document_type": "rdf-resource"
  "predicates": {
    "http://example.org/likes": [
      {
        "value": "spaghetti",
        "graph": "http://example.org/g1"
      },
      {
        "value": "spaghetti",
        "graph": "http://example.org/g2"
      }
    ]
    "http://example.org/status": [
      {
        "value": "La vita è un sogno",
        "language": "it"       // Only present for rdf:langString
      }
    ]
  }
}
```

```
{
  "entity_id" : "http://example.org/michael",
```

```
  "entity_type" : ["http://example.org/Person"],
  "document_type": "rdf-resource"
}
```

Sample Gremlin OpenSearch document

Data

```
# Vertex 1
simone   label    Person       <== Label
simone   likes    "spaghetti"  <== Property
simone   likes    "rice"       <== Property
simone   age      40           <== Property

# Vertex 2
michael  label    Person       <== Label

# Edge 1
simone  knows     michael      <== Edge
e1      updated  "2019-07-03"  <== Edge Property
e1      through  "company"     <== Edge Property
e1      since     10           <== Edge Property
```

Documents

```
{
  "entity_id": "simone",
  "entity_type": ["Person"],
  "document_type": "vertex",
  "predicates": {
    "likes": [
      {
        "value": "spaghetti"
      },
      {
        "value": "rice"
      }
    ]
  }
}
```

```
{
```

```
  "entity_id" : "michael",
  "entity_type" : ["Person"],
  "document_type": "vertex"
}
```

```
{
  "entity_id": "e1",
  "entity_type": ["knows"],
  "document_type": "edge"
  "predicates": {
    "through": [
      {
        "value": "company"
      }
    ]
  }
}
```

Neptune full-text search parameters

Amazon Neptune uses the following parameters for specifying full-text OpenSearch queries in both Gremlin and SPARQL:

- **`queryType`** – (*Required*) The type of OpenSearch query. (For a list of query types, see the OpenSearch documentation). Neptune supports the following OpenSearch query types:
  - simple_query_string – Returns documents based on a provided query string, using a parser with a limited but fault-tolerant Lucene syntax. This is the default query type.

    This query uses a simple syntax to parse and split the provided query string into terms based on special operators. The query then analyzes each term independently before returning matching documents.

    While its syntax is more limited than the `query_string` query, the `simple_query_string` query does not return errors for invalid syntax. Instead, it ignores any invalid parts of the query string.
  - match – The `match` query is the standard query for performing a full-text search, including options for fuzzy matching.
  - prefix – Returns documents that contain a specific prefix in a provided field.

- fuzzy – Returns documents that contain terms similar to the search term, as measured by a Levenshtein edit distance.

  An edit distance is the number of one-character changes needed to turn one term into another. These changes can include:

  - Changing a character (box to fox).
  - Removing a character (black to lack).
  - Inserting a character (sic to sick).
  - Transposing two adjacent characters (act to cat).

  To find similar terms, the fuzzy query creates a set of all possible variations and expansions of the search term within a specified edit distance and then returns exact matches for each of those variants.

- term – Returns documents that contain an exact match of a specified term in one of the specified fields.

  You can use the `term` query to find documents based on a precise value such as a price, a product ID, or a username.

  > **Warning**
  >
  > Avoid using the term query for text fields. By default, OpenSearch changes the values of text fields as part of its analysis, which can make finding exact matches for text field values difficult.
  > To search text field values, use the match query instead.

- query_string – Returns documents based on a provided query string, using a parser with a strict syntax (Lucene syntax).

  This query uses a syntax to parse and split the provided query string based on operators, such as AND or NOT. The query then analyzes each split text independently before returning matching documents.

  You can use the `query_string` query to create a complex search that includes wildcard characters, searches across multiple fields, and more. While versatile, the query is strict and returns an error if the query string includes any invalid syntax.

> **Warning**
>
> Because it returns an error for any invalid syntax, we don't recommend using the `query_string` query for search boxes.
> If you don't need to support a query syntax, consider using the `match` query. If you need the features of a query syntax, use the `simple_query_string` query, which is less strict.

- **`field`** – The field in OpenSearch against which to run the search. This can be omitted only if the `queryType` allows it (as `simple_query_string` and `query_string` do), in which case the search is against all fields. In Gremlin, it is implicit.

  Multiple fields can be specified if the query allows it, as do `simple_query_string` and `query_string`.
- **`query`** – (*Required*) The query to run against OpenSearch. The contents of this field might vary according to the queryType. Different queryTypes accept different syntaxes, as `Regexp` does, for example. In Gremlin, `query` is implicit.
- **`maxResults`** – The maximum number of results to return. The default is the `index.max_result_window` OpenSearch setting, which itself defaults to 10,000. The `maxResults` parameter can specify any number lower than that.

> **Important**
>
> If you set `maxResults` to a value higher than the OpenSearch `index.max_result_window` value and try to retrieve more than `index.max_result_window` results, OpenSearch fails with a `Result window is too large` error. However, Neptune handles this gracefully without propagating the error. Keep this in mind if you are trying to fetch more than `index.max_result_window` results.

- **`minScore`** – The minimum score a search result must have to be returned. See OpenSearch relevance documentation for an explanation of result scoring.
- **`batchSize`** – Neptune always fetches data in batches (the default batch size is 100). You can use this parameter to tune performance. The batch size cannot exceed the `index.max_result_window` OpenSearch setting, which defaults to 10,000.

- **`sortBy`** – An optional parameter that lets you sort the results returned by OpenSearch by one of the following:
  - *A particular string field in the document* –

    For example, in a SPARQL query, you could specify:

    ```
    neptune-fts:config neptune-fts:sortBy foaf:name .
    ```

    In a similar Gremlin query, you could specify:

    ```
    .withSideEffect('Neptune#fts.sortBy', 'name')
    ```

  - *A particular non-string field (`long`, `double`, etc.) in the document* –

    Note that when sorting on a non-string field, you need to append `.value` to the field name to differentiate it from a string field.

    For example, in a SPARQL query, you could specify:

    ```
    neptune-fts:config neptune-fts:sortBy foaf:name.value .
    ```

    In a similar Gremlin query, you could specify:

    ```
    .withSideEffect('Neptune#fts.sortBy', 'name.value')
    ```

  - `score` – Sort by match score (the default).

    If the `sortOrder` parameter is present but `sortBy` is not present, the results are sorted by `score` in the order specified by `sortOrder`.

  - `id` – Sort by ID, which means the SPARQL subject URI or the Gremlin vertex or edge ID.

    For example, in a SPARQL query, you could specify:

    ```
    neptune-fts:config neptune-fts:sortBy 'Neptune#fts.entity_id' .
    ```

    In a similar Gremlin query, you could specify:

    ```
    .withSideEffect('Neptune#fts.sortBy', 'Neptune#fts.entity_id')
    ```

- `label` – Sort by label.

  For example, in a SPARQL query, you could specify:

  ```
  neptune-fts:config neptune-fts:sortBy 'Neptune#fts.entity_type' .
  ```

  In a similar Gremlin query, you could specify:

  ```
  .withSideEffect('Neptune#fts.sortBy', 'Neptune#fts.entity_type')
  ```

- `doc_type` – Sort by document type (that is, SPARQL or Gremlin).

  For example, in a SPARQL query, you could specify:

  ```
  neptune-fts:config neptune-fts:sortBy 'Neptune#fts.document_type' .
  ```

  In a similar Gremlin query, you could specify:

  ```
  .withSideEffect('Neptune#fts.sortBy', 'Neptune#fts.document_type')
  ```

By default, OpenSearch results are not sorted and their order is non-deterministic, meaning that the same query may return items in a different order each time it is run. For this reason, if the result set is greater than `max_result_window`, a quite different subset of the total results could be returned every time a query is run. By sorting, however, you can make the results of different runs more directly comparable.

If no `sortOrder` parameter accompanies `sortBy`, descending (`DESC`) order from greatest to least is used.

- **`sortOrder`** – An optional parameter that lets you specify whether OpenSearch results are sorted from least to greatest or from greatest to least (the default):

  - `ASC` – Ascending order, from least to greatest.
  - `DESC` – Descending order, from greatest to least.

    This is the default value, used when the `sortBy` parameter is present but no `sortOrder` is specified.

~~If neither `sortBy` nor `sortOrder` is present, OpenSearch results are not sorted by default.~~

Non-string OpenSearch indexing in Amazon Neptune

Non-string OpenSearch indexing in Amazon Neptune allows replicating non-string values for predicates to OpenSearch using the stream poller. All predicate values that can safely be converted to a corresponding OpenSearch mapping or datatype is then replicated to OpenSearch.

For non-string indexing to be enabled on a new stack, the `Enable Non-String Indexing` flag in the CloudFormation template must be set to `true`. This is the default setting. To update an existing stack to support non-string indexing, see Updating an existing stack below.

> **Note**
>
> - It is best not to enable non-string indexing on engine versions earlier than **`1.0.4.2`**.
> - OpenSearch queries using regular expressions for field names that match multiple fields, some of which contain string values and others of which contain non-string values, fail with an error. The same thing happens if full-text search queries in Neptune are of that type.
> - When sorting by a non-string field, append ".value" to the field name to differentiate it from a string field.

Contents

- Updating an existing Neptune full-text search stack to support non-string indexing
- Filtering what fields are indexed in Neptune full-text search
  - Filter by property or predicate name
  - Filter by property or predicate value type
- Mapping of SPARQL and Gremlin datatypes to OpenSearch
- Validation of data mappings
- Sample non-string OpenSearch queries in Neptune
  - Get all vertices with age greater than 30 and name starting with "Si"
  - Get all nodes with age between 10 and 50 and a name with a fuzzy match with "Ronka"
  - Get all nodes with a timestamp that falls within the last 25 days
  - Get all nodes with a timestamp that falls within a given year and month

Updating an existing Neptune full-text search stack to support non-string indexing

If you are already using Neptune full-text search, here are the steps you need to take to support non-string indexing:

1. **Stop the stream poller Lambda function.** This ensures that no new updates are copied during export. Do this by disabling the cloud event rule that invokes the Lambda function:

   - In the AWS Management Console, navigate to CloudWatch.
   - Select **Rules**.
   - Choose the rule with the Lambda stream poller name.
   - Select **disable** to temporarily disable the rule.
2. **Delete the current Neptune index in OpenSearch.** Use the following `curl` query to delete the `amazon_neptune` index from your OpenSearch cluster:

   ```
   curl -X DELETE "your OpenSearch endpoint/amazon_neptune"
   ```
3. **Start a one-time export from Neptune to OpenSearch.** It is best to set up a new OpenSearch stack at this point, so that new artifacts are picked up for the poller that performs the export.

   Follow the steps listed here in GitHub to start the one-time export of your Neptune data into OpenSearch.
4. **Update the Lambda artifacts for the existing stream poller.** After the export of Neptune data to OpenSearch has completed successfully, take the following steps:

   - In the AWS Management Console, navigate to CloudFormation.
   - Choose the main parent CloudFormation stack.
   - Select the **Update** option for that stack.
   - Select **Replace current template from options**.
   - For the template source, select **Amazon S3 URL**.
   - For the Amazon S3 URL, enter:

     ```
     https://aws-neptune-customer-samples.s3.amazonaws.com/neptune-stream/
     neptune_to_elastic_search.json
     ```
   - Choose **Next** without changing any of the CloudFormation parameters.

- Select **Update stack**. CloudFormation will replace the Lambda code artifacts for the stream poller with the latest artifacts.

5. **Start the stream poller again.** Do this by enabling the appropriate CloudWatch rule:

- In the AWS Management Console, navigate to CloudWatch.
- Select **Rules**.
- Choose the rule with the Lambda stream poller name.
- Select **enable**.

Filtering what fields are indexed in Neptune full-text search

There are two fields in the CloudFormation template details that let you specify property or predicate keys or datatypes to exclude from OpenSearch indexing:

Filter by property or predicate name

You can use the optional CloudFormation template parameter named `Properties to exclude from being inserted into Elastic Search Index` to provide a comma-delimited list of property or predicate keys to exclude from OpenSearch indexing.

For example, suppose you set this parameter to bob:

```
"Properties to exclude from being inserted into Elastic Search Index" : bob
```

In that case, the stream record of the following Gremlin update query would be dropped rather than going into the index:

```
g.V("1").property("bob", "test")
```

Similarly, you could set the parameter to `http://my/example#bob`:

```
"Properties to exclude from being inserted into Elastic Search Index" : http://my/
example#bob
```

In that case, the stream record of the following SPARQL update query would be dropped rather than going into the index:

```
PREFIX ex: <http://my/example#>
INSERT DATA { ex:s1 ex:bob "test"}.
```

If you don't enter anything in this CloudFormation template parameter, all the property keys not otherwise excluded will be indexed.

Filter by property or predicate value type

You can use the optional CloudFormation template parameter named `Datatypes to exclude from being inserted into Elastic Search Index` to provide a comma-delimited list of property or predicate value datatypes to exclude from OpenSearch indexing.

For SPARQL, you don't need to list the full XSD type URI, you can just list the datatype token. Valid datatype tokens that you can list are:

- `string`
- `boolean`
- `float`
- `double`
- `dateTime`
- `date`
- `time`
- `byte`
- `short`
- `int`
- `long`
- `decimal`
- `integer`
- `nonNegativeInteger`
- `nonPositiveInteger`
- `negativeInteger`
- `unsignedByte`

- `unsignedShort`
- `unsignedInt`
- `unsignedLong`

For Gremlin, valid datatypes to list are:

- `string`
- `date`
- `bool`
- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`

For example, suppose you set this parameter to `string`:

```
"Datatypes to exclude from being inserted into Elastic Search Index" : string
```

In that case, the stream record of the following Gremlin update query would be dropped rather than going into the index:

```
g.V("1").property("myStringval", "testvalue")
```

Similarly, you could set the parameter to `int`:

```
"Datatypes to exclude from being inserted into Elastic Search Index" : int
```

In that case, the stream record of the following SPARQL update query would be dropped rather than going into the index:

```
PREFIX ex: <http://my/example#>
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
```

```
INSERT DATA { ex:s1 ex:bob "11"^^xsd:int }.
```

If you don't enter anything in this CloudFormation template parameter, all the properties whose values can be safely converted to OpenSearch equivalents will be indexed. Listed types that are unsupported by the query language are ignored.

Mapping of SPARQL and Gremlin datatypes to OpenSearch

New datatype mappings in OpenSearch are created based on the datatype being used in the property or object. Because some fields contain values of different types, the initial mapping may exclude some values of the field.

Neptune datatypes map to OpenSearch datatypes as follows:

SPARQL types	Gremlin types	OpenSearch types
XSD:int	byte	long
XSD:unsignedInt	short	
XSD:integer	int	
XSD:byte	long	
XSD:unsignedByte		
XSD:short		
XSD:unsignedShort		
XSD:long		
XSD:unsignedLong		
XSD:float	float	double
XSD:double	double	
XSD:decimal		
XSD:boolean	bool	boolean

SPARQL types	Gremlin types	OpenSearch types

`XSD:datetime`
 	
`XSD:date`
	`date`	`date`

`XSD:string`
 	
`XSD:time`
	`string`	`text`

Custom datatype	*N/A*	`text`
Any other datatype	*N/A*	`text`

For example, the following Gremlin update query causes a new mapping for "newField" to be added to OpenSearch, namely `{ "type" : "double" }`:

```
g.V("1").property("newField" 10.5)
```

Similarly, the following SPARQL update query causes a new mapping for "ex:byte" to be added to OpenSearch, namely `{ "type" : "long" }`:

```
PREFIX ex: <http://my/example#>
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>

INSERT DATA { ex:test ex:byte "123"^^xsd:byte }.
```

> **Note**
>
> As you can see, an item mapped from Neptune to OpenSearch may end up with a different datatype in OpenSearch than it has in Neptune. However, there is an explicit text field in OpenSearch, "datatype", that records the datatype that the item has in Neptune.

Validation of data mappings

Data is replicated to OpenSearch from Neptune using this process:

- If a mapping for the field in question is already present in OpenSearch:
  - If the data can be safely converted to the existing mapping using data validation rules, then store the field in OpenSearch.
  - If not, drop the corresponding stream update record.
- If there is no existing mapping for the field in question, find an OpenSearch datatype corresponding to the field's datatype in Neptune.
  - If the field data can be safely converted to the OpenSearch datatype using data validation rules, then store the new mapping and field data in OpenSearch.
  - If not, drop the corresponding stream update record.

Values are validated against equivalent OpenSearch types or existing OpenSearch mappings rather than the Neptune types. For example, validation for the value `"123"` in `"123"^^xsd:int` is done against the `long` type rather than the `int` type.

Although Neptune attempts to replicate all data to OpenSearch, there are cases where datatypes in OpenSearch are totally different from the ones in Neptune, and in such cases records are skipped rather than being indexed in OpenSearch.

For example, in Neptune one property can have multiple values of different types, whereas in OpenSearch a field must have the same type across the index.

By enabling debug logs, you can view what records have been dropped during export from Neptune to OpenSearch. An example of a debug log entry is:

```
Dropping Record : Data type not a valid Gremlin type
<Record>
```

Datatypes are validated as follows:

- **`text`** – All values in Neptune can safely be mapped to text in OpenSearch.
- **`long`** – The following rules for Neptune datatypes apply when the OpenSearch mapping type is long (in the examples below, it is assumed that `"testLong"` has a `long` mapping type):
  - `boolean` – Invalid, cannot be converted, and the corresponding stream update record is dropped.

    Invalid Gremlin examples are:

```
"testLong" : true.
"testLong" : false.
```

Invalid SPARQL examples are:

```
":testLong" : "true"^^xsd:boolean
":testLong" : "false"^^xsd:boolean
```

- `datetime` – Invalid, cannot be converted, and the corresponding stream update record is dropped.

  An invalid Gremlin example is:

  ```
  ":testLong" :  datetime('2018-11-04T00:00:00').
  ```

  An invalid SPARQL example is:

  ```
  ":testLong" : "2016-01-01"^^xsd:date
  ```

- `float`, `double`, or `decimal` – If the value in Neptune is an integer that can fit in 64 bits, it is valid and is stored in OpenSearch as a long, but if it has a fractional part, or is a `NaN` or an `INF`, or is larger than 9,223,372,036,854,775,807 or smaller than -9,223,372,036,854,775,808, then it is not valid and the corresponding stream update record is dropped.

  Valid Gremlin examples are:

  ```
  "testLong" :  145.0.
  ":testLong" :  123
  ":testLong" :  -9223372036854775807
  ```

  Valid SPARQL examples are:

  ```
  ":testLong" : "145.0"^^xsd:float
  ":testLong" :  145.0
  ":testLong" : "145.0"^^xsd:double
  ":testLong" : "145.0"^^xsd:decimal
  ":testLong" : "-9223372036854775807"
  ```

Invalid Gremlin examples are:

```
"testLong" :  123.45
":testLong" :  9223372036854775900
```

Invalid SPARQL examples are:

```
":testLong" :  123.45
":testLong" :  9223372036854775900
":testLong" : "123.45"^^xsd:float
":testLong" : "123.45"^^xsd:double
":testLong" : "123.45"^^xsd:decimal
```

- `string` – If the value in Neptune is a string representation of an integer that can be contained in a 64-bit integer, then it it is valid and is converted to a `long` in OpenSearch. Any other string value is invalid for an Elasticseearch `long` mapping, and the corresponding stream update record is dropped.

  Valid Gremlin examples are:

```
"testLong" :  "123".
":testLong" :  "145.0"
":testLong" :  "-9223372036854775807"
```

  Valid SPARQL examples are:

```
":testLong" : "145.0"^^xsd:string
":testLong" : "-9223372036854775807"^^xsd:string
```

  Invalid Gremlin examples are:

```
"testLong" :  "123.45"
":testLong" :  "9223372036854775900"
":testLong" :  "abc"
```

  Invalid SPARQL examples are:

```
":testLong" : "123.45"^^xsd:string
":testLong" : "abc"
```

```
  ":testLong" : "9223372036854775900"^^xsd:string
```

- **`double`** – If the OpenSearch mapping type is `double`, the following rules apply (here, the "testDouble" field is assumed to have a `double` mapping in OpenSearch):
  - `boolean` – Invalid, cannot be converted, and the corresponding stream update record is dropped.

    Invalid Gremlin examples are:

    ```
    "testDouble" : true.
    "testDouble" : false.
    ```

    Invalid SPARQL examples are:

    ```
    ":testDouble" : "true"^^xsd:boolean
    ":testDouble" : "false"^^xsd:boolean
    ```

  - `datetime` – Invalid, cannot be converted, and the corresponding stream update record is dropped.

    An invalid Gremlin example is:

    ```
    ":testDouble" :  datetime('2018-11-04T00:00:00').
    ```

    An invalid SPARQL example is:

    ```
    ":testDouble" : "2016-01-01"^^xsd:date
    ```

  - Floating-point `NaN` or `INF` – If the value in SPARQL is a floating-point `NaN` or `INF`, then it is not valid and the corresponding stream update record is dropped.

    Invalid SPARQL examples are:

    ```
"  :testDouble" : "NaN"^^xsd:float
    ":testDouble" : "NaN"^^double
    ":testDouble" : "INF"^^double
    ":testDouble" : "-INF"^^double
    ```

  - number or numeric string – If the value in Neptune is any other number or numeric string representation of a numnber that can safely be expressed as a `double`, then it is valid and

is converted to a `double` in OpenSearch. Any other string value is invalid for an OpenSearch `double` mapping, and the corresponding stream update record is dropped.

Valid Gremlin examples are:

```
"testDouble" :  123
":testDouble" :  "123"
":testDouble" :  145.67
":testDouble" :  "145.67"
```

Valid SPARQL examples are:

```
":testDouble" :  123.45
":testDouble" :  145.0
":testDouble" : "123.45"^^xsd:float
":testDouble" : "123.45"^^xsd:double
":testDouble" : "123.45"^^xsd:decimal
":testDouble" : "123.45"^^xsd:string
```

An invalid Gremlin example is:

```
":testDouble" :  "abc"
```

An Invalid SPARQL examples is:

```
":testDouble" : "abc"
```

- **`date`** – If the OpenSearch mapping type is `date`, Neptune `date` and `dateTime` value are valid, as is any string value that can be parsed successfully to a `dateTime` format.

  Valid examples in either Gremlin or SPARQL are:

```
Date(2016-01-01)
"2016-01-01" "
2003-09-25T10:49:41"
"2003-09-25T10:49"
"2003-09-25T10"
"20030925T104941-0300"
"20030925T104941"
"2003-Sep-25" "
```

```
  Sep-25-2003"
  "2003.Sep.25"
  "2003/09/25"
  "2003 Sep 25" "
  Wed, July 10, '96"
  "Tuesday, April 12, 1952 AD 3:30:42pm PST"
  "123"
  "-123"
  "0"
  "-0"
  "123.00"
  "-123.00"
```

Invalid examples are:

```
  123.45
  True
  "abc"
```

Sample non-string OpenSearch queries in Neptune

Neptune does not currently support OpenSearch range queries directly. However, you can achieve the same effect using Lucene syntax and query-type="query_string", as you can see in the following sample queries.

Get all vertices with age greater than 30 and name starting with "Si"

In Gremlin:

```
g.withSideEffect('Neptune#fts.endpoint', 'http://your-es-endpoint')
 .withSideEffect("Neptune#fts.queryType", "query_string")
 .V().has('*', 'Neptune#fts predicates.age.value:>30 && predicates.name.value:Si*');
```

In SPARQL:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://localhost:9200' .
```

```
    neptune-fts:config neptune-fts:queryType 'query_string' .
    neptune-fts:config neptune-fts:query "predicates.\\*foaf\\*age.value:>30 AND
 predicates.\\*foaf\\*name.value:Si*" .
    neptune-fts:config neptune-fts:field '*' .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

Here, "*foaf*age is used instead of the full URI for brevity. This regular expression will retrieve all fields have both foaf and age in the URI.

Get all nodes with age between 10 and 50 and a name with a fuzzy match with "Ronka"

In Gremlin:

```
g.withSideEffect('Neptune#fts.endpoint', 'http://your-es-endpoint')
 .withSideEffect("Neptune#fts.queryType", "query_string")
 .V().has('*', 'Neptune#fts predicates.age.value:[10 TO 50] AND
 predicates.name.value:Ronka~');
```

In SPARQL:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://localhost:9200' .
    neptune-fts:config neptune-fts:queryType 'query_string' .
    neptune-fts:config neptune-fts:query "predicates.\\*foaf\\*age.value:[10 TO 50] AND
 predicates.\\*foaf\\*name.value:Ronka~" .
    neptune-fts:config neptune-fts:field '*' .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

Get all nodes with a timestamp that falls within the last 25 days

In Gremlin:

```
g.withSideEffect('Neptune#fts.endpoint', 'http://your-es-endpoint')
 .withSideEffect("Neptune#fts.queryType", "query_string")
```

```
 .V().has('*', 'Neptune#fts predicates.timestamp.value:>now-25d');
```

In SPARQL:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://localhost:9200' .
    neptune-fts:config neptune-fts:queryType 'query_string' .
    neptune-fts:config neptune-fts:query "predicates.\\*foaf\
\*timestamp.value:>now-25d~" .
    neptune-fts:config neptune-fts:field '*' .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

Get all nodes with a timestamp that falls within a given year and month

In Gremlin, using date math expressions in Lucene syntax, for December 2020:

```
g.withSideEffect('Neptune#fts.endpoint', 'http://your-es-endpoint')
 .withSideEffect("Neptune#fts.queryType", "query_string")
 .V().has('*', 'Neptune#fts predicates.timestamp.value:>2020-12');
```

A Gremlin alternative:

```
g.withSideEffect('Neptune#fts.endpoint', 'http://your-es-endpoint')
 .withSideEffect("Neptune#fts.queryType", "query_string")
 .V().has('*', 'Neptune#fts predicates.timestamp.value:[2020-12 TO 2021-01]');
```

Full-text-search query execution in Amazon Neptune

In a query that includes full-text-search, Neptune tries to put the full-text-search calls first, before other parts of the query. This reduces the number of calls to OpenSearch and in most cases significantly improves performance. However, this is by no means a hard-and-fast rule. There are situations, for example, where a `PatternNode` or `UnionNode` may precede a full-text search call.

Consider the following Gremlin query to a database that contains 100,000 instances of `Person`:

```
g.withSideEffect('Neptune#fts.endpoint', 'your-es-endpoint-URL')
 .hasLabel('Person')
 .has('name', 'Neptune#fts marcello~');
```

If this query were executed in the order in which the steps appear then 100,000 solutions would flow into OpenSearch, causing hundreds of OpenSearch calls. In fact, Neptune calls OpenSearch first and then joins results with the Neptune results. In most cases, this is much faster than executing the query in the original order.

You can prevent this re-ordering of query-step execution using the noReordering query hint:

```
g.withSideEffect('Neptune#fts.endpoint', 'your-es-endpoint-URL')
 .withSideEffect('Neptune#noReordering', true)
 .hasLabel('Person')
 .has('name', 'Neptune#fts marcello~');
```

In this second case, the `.hasLabel` step is executed first and the `.has('name', 'Neptune#fts marcello~')` step second.

For another example, consider a SPARQL query against the same kind of data:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT ?person WHERE {
  ?person rdf:type foaf:Person .
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint.com' .
    neptune-fts:config neptune-fts:field foaf:name .
    neptune-fts:config neptune-fts:query 'mike' .
    neptune-fts:config neptune-fts:return ?person .
  }
}
```

Here again, Neptune executes the `SERVICE` part of the query first, and then joins the results with the `Person` data. You can suppress this behavior using the joinOrder query hint:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>
SELECT ?person WHERE {
  hint:Query hint:joinOrder "Ordered" .
```

```
  ?person rdf:type foaf:Person .
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint.com' .
    neptune-fts:config neptune-fts:field foaf:name .
    neptune-fts:config neptune-fts:query 'mike' .
    neptune-fts:config neptune-fts:return ?person .
  }
}
```

Again, in the second query the parts are executed in the order they appear in the query.

> **Note**
>
> Querying an opensearch alias over an index, instead of directly querying an opensearch index can produce incorrect results. You should query the opensearch index directly and not the alias.

Sample SPARQL queries using full-text search in Neptune

The following are some sample SPARQL queries that use full-text search in Amazon Neptune.

SPARQL match query example

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint.com' .
    neptune-fts:config neptune-fts:queryType 'match' .
    neptune-fts:config neptune-fts:field foaf:name .
    neptune-fts:config neptune-fts:query 'michael' .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

SPARQL prefix query example

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
```

```
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint.com' .
    neptune-fts:config neptune-fts:queryType 'prefix' .
    neptune-fts:config neptune-fts:field foaf:name .
    neptune-fts:config neptune-fts:query 'mich' .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

SPARQL fuzzy query example

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint.com' .
    neptune-fts:config neptune-fts:queryType 'fuzzy' .
    neptune-fts:config neptune-fts:field foaf:name .
    neptune-fts:config neptune-fts:query 'mikael' .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

SPARQL term query example

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint.com' .
    neptune-fts:config neptune-fts:queryType 'term' .
    neptune-fts:config neptune-fts:field foaf:name .
    neptune-fts:config neptune-fts:query 'Dr. Kunal' .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

SPARQL query_string query example

This query specifies multiple fields.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint.com' .
    neptune-fts:config neptune-fts:queryType 'query_string' .
    neptune-fts:config neptune-fts:query 'mikael~ OR rondelli' .
    neptune-fts:config neptune-fts:field foaf:name .
    neptune-fts:config neptune-fts:field foaf:surname .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

SPARQL simple_query_string query example

The following query specifies fields using the wildcard ('*') character.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint.com' .
    neptune-fts:config neptune-fts:queryType 'simple_query_string' .
    neptune-fts:config neptune-fts:query 'mikael~ | rondelli' .
    neptune-fts:config neptune-fts:field '*' .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

SPARQL sort by string field query example

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint' .
    neptune-fts:config neptune-fts:queryType 'query_string' .
    neptune-fts:config neptune-fts:query 'mikael~ | rondelli' .
    neptune-fts:config neptune-fts:field foaf:name .
    neptune-fts:config neptune-fts:sortOrder 'asc' .
    neptune-fts:config neptune-fts:sortBy foaf:name .
    neptune-fts:config neptune-fts:return ?res .
```

```
  }
}
```

SPARQL sort by non-string field query example

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint' .
    neptune-fts:config neptune-fts:queryType 'query_string' .
    neptune-fts:config neptune-fts:query 'mikael~ | rondelli' .
    neptune-fts:config neptune-fts:field foaf:name.value .
    neptune-fts:config neptune-fts:sortOrder 'asc' .
    neptune-fts:config neptune-fts:sortBy dc:date.value .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

SPARQL sort by ID query example

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint' .
    neptune-fts:config neptune-fts:queryType 'query_string' .
    neptune-fts:config neptune-fts:query 'mikael~ | rondelli' .
    neptune-fts:config neptune-fts:field foaf:name .
    neptune-fts:config neptune-fts:sortOrder 'asc' .
    neptune-fts:config neptune-fts:sortBy 'Neptune#fts.entity_id' .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

SPARQL sort by label query example

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
```

```
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint' .
    neptune-fts:config neptune-fts:queryType 'query_string' .
    neptune-fts:config neptune-fts:query 'mikael~ | rondelli' .
    neptune-fts:config neptune-fts:field foaf:name .
    neptune-fts:config neptune-fts:sortOrder 'asc' .
    neptune-fts:config neptune-fts:sortBy 'Neptune#fts.entity_type' .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

SPARQL sort by doc_type query example

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint' .
    neptune-fts:config neptune-fts:queryType 'query_string' .
    neptune-fts:config neptune-fts:query 'mikael~ | rondelli' .
    neptune-fts:config neptune-fts:field foaf:name .
    neptune-fts:config neptune-fts:sortOrder 'asc' .
    neptune-fts:config neptune-fts:sortBy 'Neptune#fts.document_type' .
    neptune-fts:config neptune-fts:return ?res .
  }
}
```

Example of using Lucene syntax in SPARQL

Lucene syntax is only supported for `query_string` queries in OpenSearch.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX neptune-fts: <http://aws.amazon.com/neptune/vocab/v01/services/fts#>
SELECT * WHERE {
  SERVICE neptune-fts:search {
    neptune-fts:config neptune-fts:endpoint 'http://your-es-endpoint' .
    neptune-fts:config neptune-fts:queryType 'query_string' .
    neptune-fts:config neptune-fts:queryType 'query_string' .
    neptune-fts:config neptune-fts:query 'predicates.\\foaf\\name.value:micheal AND
 predicates.\\foaf\\surname.value:sh' .
    neptune-fts:config neptune-fts:field ‘’ .
    neptune-fts:config neptune-fts:return ?res .
  }
```

```
}
```

Using Neptune full-text search in Gremlin queries

`NeptuneSearchStep` enables full-text search queries for the part of a Gremlin traversal that is not converted into Neptune steps. For example, consider a query like the following.

```
g.withSideEffect("Neptune#fts.endpoint", "your-es-endpoint-URL")
  .V()
      .tail(100)
      .has("name", "Neptune#fts mark*")            <== # Limit the search on name
```

This query is converted into the following optimized traversal in Neptune.

```
Neptune steps:
[
    NeptuneGraphQueryStep(Vertex) {
        JoinGroupNode {
            PatternNode[(?1, <~label>, ?2, <~>) . project distinct ?1 .],
 {estimatedCardinality=INFINITY}
        }, annotations={path=[Vertex(?1):GraphStep], maxVarId=4}
    },
    NeptuneTraverserConverterStep
]
+ not converted into Neptune steps: [NeptuneTailGlobalStep(100),
 NeptuneTinkerpopTraverserConverterStep, NeptuneSearchStep {
    JoinGroupNode {
        SearchNode[(idVar=?3, query=mark*, field=name) . project ask .],
 {endpoint=your-OpenSearch-endpoint-URL}
    }
    JoinGroupNode {
        SearchNode[(idVar=?3, query=mark*, field=name) . project ask .],
 {endpoint=your-OpenSearch-endpoint-URL}
    }
}]
```

The following examples are of Gremlin queries against air-routes data:

Gremlin basic case-insensitive `match` query

```
g.withSideEffect("Neptune#fts.endpoint",
```

```
                  "your-OpenSearch-endpoint-URL")
  .withSideEffect('Neptune#fts.queryType', 'match')
  .V().has("city","Neptune#fts dallas")

==>v[186]
==>v[8]
```

Gremlin `match` query

```
g.withSideEffect("Neptune#fts.endpoint",
                  "your-OpenSearch-endpoint-URL")
  .withSideEffect('Neptune#fts.queryType', 'match')
  .V().has("city","Neptune#fts southampton")
     .local(values('code','city').fold())
     .limit(5)

==>[SOU, Southampton]
```

Gremlin `fuzzy` query

```
g.withSideEffect("Neptune#fts.endpoint",
                  "your-OpenSearch-endpoint-URL")
  .V().has("city","Neptune#fts allas~").values('city').limit(5)

==>Dallas
==>Dallas
==>Walla Walla
==>Velas
==>Altai
```

Gremlin `query_string` fuzzy query

```
g.withSideEffect("Neptune#fts.endpoint",
                  "your-OpenSearch-endpoint-URL")
  .withSideEffect('Neptune#fts.queryType', 'query_string')
  .V().has("city","Neptune#fts allas~").values('city').limit(5)

==>Dallas
==>Dallas
```

Gremlin `query_string` regular expression query

```
g.withSideEffect("Neptune#fts.endpoint",
                 "your-OpenSearch-endpoint-URL")
  .withSideEffect('Neptune#fts.queryType', 'query_string')
  .V().has("city","Neptune#fts /[dp]allas/").values('city').limit(5)

==>Dallas
==>Dallas
```

Gremlin hybrid query

This query uses a Neptune internal index and the OpenSearch index in the same query.

```
g.withSideEffect("Neptune#fts.endpoint",
                 "your-OpenSearch-endpoint-URL")
  .V().has("region","GB-ENG")
      .has('city','Neptune#fts L*')
      .values('city')
      .dedup()
      .limit(10)

==>London
==>Leeds
==>Liverpool
==>Land's End
```

Simple Gremlin full-text search example

```
g.withSideEffect("Neptune#fts.endpoint",
                 "your-OpenSearch-endpoint-URL")
  .V().has('desc','Neptune#fts regional municipal')
      .local(values('code','desc').fold())
      .limit(100)

==>[HYA, Barnstable Municipal Boardman Polando Field]
==>[SPS, Sheppard Air Force Base-Wichita Falls Municipal Airport]
==>[ABR, Aberdeen Regional Airport]
==>[SLK, Adirondack Regional Airport]
==>[BFD, Bradford Regional Airport]
==>[EAR, Kearney Regional Airport]
==>[ROT, Rotorua Regional Airport]
```

```
==>[YHD, Dryden Regional Airport]
==>[TEX, Telluride Regional Airport]
==>[WOL, Illawarra Regional Airport]
==>[TUP, Tupelo Regional Airport]
==>[COU, Columbia Regional Airport]
==>[MHK, Manhattan Regional Airport]
==>[BJI, Bemidji Regional Airport]
==>[HAS, Hail Regional Airport]
==>[ALO, Waterloo Regional Airport]
==>[SHV, Shreveport Regional Airport]
==>[ABI, Abilene Regional Airport]
==>[GIZ, Jizan Regional Airport]
==>[USA, Concord Regional Airport]
==>[JMS, Jamestown Regional Airport]
==>[COS, City of Colorado Springs Municipal Airport]
==>[PKB, Mid Ohio Valley Regional Airport]
```

Gremlin query using `query_string` With '+' and '-' Operators

Although the `query_string` query type is much less forgiving than the default `simple_query_string` type, it does allow for more precise queries. The first query below uses `query_string`, while the second use the default `simple_query_string`:

```
g.withSideEffect("Neptune#fts.endpoint",
                 "your-OpenSearch-endpoint-URL")
  .withSideEffect('Neptune#fts.queryType', 'query_string')
 . V().has('desc','Neptune#fts +London -(Stansted|Gatwick)')
      .local(values('code','desc').fold())
      .limit(10)

==>[LHR, London Heathrow]
==>[YXU, London Airport]
==>[LTN, London Luton Airport]
==>[SEN, London Southend Airport]
==>[LCY, London City Airport]
```

Notice how `simple_query_string` in the examples below quietly ignores the '+' and '-' operators:

```
g.withSideEffect("Neptune#fts.endpoint",
                 "your-OpenSearch-endpoint-URL")
  .V().has('desc','Neptune#fts +London -(Stansted|Gatwick)')
```

```
      .local(values('code','desc').fold())
      .limit(10)

==>[LHR, London Heathrow]
==>[YXU, London Airport]
==>[LGW, London Gatwick]
==>[STN, London Stansted Airport]
==>[LTN, London Luton Airport]
==>[SEN, London Southend Airport]
==>[LCY, London City Airport]
==>[SKG, Thessaloniki Macedonia International Airport]
==>[ADB, Adnan Menderes International Airport]
==>[BTV, Burlington International Airport]
```

```
g.withSideEffect("Neptune#fts.endpoint",
                 "your-OpenSearch-endpoint-URL")
  .withSideEffect('Neptune#fts.queryType', 'query_string')
  .V().has('desc','Neptune#fts +(regional|municipal) -(international|bradford)')
      .local(values('code','desc').fold())
      .limit(10)

==>[CZH, Corozal Municipal Airport]
==>[MMU, Morristown Municipal Airport]
==>[YBR, Brandon Municipal Airport]
==>[RDD, Redding Municipal Airport]
==>[VIS, Visalia Municipal Airport]
==>[AIA, Alliance Municipal Airport]
==>[CDR, Chadron Municipal Airport]
==>[CVN, Clovis Municipal Airport]
==>[SDY, Sidney Richland Municipal Airport]
==>[SGU, St George Municipal Airport]
```

Gremlin `query_string` query with AND and OR operators

```
g.withSideEffect("Neptune#fts.endpoint",
                 "your-OpenSearch-endpoint-URL")
  .withSideEffect('Neptune#fts.queryType', 'query_string')
  .V().has('desc','Neptune#fts (St AND George) OR (St AND Augustin)')
      .local(values('code','desc').fold())
      .limit(10)

==>[YIF, St Augustin Airport]
```

```
==>[STG, St George Airport]
==>[SGO, St George Airport]
==>[SGU, St George Municipal Airport]
```

Gremlin `term` query

```
g.withSideEffect("Neptune#fts.endpoint",
                 "your-OpenSearch-endpoint-URL")
  .withSideEffect('Neptune#fts.queryType', 'term')
  .V().has("SKU","Neptune#fts ABC123DEF9")
      .local(values('code','city').fold())
      .limit(5)

==>[AUS, Austin]
```

Gremlin `prefix` query

```
g.withSideEffect("Neptune#fts.endpoint",
                 "your-OpenSearch-endpoint-URL")
  .withSideEffect('Neptune#fts.queryType', 'prefix')
  .V().has("icao","Neptune#fts ka")
      .local(values('code','icao','city').fold())
      .limit(5)

==>[AZO, KAZO, Kalamazoo]
==>[APN, KAPN, Alpena]
==>[ACK, KACK, Nantucket]
==>[ALO, KALO, Waterloo]
==>[ABI, KABI, Abilene]
```

Using Lucene syntax in Neptune Gremlin

In Neptune Gremlin, you can also write very powerful queries using the Lucene query syntax. Note that Lucene syntax is only supported for `query_string` queries in OpenSearch.

Assume the following data:

```
g.addV("person")
        .property(T.id, "p1")
        .property("name", "simone")
        .property("surname", "rondelli")
```

```
g.addV("person")
        .property(T.id, "p2")
        .property("name", "simone")
        .property("surname", "sengupta")

g.addV("developer")
        .property(T.id, "p3")
        .property("name", "simone")
        .property("surname", "rondelli")
```

Using Lucene syntax, which is invoked when the `queryType` is `query_string`, you can search this data by name and surname as follows:

```
g.withSideEffect("Neptune#fts.endpoint", "es_endpoint")
    .withSideEffect("Neptune#fts.queryType", "query_string")
    .V()
    .has("*", "Neptune#fts predicates.name.value:simone AND
 predicates.surname.value:rondelli")

==> v[p1], v[p3]
```

Note that in the `has()` step above, the field is replaced by `"*"`). Actually, any value placed there is overridden by the fields that you access within the query. You access the name field using `predicates.name.value,` because that is how the data model is structured.

You can search by name, surname and label, as follows:

```
g.withSideEffect("Neptune#fts.endpoint", getEsEndpoint())
    .withSideEffect("Neptune#fts.queryType", "query_string")
    .V()
    .has("*", "Neptune#fts predicates.name.value:simone AND
 predicates.surname.value:rondelli AND entity_type:person")

==> v[p1]
```

The label is accessed using `entity_type`, again because that is how the data model is structured.

You can also include nesting conditions:

```
g.withSideEffect("Neptune#fts.endpoint", getEsEndpoint())
    .withSideEffect("Neptune#fts.queryType", "query_string")
```

```
    .V()
    .has("*", "Neptune#fts (predicates.name.value:simone AND
 predicates.surname.value:rondelli AND entity_type:person) OR
 predicates.surname.value:sengupta")

==> v[p1], v[p2]
```

Inserting a modern TinkerPop graph

```
g.addV('person').property(T.id, '1').property('name', 'marko').property('age', 29)
 .addV('personr').property(T.id, '2').property('name', 'vadas').property('age', 27)
 .addV('software').property(T.id, '3').property('name', 'lop').property('lang', 'java')
 .addV('person').property(T.id, '4').property('name', 'josh').property('age', 32)
 .addV('software').property(T.id, '5').property('name', 'ripple').property('lang',
 'java')
 .addV('person').property(T.id, '6').property('name', 'peter').property('age', 35)

g.V('1').as('a').V('2').as('b').addE('knows').from('a').to('b').property('weight',
 0.5f).property(T.id, '7')
 .V('1').as('a').V('3').as('b').addE('created').from('a').to('b').property('weight',
 0.4f).property(T.id, '9')
 .V('4').as('a').V('3').as('b').addE('created').from('a').to('b').property('weight',
 0.4f).property(T.id, '11')
 .V('4').as('a').V('5').as('b').addE('created').from('a').to('b').property('weight',
 1.0f).property(T.id, '10')
 .V('6').as('a').V('3').as('b').addE('created').from('a').to('b').property('weight',
 0.2f).property(T.id, '12')
 .V('1').as('a').V('4').as('b').addE('knows').from('a').to('b').property('weight',
 1.0f).property(T.id, '8')
```

Sort by string field value example

```
g.withSideEffect("Neptune#fts.endpoint", "your-OpenSearch-endpoint-URL")
 .withSideEffect('Neptune#fts.queryType', 'query_string')
 .withSideEffect('Neptune#fts.sortOrder', 'asc')
 .withSideEffect('Neptune#fts.sortBy', 'name')
 .V().has('name', 'Neptune#fts marko OR vadas OR ripple')
```

Sort by non-string field value example

```
g.withSideEffect("Neptune#fts.endpoint", "your-OpenSearch-endpoint-URL")
```

```
 .withSideEffect('Neptune#fts.queryType', 'query_string')
 .withSideEffect('Neptune#fts.sortOrder', 'asc')
 .withSideEffect('Neptune#fts.sortBy', 'age.value')
 .V().has('name', 'Neptune#fts marko OR vadas OR ripple')
```

Sort by ID field value example

```
g.withSideEffect("Neptune#fts.endpoint", "your-OpenSearch-endpoint-URL")
.withSideEffect('Neptune#fts.queryType', 'query_string')
.withSideEffect('Neptune#fts.sortOrder', 'asc')
.withSideEffect('Neptune#fts.sortBy', 'Neptune#fts.entity_id')
.V().has('name', 'Neptune#fts marko OR vadas OR ripple')
```

Sort by label field value example

```
g.withSideEffect("Neptune#fts.endpoint", "your-OpenSearch-endpoint-URL")
 .withSideEffect('Neptune#fts.queryType', 'query_string')
 .withSideEffect('Neptune#fts.sortOrder', 'asc')
 .withSideEffect('Neptune#fts.sortBy', 'Neptune#fts.entity_type')
 .V().has('name', 'Neptune#fts marko OR vadas OR ripple')
```

Sort by `document_type` field value example

```
g.withSideEffect("Neptune#fts.endpoint", "your-OpenSearch-endpoint-URL")
 .withSideEffect('Neptune#fts.queryType', 'query_string')
 .withSideEffect('Neptune#fts.sortOrder', 'asc')
 .withSideEffect('Neptune#fts.sortBy', 'Neptune#fts.document_type')
 .V().has('name', 'Neptune#fts marko OR vadas OR ripple')
```

Troubleshooting Neptune full-text search

> **Note**
>
> If you have enabled fine-grained access control on your OpenSearch cluster, you need to enable IAM authentication in your Neptune database as well.

To diagnose issues with replication from Neptune to OpenSearch, consult the CloudWatch Logs for your poller Lambda function. These logs provide details about the number of records read from the stream and the number of records replicated successfully to OpenSearch.

You can also change the LOGGING level for your Lambda function by changing the `LoggingLevel`environment variable.

> **Note**
>
> With `LoggingLevel` set to DEBUG, you can view additional details, such as dropped stream records and the reason why each was dropped, while replicating data by StreamPoller from Neptune to OpenSearch. This can be useful if you find you are missing records.

The Neptune streams consumer application publishes two metrics on CloudWatch that can also help you diagnose problems:

- `StreamRecordsProcessed` – The number of records processed by the application per unit of time. Helpful in tracking the application run rate.
- `StreamLagTime` – The time difference in milliseconds between the current time and the commit time of a stream record being processed. This metric shows how much the consumer application is lagging behind.

In addition, all the metrics related to the replication process are exposed in a dashboard in CloudWatch under the same name same as the `ApplicationName` provided when you instantiated the application using the CloudWatch template.

You can also choose to create a CloudWatch alarm that is triggered whenever polling fails more than twice in a row. Do this by setting the `CreateCloudWatchAlarm` field to `true` when you instantiate the application. Then specify the email addresses that you want to be notified when the alarm is triggered.

Troubleshooting a process that fails while reading records from the stream

If a process fails while reading records from the stream, make sure that you have the following:

- The stream is enabled on your cluster.

- The Neptune stream endpoint is in the correct format:
  - For Gremlin or openCypher: `https://your cluster endpoint:your cluster port/propertygraph/stream` or its alias, `https://your cluster endpoint:your cluster port/pg/stream`
  - For SPARQL: `https://your cluster endpoint:your cluster port/sparql/stream`
- The DynamoDB endpoint is configured for your VPC.
- The monitoring endpoint is configured for your VPC subnets.

Troubleshooting a process that fails while writing data to OpenSearch

If a process fails while writing records to OpenSearch, make sure that you have the following:

- Your Elasticsearch version is 7.1 or higher, or Opensearch 2.3 and above.
- OpenSearch can be accessed from the poller Lambda function in your VPC.
- The security policy attached to OpenSearch allows inbound HTTP/HTTPS requests.

Fixing out-of-sync issues between Neptune and OpenSearch on an existing replication setup

You can use the steps below to get a Neptune database and OpenSearch domain back in sync with the latest data in case of out-of-sync issues between them resulting from an `ExpiredStreamException` or data corruption.

Note that this approach deletes all the data in the OpenSearch domain and re-syncs it from the current state of the Neptune database, so no data needs to be reloaded in the Neptune database.

1. Disable the replication process as described in Disabling (pausing) the stream poller process.
2. Delete the Neptune index on the OpenSearch domain using the following command:

```
curl -X DELETE "(your OpenSearch endpoint)/amazon_neptune"
```

3. Create a clone of the database (see Database Cloning in Neptune).
4. Get the latest `eventID` for the streams on the cloned database by executing a command of this kind against the Streams API endpoint (see Calling the Neptune Streams REST API for more information):

```
curl "https://(your neptune endpoint):(port)/(propertygraph or sparql)/stream?
iteratorType=LATEST"
```

Make a note of the values in the `commitNum` and `opNum` fields in the `lastEventId` object in the response.

5. Use the export-neptune-to-elasticsearch tool on github to perform a one-time synchronization from the cloned database to the OpenSearch domain.
6. Go to the DynamoDB table for the replication stack. The name of the table will be the **Application Name** you specified in the CloudFormation template (the default is `NeptuneStream`) with a `-LeaseTable` suffix. In other words, the default table name is `NeptuneStream-LeaseTable`.

   You can explore table rows by scanning because there should only be one row in the table. Make the following changes using the `commitNum` and `opNum` values you recorded above:

   - Change the value for the `checkpoint` field in the table to the value you noted for `commitNum`.
   - Change the value for `checkpointSubSequenceNumber` field in the table to the value you noted for `opNum`.
7. Re-enable the replication process as described in Re-enabling the stream poller process.
8. Delete the cloned database and the CloudFormation stack created for the `export-neptune-to-elasticsearch` tool.

Using AWS Lambda functions in Amazon Neptune

AWS Lambda functions have many uses in Amazon Neptune applications. Here we provide general guidance for using Lambda functions with any of the popular Gremlin drivers and language variants, and specific examples of Lambda functions written in Java, JavaScript, and Python.

> **Note**
>
> The best way to use Lambda functions with Neptune has changed with recent engine releases. Neptune used to leave idle connections open long after a Lambda execution context had been recycled, potentially leading to a resource leak on the server. To mitigate this, we used to recommend opening and closing a connection with each Lambda invocation. Starting with engine version 1.0.3.0, however, the idle connection timeout has been reduced so that connections no longer leak after an inactive Lambda execution context has been recycled, so we now recommend using a single connection for the duration of the execution context. This should include some error handling and back-off-and-retry boilerplate code to handle connections being closed unexpectedly.

Managing Gremlin WebSocket connections in AWS Lambda functions

If you use a Gremlin language variant to query Neptune, the driver connects to the database using a WebSocket connection. WebSockets are designed to support long-lived client-server connection scenarios. AWS Lambda, on the other hand, is designed to support relatively short-lived and stateless executions. This mismatch in design philosophy can lead to some unexpected issues when using Lambda to query Neptune.

An AWS Lambda function runs in an execution context which isolates the function from other functions. The execution context is created the first time the function is invoked and may be reused for subsequent invocations of the same function.

Any one execution context is never used to handle multiple concurrent invocations of the function, however. If your function is invoked simultaneously by multiple clients, Lambda spins up an additional execution context for each instance of the function. All these new execution contexts may in turn be reused for subsequent invocations of the function.

At some point, Lambda recycles execution contexts, particularly if they have been inactive for some time. AWS Lambda exposes the execution context lifecycle, including the `Init`, `Invoke` and `Shutdown` phases, through Lambda extensions. Using these extensions, you can write code that cleans up external resources such as database connections when the execution context is recycled.

A common best practice is to open the database connection outside the Lambda handler function so that it can be reused with each handler call. If the database connection drops at some point, you can reconnect from inside the handler. However, there is a danger of connection leaks with this approach. If an idle connection stays open long after an execution context is destroyed, intermittent or bursty Lambda invocation scenarios can gradually leak connections and exhaust database resources.

Neptune connection limits and connection timeouts have changed with newer engine releases. Previously, every instance supported up to 60,000 WebSocket connections. Now, the maximum number of concurrent WebSocket connections per Neptune instance varies with the instance type.

Also, starting with engine release 1.0.3.0, Neptune reduced the idle timeout for connections from one hour down to approximately 20 minutes. If a client doesn't close a connection, the connection is closed automatically after a 20- to 25-minute idle timeout. AWS Lambda doesn't document execution context lifetimes, but experiments show that the new Neptune connection timeout aligns well with inactive Lambda execution context timeouts. By the time an inactive execution context is recycled, there's a good chance its connection has already been closed by Neptune, or will be closed soon afterwards.

Recommendations for using AWS Lambda with Amazon Neptune Gremlin

We now recommend using a single connection and graph traversal source for the entire lifetime of a Lambda execution context, rather than one for each function invocation (every function invocation handles only one client request). Because concurrent client requests are handled by different function instances running in separate execution contexts, there's no need to maintain a pool of connections to handle concurrent requests inside a function instance. If the Gremlin driver you’re using has a connection pool, configure it to use just one connection.

To handle connection failures, use retry logic around each query. Even though the goal is to maintain a single connection for the lifetime of an execution context, unexpected network events can cause that connection to be terminated abruptly. Such connection failures manifest as different

errors depending on which driver you are using. You should code your Lambda function to handle these connection issues and attempt a reconnection if necessary.

Some Gremlin drivers automatically handle reconnections. The Java driver, for example, automatically attempts to reestablish connectivity to Neptune on behalf of your client code. With this driver, your function code only needs to back off and retry the query. The JavaScript and Python drivers, by contrast, do not implement any automatic reconnection logic, so with these drivers your function code must try to reconnect after backing off, and only retry the query once the connection has been re-established.

Code examples here do include reconnection logic rather than assume that the client is taking care of it.

Recommendations for using Gremlin write-requests in Lambda

If your Lambda function modifies graph data, consider adopting a back-off-and-retry strategy to handle the following exceptions. For detailed guidance on developing a practical retry strategy, see the section called "Exceptions and Retries".

- **`ConcurrentModificationException`** – The Neptune transaction semantics mean that write requests sometimes fail with a `ConcurrentModificationException`. In these situations, try an exponential back-off-based retry mechanism.
- **`ReadOnlyViolationException`** – Because the cluster topology can change at any moment as a result of planned or unplanned events, write responsibilities may migrate from one instance in the cluster to another. If your function code attempts to send a write request to an instance that is no longer the primary (writer) instance, the request fails with a `ReadOnlyViolationException`. When this happens, close the existing connection, reconnect to the cluster endpoint, and then retry the request.

Also, if you use a back-off-and-retry strategy to handle write request issues, consider implementing idempotent queries for create and update requests using the section called "Efficient upserts from 3.6.x forward".

Recommendations for using Gremlin read-requests in Lambda

If you have one or more read replicas in your cluster, it's a good idea to balance read requests across these replicas. One option is to use the reader endpoint. The reader endpoint balances

connections across replicas even if the cluster topology changes when you add or remove replicas, or promote a replica to become the new primary instance.

However, using the reader endpoint can result in an uneven use of cluster resources in some circumstances. The reader endpoint works by periodically changing the host that the DNS entry points to. If a client opens a lot of connections before the DNS entry changes, all the connection requests are sent to a single Neptune instance. This can be the case with a high-throughput Lambda scenario where a large number of concurrent requests to your Lambda function causes multiple execution contexts to be created, each with its own connection. If those connections are all created nearly simultaneously, the connections are likely to all point to the same replica in the cluster, and to stay pointing to that replica until the execution contexts are recycled.

One way you can distribute requests across instances is to configure your Lambda function to connect to an instance endpoint, chosen at random from a list of replica instance endpoints, rather than the reader endpoint. The downside of this approach is that it requires the Lambda code to handle changes in the cluster topology by monitoring the cluster and updating the endpoint list whenever the membership of the cluster changes.

If you are writing a Java Lambda function that needs to balance read requests across instances in your cluster, you can use the Gremlin client for Amazon Neptune, a Java Gremlin client that is aware of your cluster topology and which fairly distributes connections and requests across a set of instances in a Neptune cluster. This blog post includes a sample Java Lambda function that uses the Gremlin client for Amazon Neptune.

Factors that may slow down cold starts of Neptune Gremlin Lambda functions

The first time an AWS Lambda function is invoked is referred to as a cold start. There are several factors that can increase the latency of a cold start:

- **Be sure to assign enough memory to your Lambda function.** – Compilation during a cold start can be significantly slower for a Lambda function than it would be on EC2 because AWS Lambda allocates CPU cycles linearly in proportion to the memory that you assign to the function. With 1,769 MB of memory, a function receives the equivalent of one full vCPU (one vCPU-second of credits per second). The impact of not assigning enough memory to receive adequate CPU cycles is particularly pronounced for large Lambda functions written in Java.
- **Be aware that enabling IAM database authentication may slow down a cold start** – AWS Identity and Access Management (IAM) database authentication can also slow down cold starts,

particularly if the Lambda function has to generate a new signing key. This latency only affects the cold start and not subsequent requests, because once IAM DB auth has established the connection credentials, Neptune only periodically validates that they are still valid.

Using CloudFormation to Create a Lambda Function to Use in Neptune

You can use an CloudFormation template to create an AWS Lambda function that can access Neptune.

1. To launch the Lambda function stack on the CloudFormation console, choose one of the **Launch Stack** buttons in the following table.

Region	View	View in Designer	Launch
US East (N. Virginia)	View	View in Designer	Launch Stack
US East (Ohio)	View	View in Designer	Launch Stack
US West (N. California)	View	View in Designer	Launch Stack
US West (Oregon)	View	View in Designer	Launch Stack
Canada (Central)	View	View in Designer	Launch Stack
South America (São Paulo)	View	View in Designer	Launch Stack
Europe (Stockholm)	View	View in Designer	Launch Stack
Europe (Ireland)	View	View in Designer	Launch Stack

Region	View	View in Designer	Launch
Europe (London)	View	View in Designer	Launch Stack
Europe (Paris)	View	View in Designer	Launch Stack
Europe (Frankfurt)	View	View in Designer	Launch Stack
Middle East (Bahrain)	View	View in Designer	Launch Stack
Middle East (UAE)	View	View in Designer	Launch Stack
Israel (Tel Aviv)	View	View in Designer	Launch Stack
Africa (Cape Town)	View	View in Designer	Launch Stack
Asia Pacific (Hong Kong)	View	View in Designer	Launch Stack
Asia Pacific (Tokyo)	View	View in Designer	Launch Stack
Asia Pacific (Seoul)	View	View in Designer	Launch Stack
Asia Pacific (Singapore)	View	View in Designer	Launch Stack
Asia Pacific (Sydney)	View	View in Designer	Launch Stack
Asia Pacific (Mumbai)	View	View in Designer	Launch Stack
China (Beijing)	View	View in Designer	Launch Stack

Region	View	View in Designer	Launch
China (Ningxia)	View	View in Designer	Launch Stack
AWS GovCloud (US-West)	View	View in Designer	Launch Stack
AWS GovCloud (US-East)	View	View in Designer	Launch Stack

2. On the **Select Template** page, choose **Next**.
3. On the **Specify Details** page, set the following options:
   a. Choose the Lambda runtime, depending on what language you want to use in your Lambda function. These CloudFormation templates currently support the following languages:
      - **Python 3.9** (maps to `python39` in the Amazon S3 URL)
      - **NodeJS 18** (maps to `nodejs18x` in the Amazon S3 URL)
      - **Ruby 2.5** (maps to `ruby25` in the Amazon S3 URL)
   b. Provide the appropriate Neptune cluster endpoint and port number.
   c. Provide the appropriate Neptune security group.
   d. Provide the appropriate Neptune subnet parameters.
4. Choose **Next**.
5. On the **Options** page, choose **Next**.
6. On the **Review** page, select the first check box to acknowledge that CloudFormation will create IAM resources.

   Then choose **Create**.

If you need to make your own changes to the Lambda runtime, you can download a generic one from an Amazon S3 location in your Region:

```
https://s3.Amazon region.amazonaws.com/aws-neptune-customer-samples-Amazon region/
lambda/runtime-language/lambda_function.zip.
```

For example:

```
https://s3.us-west-2.amazonaws.com/aws-neptune-customer-samples-us-west-2/lambda/
python36/lambda_function.zip
```

AWS Lambda function examples for Amazon Neptune

The following example AWS Lambda functions, written in Java, JavaScript and Python, illustrate upserting a single vertex with a randomly generated ID using the `mergeV()` step (see the section called "Efficient upserts from 3.6.x forward").

Much of the code in each function is boilerplate code, responsible for managing connections and retrying connections and queries if an error occurs. The real application logic and the Gremlin query are implemented in `doQuery()` and `query()` methods respectively. If you use these examples as a basis for your own Lambda functions, you can concentrate on modifying `doQuery()` and `query()`.

The functions are configured to retry failed queries 5 times, waiting 1 second between retries. In a production application, consider use of exponential backoff with jitter rather than a fixed interval. For detailed guidance, see the section called "Exceptions and Retries".

The functions require values to be present in the following Lambda environment variables:

- **`NEPTUNE_ENDPOINT`** – Your Neptune DB cluster endpoint. For Python, this should be `neptuneEndpoint`.
- **`NEPTUNE_PORT`** – The Neptune port. For Python, this should be `neptunePort`.
- **`USE_IAM`** – (`true` or `false`) If your database has AWS Identity and Access Management (IAM) database authentication enabled, set the `USE_IAM` environment variable to `true`. This causes the Lambda function to Sigv4-sign connection requests to Neptune. For such IAM DB auth requests, ensure that the Lambda function's execution role has an appropriate IAM policy attached that allows the function to connect to your Neptune DB cluster (see Types of IAM policies).

Java Lambda function example for Amazon Neptune

Here are some things to keep in mind about Java AWS Lambda functions:

- The Java driver maintains its own connection pool, which you do not need, so configure your `Cluster` object with `minConnectionPoolSize(1)` and `maxConnectionPoolSize(1)`.
- The `Cluster` object can be slow to build because it creates one or more serializers (Gyro by default, plus another if you've configured it for additional output formats such as `binary`). These can take a while to instantiate.
- The connection pool is initialized with the first request. At this point, the driver sets up the `Netty` stack, allocates byte buffers, and creates a signing key if you are using IAM DB auth. All of which can add to the cold-start latency.
- The Java driver's connection pool monitors the availability of server hosts and automatically attempts to reconnect if a connection fails. It starts a background task to try to re-establish the connection. Use `reconnectInterval( )` to configure the interval between reconnection attempts. While the driver is attempting to reconnect, your Lambda function can simply retry the query.

  If the interval between retries is smaller than the interval between reconnect attempts, retries on a failed connection fail again because the host is considered unavailable. This does not apply to retries for a `ConcurrentModificationException`.
- Use Java 11 or later. TinkerPop 3.8.0 drops support for Java 8, so the Gremlin Java driver requires Java 11 as a minimum.
- This example uses Retry4j for retries.
- To use the `Sigv4` signing driver in your Java Lambda function, see the dependency requirements in Connecting to Amazon Neptune databases using IAM with Gremlin Java.

> **Warning**
>
> The `CallExecutor` from Retry4j may not be thread-safe. Consider having each thread use its own `CallExecutor` instance.

> **Note**
>
> The following example has been updated to include the use of requestInterceptor(). This was added in TinkerPop 3.6.6. Prior to TinkerPop version 3.6.6, the code example used handshakeInterceptor(), which was deprecated with that release.

```
package com.amazonaws.examples.social;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestStreamHandler;
import com.evanlennick.retry4j.CallExecutor;
import com.evanlennick.retry4j.CallExecutorBuilder;
import com.evanlennick.retry4j.Status;
import com.evanlennick.retry4j.config.RetryConfig;
import com.evanlennick.retry4j.config.RetryConfigBuilder;
import org.apache.tinkerpop.gremlin.driver.Cluster;
import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
import com.amazonaws.neptune.auth.NeptuneNettyHttpSigV4Signer;
import org.apache.tinkerpop.gremlin.driver.remote.DriverRemoteConnection;
import org.apache.tinkerpop.gremlin.driver.ser.Serializers;
import org.apache.tinkerpop.gremlin.process.traversal.AnonymousTraversalSource;
import org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversalSource;
import org.apache.tinkerpop.gremlin.process.traversal.Merge;
import org.apache.tinkerpop.gremlin.structure.T;

import java.io.*;
import java.time.temporal.ChronoUnit;
import java.util.HashMap;
import java.util.Map;
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.function.Function;

import static java.nio.charset.StandardCharsets.UTF_8;

public class MyHandler implements RequestStreamHandler {

  private final GraphTraversalSource g;
  private final CallExecutor<Object> executor;
  private final Random idGenerator = new Random();

  public MyHandler() {

    this.g = AnonymousTraversalSource
            .traversal()
            .withRemote(DriverRemoteConnection.using(createCluster()));


    this.executor = new CallExecutorBuilder<Object>()
```

```
            .config(createRetryConfig())
            .build();

  }

  @Override
  public void handleRequest(InputStream input,
                            OutputStream output,
                            Context context) throws IOException {

    doQuery(input, output);
  }

  private void doQuery(InputStream input, OutputStream output) throws IOException {
    try {

      Map<String, Object> args = new HashMap<>();
      args.put("id", idGenerator.nextInt());

      String result = query(args);

      try (Writer writer = new BufferedWriter(new OutputStreamWriter(output, UTF_8))) {
          writer.write(result);
      }

    } finally {
        input.close();
        output.close();
    }
  }

  private String query(Map<String, Object> args) {
    int id = (int) args.get("id");

    @SuppressWarnings("unchecked")
    Callable<Object> query = () -> g.mergeV(Map.of(T.id, id))
      .option(Merge.onCreate, Map.of(T.label, "Person"))
      .id().next();

    Status<Object> status = executor.execute(query);

    return status.getResult().toString();
  }
```

```
  private Cluster createCluster() {
    Cluster.Builder builder = Cluster.build()

 .addContactPoint(System.getenv("NEPTUNE_ENDPOINT"))

 .port(Integer.parseInt(System.getenv("NEPTUNE_PORT")))
                                     .enableSsl(true)
                                     .minConnectionPoolSize(1)
                                     .maxConnectionPoolSize(1)
                                     .serializer(Serializers.GRAPHBINARY_V1D0)
                                     .reconnectInterval(2000);

  if (Boolean.parseBoolean(getOptionalEnv("USE_IAM", "true"))) {
    // The following example uses requestInterceptor(), which was introduced
    // in TinkerPop 3.6.6. If you are using a TinkerPop version earlier than
    // 3.6.6 (but 3.5.5 or higher), use handshakeInterceptor() instead.
    builder.requestInterceptor( r ->
      {
        NeptuneNettyHttpSigV4Signer sigV4Signer = new
 NeptuneNettyHttpSigV4Signer(region, DefaultCredentialsProvider.create());
        sigV4Signer.signRequest(r);
        return r;
      }
    )

    return builder.create();
  }

  // NOTE: This example uses a fixed backoff for simplicity.
  // In a production application consider use of exponential backoff with jitter.
  private RetryConfig createRetryConfig() {
    return new RetryConfigBuilder().retryOnCustomExceptionLogic(retryLogic())
                                   .withDelayBetweenTries(1000, ChronoUnit.MILLIS)
                                   .withMaxNumberOfTries(5)
                                   .withFixedBackoff()
                                   .build();
  }

  private Function<Exception, Boolean> retryLogic() {
    return e -> {
      StringWriter stringWriter = new StringWriter();
      e.printStackTrace(new PrintWriter(stringWriter));
      String message = stringWriter.toString();
```

```
      // Check for connection issues
      if ( message.contains("Timed out while waiting for an available host") ||
           message.contains("Timed-out waiting for connection on Host") ||
           message.contains("Connection to server is no longer active") ||
           message.contains("Connection reset by peer") ||
           message.contains("SSLEngine closed already") ||
           message.contains("Pool is shutdown") ||
           message.contains("ExtendedClosedChannelException") ||
           message.contains("Broken pipe")) {
        return true;
      }

      // Concurrent writes can sometimes trigger a ConcurrentModificationException.
      // In these circumstances you may want to backoff and retry.
      if (message.contains("ConcurrentModificationException")) {
          return true;
      }

      // If the primary fails over to a new instance, existing connections to the old
 primary will
      // throw a ReadOnlyViolationException. You may want to back and retry.
      if (message.contains("ReadOnlyViolationException")) {
          return true;
      }

      return false;
    };
  }

  private String getOptionalEnv(String name, String defaultValue) {
    String value = System.getenv(name);
    if (value != null && value.length() > 0) {
      return value;
    } else {
      return defaultValue;
    }
  }
}
```

If you want to include reconnect logic in your function, see Java reconnect sample.

JavaScript Lambda function example for Amazon Neptune

Notes about this example

- The JavaScript driver doesn't maintain a connection pool. It always opens a single connection.
- The example function uses the Sigv4 signing utilities from gremlin-aws-sigv4 for signing requests to an IAM authentication-enabled database.
- It uses the retry( ) function from the open-source async utility module to handle backoff-and-retry attempts.
- Gremlin terminal steps return a JavaScript `promise` (see the TinkerPop documentation). For `next()`, this is a `{value, done}` tuple.
- Connection errors are raised inside the handler, and dealt with using some backoff-and-retry logic in line with the recommendations outlined here, with one exception. There is one kind of connection issue that the driver does not treat as an exception, and which cannot therefore be accommodated by this backoff-and-retry logic.

  The problem is that if a connection is closed after a driver sends a request but before the driver receives a response, the query appears to complete but returns a null value. As far as the lambda function client is concerned, the function appears to complete successfully, but with an empty response.

  The impact of this issue depends on how your application treats an empty response. Some applications may treat an empty response from a read request as an error, but others may mistakenly treat it as an empty result.

  Write requests that encounter this connection issue will also return an empty response. Does a successful invocation with an empty response signal success or failure? If the client invoking a write function simply treats the successful invocation of the function to mean the write to the database has been committed, rather than inspecting the body of the response, the system may appear to lose data.

  This issue results from how the driver treats events emitted by the underlying socket. When the underlying network socket is closed with an `ECONNRESET` error, the WebSocket used by the driver is closed and emits a `'ws close'` event. There's nothing in the driver, however, to handle that event in a way that could be used to raise an exception. As a result, the query simply disappears.

To work around this issue, the example lambda function here adds a `'ws close'` event handler that throws an exception to the driver when creating a remote connection. This exception is not, however, raised along the Gremlin query's request-response path, and can't therefore be used to trigger any backoff-and-retry logic within the lambda function itself. Instead, the exception thrown by the `'ws close'` event handler results in an unhandled exception that causes the lambda invocation to fail. This allows the client that invokes the function to handle the error and retry the lambda invocation if appropriate.

We recommend that you implement backoff-and-retry logic in the lambda function itself to protect your clients from intermittent connection issues. However, the workaround for the above issue requires the client to implement retry logic too, to handle failures that result from this particular connection issue.

Javascript code

```
const gremlin = require('gremlin');
const async = require('async');
const {getUrlAndHeaders} = require('gremlin-aws-sigv4/lib/utils');

const traversal = gremlin.process.AnonymousTraversalSource.traversal;
const DriverRemoteConnection = gremlin.driver.DriverRemoteConnection;
const t = gremlin.process.t;
const Merge = gremlin.process.merge;

let conn = null;
let g = null;

async function query(context) {

  const id = context.id;

  return g.mergeV({[t.id]: id})
    .option(Merge.onCreate, {[t.label]: 'User'})
    .id().next();
}

async function doQuery() {
  const id = Math.floor(Math.random() * 10000).toString();

  let result = await query({id: id});
```

```
  return result['value'];
}

exports.handler = async (event, context) => {

  const getConnectionDetails = () => {
    if (process.env['USE_IAM'] == 'true'){
       return getUrlAndHeaders(
         process.env['NEPTUNE_ENDPOINT'],
         process.env['NEPTUNE_PORT'],
         {},
         '/gremlin',
         'wss');
    } else {
      const database_url = 'wss://' + process.env['NEPTUNE_ENDPOINT'] + ':' +
 process.env['NEPTUNE_PORT'] + '/gremlin';
      return { url: database_url, headers: {}};
    }
  };


  const createRemoteConnection = () => {
    const { url, headers } = getConnectionDetails();

    const c = new DriverRemoteConnection(
      url,
      {
        headers: headers
      });

     c._client._connection.on('close', (code, message) => {
         console.info(`close - ${code} ${message}`);
         if (code == 1006){
           console.error('Connection closed prematurely');
           throw new Error('Connection closed prematurely');
         }
       });

     return c;
  };

  const createGraphTraversalSource = (conn) => {
    return traversal().withRemote(conn);
  };
```

```
if (conn == null){
  console.info("Initializing connection")
  conn = createRemoteConnection();
  g = createGraphTraversalSource(conn);
}

// NOTE: This example uses a fixed interval for simplicity.
// In a production application consider use of exponential backoff with jitter.
return async.retry(
  {
    times: 5,
    interval: 1000,
    errorFilter: function (err) {

      // Add filters here to determine whether error can be retried
      console.warn('Determining whether retriable error: ' + err.message);

      // Check for connection issues
      if (err.message.startsWith('WebSocket is not open')){
        console.warn('Reopening connection');
        conn.close();
        conn = createRemoteConnection();
        g = createGraphTraversalSource(conn);
        return true;
      }

      // Check for ConcurrentModificationException
      if (err.message.includes('ConcurrentModificationException')){
        console.warn('Retrying query because of ConcurrentModificationException');
        return true;
      }

      // Check for ReadOnlyViolationException
      if (err.message.includes('ReadOnlyViolationException')){
        console.warn('Retrying query because of ReadOnlyViolationException');
        return true;
      }

      return false;
    }

  },
  doQuery);
```

```
};
```

Python Lambda function example for Amazon Neptune

Here are some things to notice about the following Python AWS Lambda example function:

- It uses the backoff module.
- It sets `pool_size=1` to keep from creating an unnecessary connection pool.
- Requires the Python 3.10 or later Lambda runtime, because the `gremlinpython` library aligned with TinkerPop 3.8.0 requires Python 3.10 or later.

```
import os, sys, backoff, math
from random import randint
from gremlin_python import statics
from gremlin_python.driver.driver_remote_connection import DriverRemoteConnection
from gremlin_python.driver.protocol import GremlinServerError
from gremlin_python.driver import serializer
from gremlin_python.process.anonymous_traversal import traversal
from gremlin_python.process.graph_traversal import __
from gremlin_python.process.strategies import *
from gremlin_python.process.traversal import T
from aiohttp.client_exceptions import ClientConnectorError
from botocore.auth import SigV4Auth
from botocore.awsrequest import AWSRequest
from botocore.credentials import ReadOnlyCredentials
from types import SimpleNamespace

import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)


reconnectable_err_msgs = [
    'ReadOnlyViolationException',
    'Server disconnected',
    'Connection refused',
    'Connection was already closed',
    'Connection was closed by server',
    'Failed to connect to server: HTTP Error code 403 - Forbidden'
]
```

```
retriable_err_msgs = ['ConcurrentModificationException'] + reconnectable_err_msgs

network_errors = [OSError, ClientConnectorError]

retriable_errors = [GremlinServerError, RuntimeError, Exception] + network_errors

def prepare_iamdb_request(database_url):

    service = 'neptune-db'
    method = 'GET'

    access_key = os.environ['AWS_ACCESS_KEY_ID']
    secret_key = os.environ['AWS_SECRET_ACCESS_KEY']
    region = os.environ['AWS_REGION']
    session_token = os.environ['AWS_SESSION_TOKEN']


    creds = SimpleNamespace(
        access_key=access_key, secret_key=secret_key, token=session_token,
 region=region,
    )

    request = AWSRequest(method=method, url=database_url, data=None)
    SigV4Auth(creds, service, region).add_auth(request)

    return database_url, request.headers.items()

def is_retriable_error(e):

    is_retriable = False
    err_msg = str(e)

    if isinstance(e, tuple(network_errors)):
        is_retriable = True
    else:
        is_retriable = any(retriable_err_msg in err_msg for retriable_err_msg in
 retriable_err_msgs)

    logger.error('error: [{}] {}'.format(type(e), err_msg))
    logger.info('is_retriable: {}'.format(is_retriable))

    return is_retriable

def is_non_retriable_error(e):
```

```
    return not is_retriable_error(e)

def reset_connection_if_connection_issue(params):

    is_reconnectable = False

    e = sys.exc_info()[1]
    err_msg = str(e)

    if isinstance(e, tuple(network_errors)):
        is_reconnectable = True
    else:
        is_reconnectable = any(reconnectable_err_msg in err_msg for
 reconnectable_err_msg in reconnectable_err_msgs)

    logger.info('is_reconnectable: {}'.format(is_reconnectable))

    if is_reconnectable:
        global conn
        global g
        conn.close()
        conn = create_remote_connection()
        g = create_graph_traversal_source(conn)

@backoff.on_exception(backoff.constant,
                      tuple(retriable_errors),
                      max_tries=5,
                      jitter=None,
                      giveup=is_non_retriable_error,
                      on_backoff=reset_connection_if_connection_issue,
                      interval=1)
def query(**kwargs):

    id = kwargs['id']

    return (g.V().hasLabel('column').has('column_name',
 'amhstr_ag_type').in_('hascolumn').dedup().valueMap().limit(10).toList())

def doQuery(event):
    return query(id=str(randint(0, 10000)))

def lambda_handler(event, context):
    result = doQuery(event)
    logger.info('result - {}'.format(result))
```

```
    return result

def create_graph_traversal_source(conn):
    return traversal().withRemote(conn)

def create_remote_connection():
    logger.info('Creating remote connection')

    (database_url, headers) = connection_info()

    # Convert headers to a dictionary if it's not already
    headers_dict = dict(headers) if isinstance(headers, list) else headers

    print(headers)
    return DriverRemoteConnection(
        database_url,
        'g',
        pool_size=1,
        headers=headers_dict)

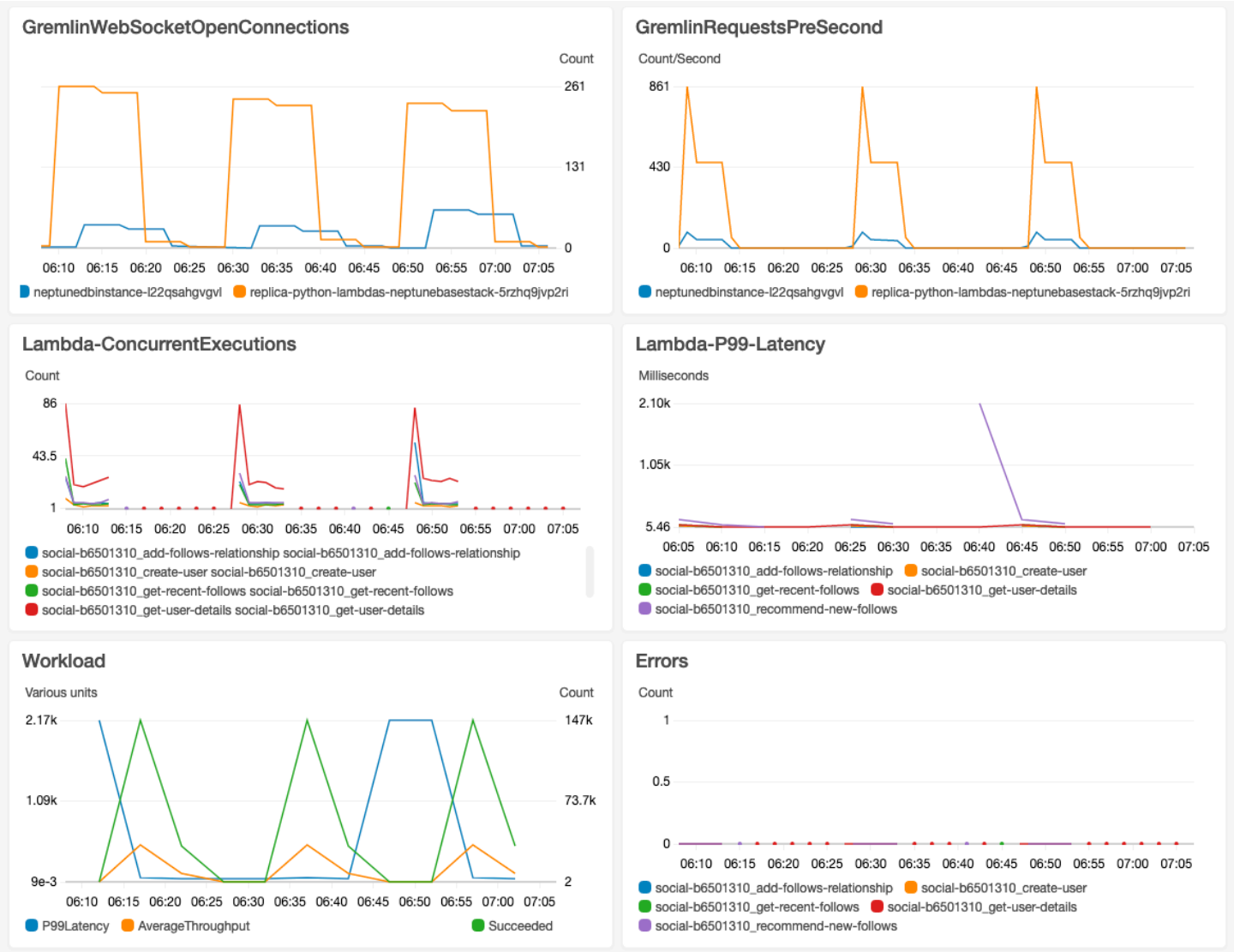

def connection_info():

    database_url = 'wss://{}:{}/gremlin'.format(os.environ['neptuneEndpoint'],
 os.environ['neptunePort'])

    if 'USE_IAM' in os.environ and os.environ['USE_IAM'] == 'true':
        return prepare_iamdb_request(database_url)
    else:
        return (database_url, {})

conn = create_remote_connection()
g = create_graph_traversal_source(conn)
```

Here are sample results, showing alternating periods of heavy and light load:


GremlinWebSocketOpenConnections
Count
261
131
0
06:10 06:15 06:20 06:25 06:30 06:35 06:40 06:45 06:50 06:55 07:00 07:05
neptunedbinstance-l22qsahgvgvl
replica-python-lambdas-neptunebasestack-5rzhq9jvp2ri
GremlinRequestsPreSecond
Count/Second
861
430
0
06:10 06:15 06:20 06:25 06:30 06:35 06:40 06:45 06:50 06:55 07:00 07:05
neptunedbinstance-l22qsahgvgvl
replica-python-lambdas-neptunebasestack-5rzhq9jvp2ri
Lambda-ConcurrentExecutions
Count
86
43.5
1
06:10 06:15 06:20 06:25 06:30 06:35 06:40 06:45 06:50 06:55 07:00 07:05
social-b6501310_add-follows-relationship social-b6501310_add-follows-relationship
social-b6501310_create-user social-b6501310_create-user
social-b6501310_get-recent-follows social-b6501310_get-recent-follows
social-b6501310_get-user-details social-b6501310_get-user-details
Lambda-P99-Latency
Milliseconds
2.10k
1.05k
5.46
06:05 06:10 06:15 06:20 06:25 06:30 06:35 06:40 06:45 06:50 06:55 07:00 07:05
social-b6501310_add-follows-relationship
social-b6501310_create-user
social-b6501310_get-recent-follows
social-b6501310_get-user-details
social-b6501310_recommend-new-follows
Workload
Various units
Count
2.17k
1.09k
9e-3
147k
73.7k
2
06:10 06:15 06:20 06:25 06:30 06:35 06:40 06:45 06:50 06:55 07:00 07:05
P99Latency
AverageThroughput
Succeeded
Errors
Count
1
0.5
0
06:10 06:15 06:20 06:25 06:30 06:35 06:40 06:45 06:50 06:55 07:00 07:05
social-b6501310_add-follows-relationship
social-b6501310_create-user
social-b6501310_get-recent-follows
social-b6501310_get-user-details
social-b6501310_recommend-new-follows

Amazon Neptune ML for machine learning on graphs

There is often valuable information in large connected datasets that can be hard to extract using queries based on human intuition alone. Machine learning (ML) techniques can help find hidden correlations in graphs with billions of relationships. These correlations can be helpful for recommending products, predicting credit worthiness, identifying fraud, and many other things.

The Neptune ML feature makes it possible to build and train useful machine learning models on large graphs in hours instead of weeks. To accomplish this, Neptune ML uses graph neural network (GNN) technology powered by Amazon SageMaker AI and the Deep Graph Library (DGL) (which is open-source). Graph neural networks are an emerging field in artificial intelligence (see, for example, A Comprehensive Survey on Graph Neural Networks). For a hands-on tutorial about using GNNs with DGL, see Learning graph neural networks with Deep Graph Library.

> **Note**
>
> Graph vertices are identified in Neptune ML models as "nodes". For example, vertex classification uses a node-classification machine learning model, and vertex regression uses a node-regression model.

What Neptune ML can do

Neptune supports both transductive inference, which returns predictions that were pre-computed at the time of training, based on your graph data at that time, and inductive inference, which returns applies data processing and model evaluation in real time, based on current data. See The difference between inductive and transductive inference.

Neptune ML can train machine learning models to support five different categories of inference:

Types of inference task currently supported by Neptune ML

- **Node classification** – predicting the categorical feature of a vertex property.

  For example, given the movie *The Shawshank Redemption*, Neptune ML can predict its `genre` property as `story` from a candidate set of `[story, crime, action, fantasy, drama, family, ...]`.

There are two types of node-classification tasks:

- **Single-class classification**: In this kind of task, each node has only one target feature. For example, the property, `Place_of_birth` of `Alan Turing` has the value UK.
- **Multi-class classification**: In this kind of task, each node can have more than one target feature. For example, the property `genre` of the film *The Godfather* has the values `crime` and `story`.

- **Node regression** – predicting a numerical property of a vertex.

  For example, given the movie *Avengers: Endgame*, Neptune ML can predict that its property `popularity` has a value of `5.0`.

- **Edge classification** – predicting the categorical feature of an edge property.

  There are two types of edge-classification tasks:

  - **Single-class classification**: In this kind of task, each edge has only one target feature. For example, a ratings edge between a user and a movie might have the property, `liked`, with a value of either "Yes" or "No".
  - **Multi-class classification**: In this kind of task, each edge can have more than one target feature. For example, a ratings between a user and movie might have multiple values for the property tag such as "Funny", "Heartwarming", "Chilling", and so on.

- **Edge regression** – predicting a numerical property of an edge.

  For example, a rating edge between a user and a movie might have the numerical property, `score`, for which Neptune ML could predict a value given a user and a movie.

- **Link prediction** – predicting the most likely destination nodes for a particular source node and outgoing edge, or the most likely source nodes for a given destination node and incoming edge.

  For example, with a drug-disease knowledge graph, given `Aspirin` as the source node, and `treats` as the outgoing edge, Neptune ML can predict the most relevant destination nodes as `heart disease`, `fever`, and so on.

  Or, with the Wikimedia knowledge graph, given `President-of` as the edge or relation and `United-States` as the destination node, Neptune ML can predict the most relevant heads as `George Washington`, `Abraham Lincoln`, `Franklin D. Roosevelt`, and so on.

> **Note**
>
> Node classification and Edge classification only support string values. That means that numerical property values such as `0` or `1` are not supported, although the string equivalents `"0"` and `"1"` are. Similarly, the Boolean property values `true` and `false` don't work, but `"true"` and `"false"` do.

With Neptune ML, you can use machine learning models that fall in two general categories:

Types of machine learning model currently supported by Neptune ML

- **Graph Neural Network (GNN) models** – These include Relational Graph Convolutional Networks (R-GCNs). GNN models work for all three types of task above.
- **Knowledge-Graph Embedding (KGE) models** – These include `TransE`, `DistMult`, and `RotatE` models. They only work for link prediction.

User defined models – Neptune ML also lets you provide your own custom model implementation for all the types of tasks listed above. You can use the Neptune ML toolkit to develop and test your python-based custom model implementation before using the Neptune ML training API with your model. See Custom models in Neptune ML for details about how to structure and organize your implementation so that it's compatible with Neptune ML's training infrastructure.

Setting up Neptune ML

The easiest way to get started with Neptune ML is to use the CloudFormation quick-start template. This template installs all necessary components, including a new Neptune DB cluster, all the necessary IAM roles, and a new Neptune graph-notebook to make working with Neptune ML easier.

You can also install Neptune ML manually, as explained in Setting up Neptune ML without using the quick-start CloudFormation template.

Using the Neptune ML AWS CloudFormation template to get started quickly in a new DB cluster

The easiest way to get started with Neptune ML is to use the CloudFormation quick-start template. This template installs all necessary components, including a new Neptune DB cluster, all the necessary IAM roles, and a new Neptune graph-notebook to make working with Neptune ML easier.

To create the Neptune ML quick-start stack

1. To launch the CloudFormation stack on the CloudFormation console, choose one of the **Launch Stack** buttons in the following table:

Region	View	Launch
US East (N. Virginia)	View	Launch Stack
US East (Ohio)	View	Launch Stack
US West (N. California)	View	Launch Stack
US West (Oregon)	View	Launch Stack
Canada (Central)	View	Launch Stack
South America (São Paulo)	View	Launch Stack
Europe (Stockholm)	View	Launch Stack
Europe (Ireland)	View	Launch Stack
Europe (London)	View	Launch Stack
Europe (Paris)	View	Launch Stack

Region	View	Launch
Europe (Frankfurt)	View	Launch Stack
Middle East (Bahrain)	View	Launch Stack
Middle East (UAE)	View	Launch Stack
Israel (Tel Aviv)	View	Launch Stack
Africa (Cape Town)	View	Launch Stack
Asia Pacific (Hong Kong)	View	Launch Stack
Asia Pacific (Tokyo)	View	Launch Stack
Asia Pacific (Seoul)	View	Launch Stack
Asia Pacific (Singapore)	View	Launch Stack
Asia Pacific (Sydney)	View	Launch Stack
Asia Pacific (Mumbai)	View	Launch Stack
China (Beijing)	View	Launch Stack
China (Ningxia)	View	Launch Stack
AWS GovCloud (US-West)	View	Launch Stack

2. On the **Select Template** page, choose **Next**.
3. On the **Specify Details** page, choose **Next**.

4. On the **Options** page, choose **Next**.
5. On the **Review** page, there are two check boxes that you need to check:

   - The first one acknowledges that AWS CloudFormation might create IAM resources with custom names.
   - The second acknowledges that AWS CloudFormation might require the `CAPABILITY_AUTO_EXPAND` capability for the new stack. `CAPABILITY_AUTO_EXPAND` explicitly allows CloudFormation to expand macros automatically when creating the stack, without prior review.

     Customers often create a change set from a processed template so that the changes made by macros can be reviewed before actually creating the stack. For more information, see the CloudFormation CreateStack API.

   Then choose **Create**.

The quick-start template creates and sets up the following:

- A Neptune DB cluster.
- The necessary IAM roles (and attaches them).
- The necessary Amazon EC2 security group.
- The necessary SageMaker AI VPC endpoints.
- A DB cluster parameter group for Neptune ML.
- The necessary parameters in that parameter group.
- The Neptune-Export service.

You can also use the following AWS CLI command to run the quick-start CloudFormation template:

```
aws cloudformation create-stack \
  --stack-name neptune-ml-fullstack-$(date '+%Y-%m-%d-%H-%M') \
  --template-url https://aws-neptune-customer-samples.s3.amazonaws.com/v2/
cloudformation-templates/neptune-ml-nested-stack.json \
  --parameters ParameterKey=EnableIAMAuthOnExportAPI,ParameterValue=(true if you have
 IAM auth enabled, or false otherwise) \
               ParameterKey=Env,ParameterValue=test$(date '+%H%M')\
  --capabilities CAPABILITY_IAM \
```

```
  --region (the AWS region, like us-east-1) \
  --disable-rollback \
  --profile (optionally, a named CLI profile of yours)
```

Setting up Neptune ML without using the quick-start CloudFormation template

This guide provides step-by-step instructions for setting up Amazon Neptune ML without using the AWS CloudFormation quick-start template. It assumes you already have a working Neptune DB cluster and covers the necessary setup, including installing the Neptune-Export service, creating custom IAM roles, and configuring your DB cluster to enable Neptune ML. The guide also explains how to create two SageMaker AI endpoints in your Neptune VPC to give the Neptune engine access to the necessary SageMaker AI management APIs. By following these instructions, you can set up Neptune ML on your existing Neptune infrastructure without relying on the CloudFormation template.

Start with a working Neptune DB cluster

If you don't use the CloudFormation quick-start template to set up Neptune ML, you will need an existing Neptune DB cluster to work with. If you want, you can use one you already have, or clone one that you are already using, or you can create a new one (see Create Neptune cluster).

Install the Neptune-Export service

If you haven't already done so, install the Neptune-Export service, as explained in Using the Neptune-Export service to export Neptune data.

Add an inbound rule to the `NeptuneExportSecurityGroup` security group that the install creates, with the following settings:

- *Type*: `Custom TCP`
- *Protocol*: `TCP`
- *Port range*: `80 - 443`
- *Source*: *`(Neptune DB cluster security group ID)`*

Create a custom NeptuneLoadFromS3 IAM role

If you have not already done so, create a custom `NeptuneLoadFromS3` IAM role, as explained in Creating an IAM role to access Amazon S3.

Create a custom NeptuneSageMakerIAMRole role

Use the IAM console to create a custom `NeptuneSageMakerIAMRole`, using the following policy:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Action": [
        "ec2:CreateNetworkInterface",
        "ec2:CreateNetworkInterfacePermission",
        "ec2:CreateVpcEndpoint",
        "ec2:DeleteNetworkInterface",
        "ec2:DeleteNetworkInterfacePermission",
        "ec2:DescribeDhcpOptions",
        "ec2:DescribeNetworkInterfaces",
        "ec2:DescribeRouteTables",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcEndpoints",
        "ec2:DescribeVpcs"
      ],
      "Resource": "*",
      "Effect": "Allow"
    },
    {
      "Action": [
        "ecr:GetAuthorizationToken",
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage",
        "ecr:BatchCheckLayerAvailability"
      ],
      "Resource": "*",
      "Effect": "Allow"
    },
    {
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws-eusc:iam::*:role/*"
      ],
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": [
```

```
            "sagemaker.amazonaws.com"
          ]
        }
      },
      "Effect": "Allow"
    },
    {
      "Action": [
        "kms:CreateGrant",
        "kms:Decrypt",
        "kms:GenerateDataKey*"
      ],
      "Resource": "arn:aws-eusc:kms:*:*:key/*",
      "Effect": "Allow"
    },
    {
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogGroups",
        "logs:DescribeLogStreams",
        "logs:GetLogEvents"
      ],
      "Resource": [
        "arn:aws-eusc:logs:*:*:log-group:/aws/sagemaker/*"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "sagemaker:AddTags",
        "sagemaker:CreateEndpoint",
        "sagemaker:CreateEndpointConfig",
        "sagemaker:CreateHyperParameterTuningJob",
        "sagemaker:CreateModel",
        "sagemaker:CreateProcessingJob",
        "sagemaker:CreateTrainingJob",
        "sagemaker:CreateTransformJob",
        "sagemaker:DeleteEndpoint",
        "sagemaker:DeleteEndpointConfig",
        "sagemaker:DeleteModel",
        "sagemaker:DescribeEndpoint",
        "sagemaker:DescribeEndpointConfig",
```

```
        "sagemaker:DescribeHyperParameterTuningJob",
        "sagemaker:DescribeModel",
        "sagemaker:DescribeProcessingJob",
        "sagemaker:DescribeTrainingJob",
        "sagemaker:DescribeTransformJob",
        "sagemaker:InvokeEndpoint",
        "sagemaker:ListTags",
        "sagemaker:ListTrainingJobsForHyperParameterTuningJob",
        "sagemaker:StopHyperParameterTuningJob",
        "sagemaker:StopProcessingJob",
        "sagemaker:StopTrainingJob",
        "sagemaker:StopTransformJob",
        "sagemaker:UpdateEndpoint",
        "sagemaker:UpdateEndpointWeightsAndCapacities"
      ],
      "Resource": [
        "arn:aws-eusc:sagemaker:*:*:*"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "sagemaker:ListEndpointConfigs",
        "sagemaker:ListEndpoints",
        "sagemaker:ListHyperParameterTuningJobs",
        "sagemaker:ListModels",
        "sagemaker:ListProcessingJobs",
        "sagemaker:ListTrainingJobs",
        "sagemaker:ListTransformJobs"
      ],
      "Resource": "*",
      "Effect": "Allow"
    },
    {
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObject",
        "s3:AbortMultipartUpload",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws-eusc:s3:::*"
      ],
```

```
      "Effect": "Allow"
    }
  ]
}
```

While creating this role, edit the trust relationship so that it reads as follows:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "ec2.amazonaws.com",
          "rds.amazonaws.com",
          "sagemaker.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Finally, copy the ARN assigned to this new `NeptuneSageMakerIAMRole` role.

Important

- Be sure that the Amazon S3 permissions in the `NeptuneSageMakerIAMRole` match those above.
- The universal ARN, `arn:aws:s3:::*` is used for the Amazon S3 resource in the policy above. If for some reason the universal ARN cannot be used, then `arn:aws:s3:::graphlytics*` and the ARN for any other customer Amazon S3 resource that NeptuneML commands will use must be added to the resource section.

Configure your DB cluster to enable Neptune ML

To set up your DB cluster for Neptune ML

1. In the Neptune console, navigate to **Parameter Groups** and then to the DB cluster parameter group associated with the DB cluster you will be using. Set the `neptune_ml_iam_role` parameter to the ARN assigned to the `NeptuneSageMakerIAMRole` role that you just created.
2. Navigate to Databases, then select the DB cluster you will be using for Neptune ML. Select **Actions** then **Manage IAM roles**.
3. On the **Manage IAM roles** page, select **Add role** and add the `NeptuneSageMakerIAMRole`. Then add the `NeptuneLoadFromS3` role.
4. Reboot the writer instance of your DB cluster.

Create two SageMaker AI endpoints in your Neptune VPC

Finally, to give the Neptune engine access the necessary SageMaker AI management APIs, you need to create two SageMaker AI endpoints in your Neptune VPC, as explained in Create two endpoints for SageMaker AI in your Neptune VPC.

Manually configuring a Neptune notebook for Neptune ML

Neptune SageMaker AI notebooks come pre-loaded with a variety of sample notebooks for Neptune ML. You can preview these samples in the open source graph-notebook GitHub repository.

You can use one of the existing Neptune notebooks, or if you want you can create one of your own, following the instructions in ???.

You can also configure a default Neptune notebook for use with Neptune ML by following these steps:

Modify a notebook for Neptune ML

1. Open the Amazon SageMaker AI console at https://eusc-de-east-1.console.amazonaws-eusc.eu/sagemaker/.
2. On the navigation pane on the left, choose **Notebook**, then **Notebook Instances**. Look for the name of the Neptune notebook that you would like to use for Neptune ML and select it to go to its details page.

3. If the notebook instance is running, select the **Stop** button at the top right of the notebook details page.
4. In **Notebook instance settings**, under **Lifecycle Configuration**, select the link to open the page for the notebook's lifecycle.
5. Select **Edit** at the top right, then **Continue**.
6. In the **Start notebook** tab, modify the script to include additional export commands and to fill in the fields for your Neptune ML IAM role and Export service URI, something like this depending on your shell:

```
echo "export NEPTUNE_ML_ROLE_ARN=(your Neptune ML IAM role ARN)" >> ~/.bashrc
echo "export NEPTUNE_EXPORT_API_URI=(your export service URI)" >> ~/.bashrc
```

7. Select **Update**.
8. Return to the notebook instance page. Under **Permissions and encryption** there is a field for **IAM role ARN**. Select the link in this field to go to the IAM role that this notebook instance runs with.
9. Create a new inline policy like this:

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Action": [
        "cloudwatch:PutMetricData"
      ],
      "Resource": "arn:aws-eusc:cloudwatch:eusc-de-east-1:111122223333:*",
      "Sid": "AllowPutMetrics",
      "Effect": "Allow"
    },
    {
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:DescribeLogStreams",
        "logs:PutLogEvents",
        "logs:GetLogEvents"
```

```
        ],
        "Resource": "arn:aws-eusc:logs:eusc-de-east-1:111122223333:*",
        "Sid": "AllowCreateLogs",
        "Effect": "Allow"
      },
      {
        "Action": [
          "s3:Put*",
          "s3:Get*",
          "s3:List*"
        ],
        "Resource": "arn:aws-eusc:s3:::*",
        "Sid": "AllowS3Actions",
        "Effect": "Allow"
      },
      {
        "Action": "execute-api:Invoke",
        "Resource": "arn:aws-eusc:execute-api:eusc-de-east-1:111122223333:*/*",
        "Sid": "AllowExecute",
        "Effect": "Allow"
      },
      {
        "Action": [
          "sagemaker:CreateModel",
          "sagemaker:CreateEndpointConfig",
          "sagemaker:CreateEndpoint",
          "sagemaker:DescribeModel",
          "sagemaker:DescribeEndpointConfig",
          "sagemaker:DescribeEndpoint",
          "sagemaker:DeleteModel",
          "sagemaker:DeleteEndpointConfig",
          "sagemaker:DeleteEndpoint"
        ],
        "Resource": "arn:aws-eusc:sagemaker:eusc-de-east-1:111122223333:*/*",
        "Sid": "AllowApiActions",
        "Effect": "Allow"
      },
      {
        "Action": [
          "iam:PassRole"
        ],
        "Resource": "arn:aws-eusc:iam::111122223333:role/role-name",
        "Sid": "AllowPassRole",
        "Effect": "Allow"
```

```
    }
  ]
}
```

10. Save this new policy and attach it to the IAM role in Step 8.
11. Select **Start** at the top right of of the SageMaker AI notebook instance details page to start the notebook instance.

Using the AWS CLI to set up Neptune ML on a DB cluster

In addition to the CloudFormation quick-start template and the AWS Management Console, you can also set up Neptune ML using the AWS CLI.

Create a DB cluster parameter group for your new Neptune ML cluster

The following AWS CLI commands create a new DB cluster parameter group and set it up to work with Neptune ML:

To create and configure a DB cluster parameter group for Neptune ML

1. Create a new DB cluster parameter group:

```
aws neptune create-db-cluster-parameter-group \
  --db-cluster-parameter-group-name (name of the new DB cluster parameter group) \
  --db-parameter-group-family neptune1
  --description "(description of your machine learning project)" \
  --region (AWS region, such as us-east-1)
```

2. Create a `neptune_ml_iam_role` DB cluster parameter set to the ARN of the `SageMakerExcecutionIAMRole` for your DB cluster to use while calling SageMaker AI for creating jobs and getting prediction from hosted ML models:

```
aws neptune modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name (name of the new DB cluster parameter group) \
  --parameters "ParameterName=neptune_ml_iam_role, \
                ParameterValue=ARN of the SageMakerExcecutionIAMRole, \
                Description=NeptuneMLRole, \
                ApplyMethod=pending-reboot" \
  --region (AWS region, such as us-east-1)
```

   Setting this parameter allows Neptune to access SageMaker AI without you having to pass in the role with every call.

   For information about how to create the `SageMakerExcecutionIAMRole`, see Create a custom NeptuneSageMakerIAMRole role.

3. Finally, use `describe-db-cluster-parameters` to check that all the parameters in the new DB cluster parameter group are set as you want them to be:

```
aws neptune describe-db-cluster-parameters \
  --db-cluster-parameter-group-name (name of the new DB cluster parameter group) \
  --region (AWS region, such as us-east-1)
```

Attach the new DB cluster parameter group to the DB cluster you will use with Neptune ML

Now you can attach the new DB cluster parameter group that you just created to an existing DB cluster by using the following command:

```
aws neptune modify-db-cluster \
  --db-cluster-identifier (the name of your existing DB cluster) \
  --apply-immediately
  --db-cluster-parameter-group-name (name of your new DB cluster parameter group) \
  --region (AWS region, such as us-east-1)
```

To make all the parameters effective, you can then reboot the DB cluster:

```
aws neptune reboot-db-instance
  --db-instance-identifier (name of the primary instance of your DB cluster) \
  --profile (name of your AWS profile to use) \
  --region (AWS region, such as us-east-1)
```

Or, if you're creating a new DB cluster to use with Neptune ML, you can use the following command to create the cluster with the new parameter group attached, and then create a new primary (writer) instance:

```
cluster-name=(the name of the new DB cluster)
aws neptune create-db-cluster
  --db-cluster-identifier ${cluster-name}
  --engine graphdb \
  --engine-version 1.0.4.1 \
  --db-cluster-parameter-group-name (name of your new DB cluster parameter group) \
  --db-subnet-group-name (name of the subnet to use) \
  --region (AWS region, such as us-east-1)

aws neptune create-db-instance
  --db-cluster-identifier ${cluster-name}
  --db-instance-identifier ${cluster-name}-i \
```

```
  --db-instance-class (the instance class to use, such as db.r5.xlarge)
  --engine graphdb \
  --region (AWS region, such as us-east-1)
```

Attach the `NeptuneSageMakerIAMRole` to your DB cluster so that it can access SageMaker AI and Amazon S3 resources

Finally, follow the instructions in Create a custom NeptuneSageMakerIAMRole role to create an IAM role that will allow your DB cluster to communicate with SageMaker AI and Amazon S3. Then, use the following command to attach the `NeptuneSageMakerIAMRole` role you created to your DB cluster:

```
aws neptune add-role-to-db-cluster
  --db-cluster-identifier ${cluster-name}
  --role-arn arn:aws:iam::(the ARN number of the role's ARN):role/NeptuneMLRole \
  --region (AWS region, such as us-east-1)
```

Create two endpoints for SageMaker AI in your Neptune VPC

Neptune ML needs two SageMaker AI endpoints in your Neptune DB cluster's VPC:

- `com.amazonaws.`*(AWS region, like us-east-1)*`.sagemaker.runtime`
- `com.amazonaws.`*(AWS region, like us-east-1)*`.sagemaker.api`

If you haven't used the quick-start CloudFormation template, which creates these automatically for you, you can use the following AWS CLI commands to create them:

This one creates the `sagemaker.runtime` endpoint:

```
aws ec2 create-vpc-endpoint
  --vpc-id (the ID of your Neptune DB cluster's VPC)
  --vpc-endpoint-type Interface
  --service-name com.amazonaws.(AWS region, like us-east-1).sagemaker.runtime
  --subnet-ids (the subnet ID or IDs that you want to use)
  --security-group-ids (the security group for the endpoint network interface, or omit
 to use the default)
  --private-dns-enabled
```

And this one creates the `sagemaker.api` endpoint:

```
aws ec2 create-vpc-endpoint
  --vpc-id (the ID of your Neptune DB cluster's VPC)
  --vpc-endpoint-type Interface
  --service-name com.amazonaws.(AWS region, like us-east-1).sagemaker.api
  --subnet-ids (the subnet ID or IDs that you want to use)
  --security-group-ids (the security group for the endpoint network interface, or omit
 to use the default)
  --private-dns-enabled
```

You can also use the VPC console to create these endpoints. See Secure prediction calls in Amazon SageMaker with AWS PrivateLink and Securing all Amazon SageMaker API calls with AWS PrivateLink.

Create a SageMaker AI inference endpoint parameter in your DB cluster parameter group

To avoid having to specify the SageMaker AI inference endpoint of the model that you're using in every query you make to it, create a DB cluster parameter named `neptune_ml_endpoint` in the DB cluster parameter group for Neptune ML. Set the parameter to the `id` of the instance endpoint in question.

You can use the following AWS CLI command to do that:

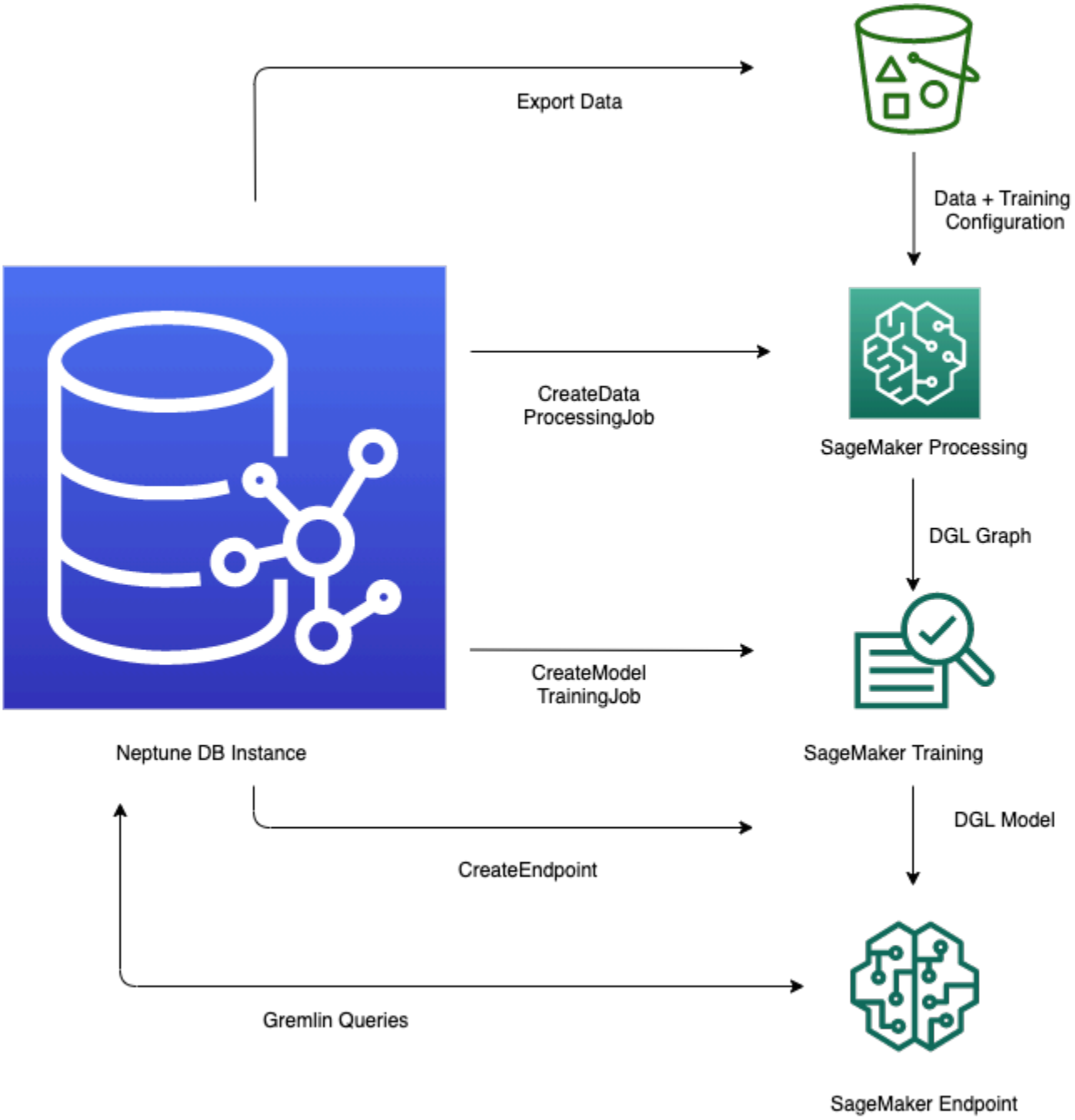
```
aws neptune modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name neptune-ml-demo \
  --parameters "ParameterName=neptune_ml_endpoint, \
                ParameterValue=(the name of the SageMaker AI inference endpoint you
 want to query), \
                Description=NeptuneMLEndpoint, \
                ApplyMethod=pending-reboot" \
  --region (AWS region, such as us-east-1)
```

Overview of how to use the Neptune ML feature

The Neptune ML feature in Amazon Neptune provides a streamlined workflow for leveraging machine learning models within a graph database. The process involves several key steps - exporting data from Neptune into CSV format, preprocessing the data to prepare it for model training, training the machine learning model using Amazon SageMaker AI, creating an inference endpoint to serve predictions, and then querying the model directly from Gremlin queries.
The Neptune workbench provides convenient line and cell magic commands to help manage and automate these steps. By integrating machine learning capabilities directly into the graph database, Neptune ML enables users to derive valuable insights and make predictions using the rich relational data stored in the Neptune graph.

Starting workflow for using Neptune ML

Using the Neptune ML feature in Amazon Neptune generally involves the following five steps to begin with:



1. **Data export and configuration** – The data-export step uses the Neptune-Export service or the `neptune-export` command line tool to export data from Neptune into Amazon Simple Storage Service (Amazon S3) in CSV form. A configuration file named `training-data-configuration.json` is automatically generated at the same time, which specifies how the exported data can be loaded into a trainable graph.
2. **Data preprocessing** – In this step, the exported dataset is preprocessed using standard techniques to prepare it for model training. Feature normalization can be performed for numeric

data, and text features can be encoded using `word2vec`. At the end of this step, a DGL (Deep Graph library) graph is generated from the exported dataset for the model training step to use.

This step is implemented using a SageMaker AI processing job in your account, and the resulting data is stored in an Amazon S3 location that you have specified.

3. **Model training** – The model training step trains the machine learning model that will be used for predictions.

   Model training is done in two stages:

   - The first stage uses a SageMaker AI processing job to generate a model training strategy configuration set that specifies what type of model and model hyperparameter ranges will be used for the model training.
   - The second stage then uses a SageMaker AI model tuning job to try different hyperparameter configurations and select the training job that produced the best-performing model. The tuning job runs a pre-specified number of model training job trials on the processed data. At the end of this stage, the trained model parameters of the best training job are used to generate model artifacts for inference.
4. **Create an inference endpoint in Amazon SageMaker AI** – The inference endpoint is a SageMaker AI endpoint instance that is launched with the model artifacts produced by the best training job. Each model is tied to a single endpoint. The endpoint is able to accept incoming requests from the graph database and return the model predictions for inputs in the requests. After you have created the endpoint, it stays active until you delete it.
5. **Query the machine learning model using Gremlin** – You can use extensions to the Gremlin query language to query predictions from the inference endpoint.

Making predictions based on evolving graph data

With a continuously changing graph, you may want to create new batch predictions periodically using fresh data. Querying pre-computed predictions (transductive inference) can be significantly faster than generating new predictions on the fly based on the very latest data (inductive inference). Both approaches have their place, depending on how rapidly your data changes and on your performance requirements.

The difference between inductive and transductive inference

When performing transductive inference, Neptune looks up and returns predictions that were pre-computed at the time of training.

When performing inductive inference, Neptune constructs the relevant subgraph and fetches its properties. The DGL GNN model then applies data processing and model evaluation in real-time.

Inductive inference can therefore generate predictions involving nodes and edges that were not present at the time of training and that reflect the current state of the graph. This comes, however, at the cost of higher latency.

If your graph is dynamic, you may want to use inductive inference to be sure to take into account the latest data, but if your graph is static, transductive inference is faster and more efficient.

Inductive inference is disabled by default. You can enable it for a query by using the Gremlin Neptune#ml.inductiveInference predicate in the query as follows:

```
.with( "Neptune#ml.inductiveInference")
```

Incremental transductive workflows

While you update model artifacts simply by re-running the steps one through three (from **Data export and configuration** to **Model transform**), Neptune ML supports simpler ways to update your batch ML predictions using new data. One is to use an incremental-model workflow, and another is to use model retraining with a warm start.

Incremental-model workflow

In this workflow, you update the ML predictions without retraining the ML model.

> **Note**
>
> You can only do this when the graph data has been updated with new nodes and/or edges. It will not currently work when nodes are removed.

1. **Data export and configuration** – This step is the same as in the main workflow.
2. **Incremental data preprocessing** – This step is similar to the data preprocessing step in the main workflow, but uses the same processing configuration used previously, that corresponds to a specific trained model.
3. **Model transform** – Instead of a model training step, this model-transform step takes the trained model from the main workflow and the results of the incremental data preprocessing

step, and generates new model artifacts to use for inference. The model-transform step launches a SageMaker AI processing job to perform the computation that generates the updated model artifacts.

4. **Update the Amazon SageMaker AI inference endpoint** – Optionally, if you have an existing inference endpoint, this step updates the endpoint with the new model artifacts generated by the model-transform step. Alternatively, you can also create a new inference endpoint with the new model artifacts.

Model re-training with a warm start

Using this workflow, you can train and deploy a new ML model for making predictions using the incremental graph data, but start from an existing model generated using the main workflow:

1. **Data export and configuration** – This step is the same as in the main workflow.
2. **Incremental data preprocessing** – This step is the same as in the incremental model inference workflow. The new graph data should be processed with the same processing method that was used previously for model training.
3. **Model training with a warm start** – Model training is similar to what happens in the main workflow, but you can speed up model hyperparameter search by leveraging the information from the previous model training task.
4. **Update the Amazon SageMaker AI inference endpoint** – This step is the same as in the incremental model inference workflow.

Workflows for custom models in Neptune ML

Neptune ML lets you implement, train and deploy custom models of your own for any of the tasks that Neptune ML supports. The workflow for developing and deploying a custom model is essentially the same as for the built-in models, with a few differences, as explained in Custom model workflow.

Instance selection for the Neptune ML stages

The different stages of Neptune ML processing use different SageMaker AI instances. Here, we discuss how to choose the right instance type for each stage. You can find information about SageMaker AI instance types and pricing at Amazon SageMaker Pricing.

Selecting an instance for data processing

The SageMaker AI data-processing step requires a processing instance that has enough memory and disk storage for the input, intermediate, and output data. The specific amount of memory and disk storage needed depends on the characteristics of the Neptune ML graph and its exported features.

By default, Neptune ML chooses the smallest `ml.r5` instance whose memory is ten times larger than the size of the exported graph data on disk.

Selecting an instance for model training and model transform

Selecting the right instance type for model training or model transform depends on the task type, the graph size, and your turn-around requirements. GPU instances provide the best performance. We generally recommend `p3` and `g4dn` serial instances. You can also use `p2` or `p4d` instances.

By default, Neptune ML chooses the smallest GPU instance with more memory than model training and model transform requires. You can find what that selection is in the `train_instance_recommendation.json` file, in the Amazon S3 data processing output location. Here is an example of the contents of a `train_instance_recommendation.json` file:

```
{
  "instance":     "(the recommended instance type for model training and transform)",
  "cpu_instance": "(the recommended instance type for base processing instance)",
  "disk_size":    "(the estimated disk space required)",
  "mem_size":     "(the estimated memory required)"
}
```

Selecting an instance for an inference endpoint

Selecting the right instance type for an inference endpoint depends on the task type, the graph size and your budget. By default, Neptune ML chooses the smallest `ml.m5d` instance with more memory the inference endpoint requires.

> **Note**
>
> If more than 384 GB of memory is needed, Neptune ML uses an `ml.r5d.24xlarge` instance.

You can see what instance type Neptune ML recommends in the `infer_instance_recommendation.json` file located in the Amazon S3 location you are using for model training. Here is an example of that file's contents:

```
{
  "instance" :   "(the recommended instance type for an inference endpoint)",
  "disk_size" :  "(the estimated disk space required)",
  "mem_size" :   "(the estimated memory required)"
}
```

Using the neptune-export tool or Neptune-Export service to export data from Neptune for Neptune ML

Neptune ML requires that you provide training data for the Deep Graph Library (DGL) to create and evaluate models.

You can export data from Neptune using either the Neptune-Export service, or `neptune-export` utility. Both the service and the command line tool publish data to Amazon Simple Storage Service (Amazon S3) in a CSV format, encrypted using Amazon S3 server-side encryption (`SSE-S3`). See Files exported by Neptune-Export and `neptune-export`.

In addition, when you configure an export of training data for Neptune ML the export job creates and publishes an encrypted model-training configuration file along with the exported data. By default, this file is named `training-data-configuration.json`.

Examples of using the Neptune-Export service to export training data for Neptune ML

This request exports property-graph training data for a node classification task:

curl

```
curl \
  (your NeptuneExportApiUri) \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "command": "export-pg",
        "outputS3Path": "s3:////(your Amazon S3 bucket)/neptune-export",
        "params": {
          "endpoint": "(your Neptune endpoint DNS name)",
          "profile": "neptune_ml"
        },
        "additionalParams": {
          "neptune_ml": {
            "version": "v2.0",
            "targets": [
              {
                "node": "Movie",
                "property": "genre",
```

```
                "type": "classification"
              }
            ]
          }
        }
      }'
```

awscurl

```
awscurl (your NeptuneExportApiUri) \
  --region us-east-1 \
  --service execute-api \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "command": "export-pg",
        "outputS3Path": "s3:////(your Amazon S3 bucket)/neptune-export",
        "params": {
          "endpoint": "(your Neptune endpoint DNS name)",
          "profile": "neptune_ml"
        },
        "additionalParams": {
          "neptune_ml": {
            "version": "v2.0",
            "targets": [
              {
                "node": "Movie",
                "property": "genre",
                "type": "classification"
              }
            ]
          }
        }
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

This request exports RDF training data for a node classification task:

curl

```
curl \
  (your NeptuneExportApiUri) \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "command": "export-rdf",
        "outputS3Path": "s3:///(your Amazon S3 bucket)/neptune-export",
        "params": {
          "endpoint": "(your Neptune endpoint DNS name)",
          "profile": "neptune_ml"
        },
        "additionalParams": {
          "neptune_ml": {
            "version": "v2.0",
            "targets": [
              {
                "node": "http://aws.amazon.com/neptune/csv2rdf/class/Movie",
                "predicate": "http://aws.amazon.com/neptune/csv2rdf/
datatypeProperty/genre",
                "type": "classification"
              }
            ]
          }
        }
      }'
```

awscurl

```
awscurl (your NeptuneExportApiUri) \
  --region us-east-1 \
  --service execute-api \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "command": "export-rdf",
        "outputS3Path": "s3:///(your Amazon S3 bucket)/neptune-export",
        "params": {
          "endpoint": "(your Neptune endpoint DNS name)",
          "profile": "neptune_ml"
        },
        "additionalParams": {
```

```
          "neptune_ml": {
            "version": "v2.0",
            "targets": [
              {
                "node": "http://aws.amazon.com/neptune/csv2rdf/class/Movie",
                "predicate": "http://aws.amazon.com/neptune/csv2rdf/
datatypeProperty/genre",
                "type": "classification"
              }
            ]
          }
        }
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

Fields to set in the params object when exporting training data

The `params` object in an export request can contain various fields, as described in the `params` documentation. The following ones are most relevant for exporting machine-learning training data:

- **`endpoint`** – Use `endpoint` to specify an endpoint of a Neptune instance in your DB cluster that the export process can query to extract data.
- **`profile`** – The `profile` field in the `params` object must be set to **`neptune-ml`**.

  This causes the export process to format the exported data appropriately for Neptune ML model training, in a CSV format for property-graph data or as N-Triples for RDF data. It also causes a `training-data-configuration.json` file to be created and written to the same Amazon S3 location as the exported training data.
- **`cloneCluster`** – If set to `true`, the export process clones your DB cluster, exports from the clone, and then deletes the clone when it is finished.
- **`useIamAuth`** – If your DB cluster has IAM authentication enabled, you must include this field set to `true`.

The export process also provides several ways to filter the data you export (see these examples).

Using the additionalParams object to tune the export of model-training information

The `additionalParams` object contains fields that you can use to specify machine-learning class labels and features for training purposes and guide the creation of a training data configuration file.

The export process cannot automatically infer which node and edge properties should be the machine learning class labels to serve as examples for training purposes. It also cannot automatically infer the best feature encoding for numeric, categorical and text properties, so you need to supply hints using fields in the `additionalParams` object to specify these things, or to override the default encoding.

For property-graph data, the top-level structure of `additionalParams` in an export request might look like this:

```
{
  "command": "export-pg",
  "outputS3Path": "s3:////(your Amazon S3 bucket)/neptune-export",
  "params": {
    "endpoint": "(your Neptune endpoint DNS name)",
    "profile": "neptune_ml"
  },
  "additionalParams": {
      "neptune_ml": {
        "version": "v2.0",
        "targets": [ (an array of node and edge class label targets) ],
        "features": [ (an array of node feature hints) ]
    }
  }
}
```

For RDF data, its top-level structure might look like this:

```
{
  "command": "export-rdf",
  "outputS3Path": "s3:////(your Amazon S3 bucket)/neptune-export",
  "params": {
    "endpoint": "(your Neptune endpoint DNS name)",
```

```
    "profile": "neptune_ml"
  },
  "additionalParams": {
      "neptune_ml": {
        "version": "v2.0",
        "targets": [ (an array of node and edge class label targets) ]
    }
  }
}
```

You can also supply multiple export configurations, using the `jobs` field:

```
{
  "command": "export-pg",
  "outputS3Path": "s3:////(your Amazon S3 bucket)/neptune-export",
  "params": {
    "endpoint": "(your Neptune endpoint DNS name)",
    "profile": "neptune_ml"
  },
  "additionalParams" : {
    "neptune_ml" : {
      "version": "v2.0",
      "jobs": [
        {
          "name" : "(training data configuration name)",
          "targets": [ (an array of node and edge class label targets) ],
          "features": [ (an array of node feature hints) ]
        },
        {
          "name" : "(another training data configuration name)",
          "targets": [ (an array of node and edge class label targets) ],
          "features": [ (an array of node feature hints) ]
        }
      ]
    }
  }
}
```

Top-level elements in the neptune_ml field in additionalParams

The version element in neptune_ml

Specifies the version of training data configuration to generate.

(*Optional*), *Type*: string, *Default*: "v2.0".

If you do include `version`, set it to `v2.0`.

The jobs field in neptune_ml

Contains an array of training-data configuration objects, each of which defines a data processing job, and contains:

- **`name`** – The name of the training data configuration to be created.

  For example, a training data configuration with the name "job-number-1" results in a training data configuration file named `job-number-1.json`.
- **`targets`** – A JSON array of node and edge class label targets that represent the machine-learning class labels for training purposes. See The targets field in a neptune_ml object.
- **`features`** – A JSON array of node property features. See The features field in neptune_ml.

The targets field in a neptune_ml object

The `targets` field in a JSON training data export configuration contains an array of target objects that specify a training task and and the machine-learning class labels for training this task. The contents of the target objects varies depending on whether you are training on property-graph data or RDF data.

For property-graph node classification and regression tasks, target objects in the array can look like this:

```
{
  "node": "(node property-graph label)",
  "property": "(property name)",
  "type" : "(used to specify classification or regression)",
  "split_rate": [0.8,0.2,0.0],
  "separator": ","
}
```

For property-graph edge classification, regression or link prediction tasks, they can look like this:

```
{
  "edge": "(edge property-graph label)",
  "property": "(property name)",
  "type" : "(used to specify classification, regression or link_prediction)",
  "split_rate": [0.8,0.2,0.0],
  "separator": ","
}
```

For RDF classification and regression tasks, target objects in the array can look like this:

```
{
  "node": "(node type of an RDF node)",
  "predicate": "(predicate IRI)",
  "type" : "(used to specify classification or regression)",
  "split_rate": [0.8,0.2,0.0]
}
```

For RDF link prediction tasks, target objects in the array can look like this::

```
{
  "subject": "(source node type of an edge)",
```

```
  "predicate": "(relation type of an edge)",
  "object": "(destination node type of an edge)",
  "type" : "link_prediction",
  "split_rate": [0.8,0.2,0.0]
}
```

Target objects can contain the following fields:

Contents

- Fields in a property-graph target object
  - The node (vertex) field in a target object
  - The edge field in a property-graph target object
  - The property field in a property-graph target object
  - The type field in a property-graph target object
  - The split_rate field in a property-graph target object
  - The separator field in a property-graph target object
- Fields in an RDF target object
  - The node field in an RDF target object
  - The subject field in an RDF target object
  - The predicate field in an RDF target object
  - The object field in an RDF target object
  - The type field in an RDF target object
  - The split_rate field in a property-graph target object

Fields in a property-graph target object

The node (vertex) field in a target object

The property-graph label of a target node (vertex). A target object must contain a node element or an edge element, but not both.

A node can take either a single value, like this:

```
  "node": "Movie"
```

Or, in the case of a multi-label vertex, it can take an array of values, like this:

```
"node": ["Content", "Movie"]
```

The edge field in a property-graph target object

Specifies a target edge by its start node label(s), its own label, and its end-node label(s). A target object must contain an `edge` element or a `node` element, but not both.

The value of an `edge` field is a JSON array of three strings that represent the start-node's property-graph label(s), the property-graph label of the edge itself, and the end-node's property-graph label(s), like this:

```
"edge": ["Person_A", "knows", "Person_B"]
```

If the start node and/or end node has multiple labels, enclose them in an array, like this:

```
"edge": [ ["Admin", Person_A"], "knows", ["Admin", "Person_B"] ]
```

The property field in a property-graph target object

Specifies a property of the target vertex or edge, like this:

```
"property" : "rating"
```

This field is required, except when the target task is link prediction.

The type field in a property-graph target object

Indicates the type of target task to be performed on the `node` or `edge`, like this:

```
"type" : "regression"
```

The supported task types for nodes are:

- `classification`
- `regression`

The supported task types for edges are:

- `classification`

- `regression`
- `link_prediction`

This field is required.

The split_rate field in a property-graph target object

(*Optional*) An estimate of the proportions of nodes or edges that the training, validation, and test stages will use, respectively. These proportions are represented by a JSON array of three numbers between zero and one that add up to one:

```
"split_rate": [0.7, 0.1, 0.2]
```

If you do not supply the optional `split_rate` field, the default estimated value is `[0.9, 0.1, 0.0]` for classification and regression tasks, and `[0.9,0.05, 0.05]` for link prediction tasks.

The separator field in a property-graph target object

(*Optional*) Used with a classification task.

The `separator` field specifies a character used to split a target property value into multiple categorical values when it is used to store multiple category values in a string. For example:

```
"separator": "|"
```

The presence of a `separator` field indicates that the task is a multi-target classification task.

Fields in an RDF target object

The node field in an RDF target object

Defines the node type of target nodes. Used with node classification tasks or node regression tasks. The node type of a node in RDF is defined by:

```
  node_id, <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>, node_type
```

An RDF node can only take a single value, like this:

```
  "node": "http://aws.amazon.com/neptune/csv2rdf/class/Movie"
```

The subject field in an RDF target object

For link prediction tasks, `subject` defines the source node type of target edges.

```
"subject": "http://aws.amazon.com/neptune/csv2rdf/class/Director"
```

> **Note**
>
> For link prediction tasks, `subject` should be used together with `predicate` and `object`. If any of these three is not provided, all edges are treated as the training target.

The predicate field in an RDF target object

For node classification and node regression tasks, `predicate` defines what literal data is used as the target node feature of a target node.

```
"predicate": "http://aws.amazon.com/neptune/csv2rdf/datatypeProperty/genre"
```

> **Note**
>
> If the target nodes have only one predicate defining the target node feature, the `predicate` field can be omitted.

For link prediction tasks, `predicate` defines the relation type of target edges:

```
"predicate": "http://aws.amazon.com/neptune/csv2rdf/datatypeProperty/direct"
```

> **Note**
>
> For link prediction tasks, `predicate` should be used together with `subject` and `object`. If any of these three is not provided, all edges are treated as the training target.

The object field in an RDF target object

For link prediction tasks, `object` defines the destination node type of target edges:

```
"object": "http://aws.amazon.com/neptune/csv2rdf/class/Movie"
```

> **Note**
>
> For link prediction tasks, `object` should be used together with `subject` and `predicate`. If any of these three is not provided, all edges are treated as the training target.

The type field in an RDF target object

Indicates the type of target task to be performed, like this:

```
"type" : "regression"
```

The supported task types for RDF data are:

- `link_prediction`
- `classification`
- `regression`

This field is required.

The `split_rate` field in a property-graph target object

(*Optional*) An estimate of the proportions of nodes or edges that the training, validation, and test stages will use, respectively. These proportions are represented by a JSON array of three numbers between zero and one that add up to one:

```
"split_rate": [0.7, 0.1, 0.2]
```

If you do not supply the optional `split_rate` field, the default estimated value is `[0.9, 0.1, 0.0]`.

The features field in neptune_ml

Property values and RDF literals come in different formats and data types. To achieve good performance in machine learning, it is essential to convert those values to numerical encodings known as *features*.

Neptune ML performs feature extraction and encoding as part of the data-export and data-processing steps, as described in Feature encoding in Neptune ML.

For property-graph datasets, the export process automatically infers `auto` features for string properties and for numeric properties that contain multiples values. For numeric properties containing single values, it infers `numerical` features. For date properties it infers `datetime` features.

If you want to override an auto-inferred feature specification, or add a bucket numerical, TF-IDF, FastText, or SBERT specification for a property, you can control the feature encoding using the features field.

> **Note**
>
> You can only use the `features` field to control the feature specifications for property-graph data, not for RDF data.

For free-form text, Neptune ML can use several different models to convert the sequence of tokens in a string property value into a fixed-size real-value vector:

- `text_fasttext` – Uses fastText encoding. This is the recommended encoding for features that use one and only one of the five languages that fastText supports.
- `text_sbert` – Uses the Sentence BERT (SBERT) encoding models. This is the recommended encoding for text that `text_fasttext` does not support.
- `text_word2vec` – Uses Word2Vec algorithms originally published by Google to encode text. Word2Vec only supports English.
- `text_tfidf` – Uses a term frequency–inverse document frequency (TF-IDF) vectorizer for encoding text. TF-IDF encoding supports statistical features that the other encodings do not.

The `features` field contains a JSON array of node property features. Objects in the array can contain the following fields:

Contents

- The node field in features
- The edge field in features
- The property field in features
- Possible values of the type field for features
- The norm field
- The language field
- The max_length field
- The separator field
- The range field
- The bucket_cnt field
- The slide_window_size field
- The imputer field
- The max_features field
- The min_df field
- The ngram_range field
- The datetime_parts field

The node field in features

The node field specifies a property-graph label of a feature vertex. For example:

```
"node": "Person"
```

If a vertex has multiple labels, use an array to contain them. For example:

```
"node": ["Admin", "Person"]
```

The edge field in features

The edge field specifies the edge type of a feature edge. An edge type consists of an array containing the property-graph label(s) of the source vertex, the property-graph label of the edge, and the property-graph label(s) of the destination vertex. You must supply all three values when specifying an edge feature. For example:

```
"edge": ["User", "reviewed", "Movie"]
```

If a source or destination vertex of an edge type has multiple labels, use another array to contain them. For example:

```
"edge": [["Admin", "Person"]. "edited", "Post"]
```

The property field in features

Use the property parameter to specify a property of the vertex identified by the node parameter. For example:

```
"property" : "age"
```

Possible values of the type field for features

The `type` parameter specifies the type of feature being defined. For example:

```
"type": "bucket_numerical"
```

Possible values of the `type` parameter

- **`"auto"`** – Specifies that Neptune ML should automatically detect the property type and apply a proper feature encoding. An `auto` feature can also have an optional `separator` field.

  See Auto feature encoding in Neptune ML.
- **`"category"`** – This feature encoding represents a property value as one of a number of categories. In other words, the feature can take one or more discrete values. A `category` feature can also have an optional `separator` field.

  See Categorical features in Neptune ML.
- **`"numerical"`** – This feature encoding represents numerical property values as numbers in a continuous interval where "greater than" and "less than" have meaning.

  A `numerical` feature can also have optional `norm`, `imputer`, and `separator` fields.

  See Numerical features in Neptune ML.

- **`"bucket_numerical"`** – This feature encoding divides numerical property values into a set of *buckets* or categories.

  For example, you could encode people's ages in 4 buckets: kids (0-20), young-adults (20-40), middle-aged (40-60), and elders (60 and up).

  A `bucket_numerical` feature requires a `range` and a `bucket_cnt` field, and can optionally also include an `imputer` and/or `slide_window_size` field.

  See Bucket-numerical features in Neptune ML.

- **`"datetime"`** – This feature encoding represents a datetime property value as an array of these categorical features: year, month, weekday, and hour.

  One or more of these four categories can be eliminated using the `datetime_parts` parameter.

  See Datetime features in Neptune ML.

- **`"text_fasttext"`** – This feature encoding converts property values that consist of sentences or free-form text into numeric vectors using fastText models. It supports five languages, namely English (`en`), Chinese (`zh`), Hindi (`hi`), Spanish (`es`), and French (`fr`). For text property values in any one those five languages, `text_fasttext` is the recommended encoding. However, it cannot handle cases where the same sentence contains words in more than one language.

  For other languages than the ones that fastText supports, use `text_sbert` encoding.

  If you have many property value text strings longer than, say, 120 tokens, use the `max_length` field to limit the number of tokens in each string that `"text_fasttext"` encodes.

  See *fastText* encoding of text property values in Neptune ML.

- **`"text_sbert"`** – This encoding converts text property values into numeric vectors using Sentence BERT (SBERT) models. Neptune supports two SBERT methods, namely `text_sbert128`, which is the default if you just specify `text_sbert`, and `text_sbert512`. The difference between them is the maximum number of tokens in a text property that gets encoded. The `text_sbert128` encoding only encodes the first 128 tokens, while `text_sbert512` encodes up to 512 tokens. As a result, using `text_sbert512` can require more processing time than `text_sbert128`. Both methods are slower than `text_fasttext`.

  The `text_sbert*` methods support many languages, and can encode a sentence that contains more than one language.

See Sentence BERT (SBERT) encoding of text features in Neptune ML.

- **`"text_word2vec"`** – This encoding converts text property values into numeric vectors using Word2Vec algorithms. It only supports English.

  See Word2Vec encoding of text features in Neptune ML.

- **`"text_tfidf"`** – This encoding converts text property values into numeric vectors using a term frequency–inverse document frequency (TF-IDF) vectorizer.

  You define the parameters of a `text_tfidf` feature encoding using the `ngram_range` field, the `min_df` field, and the `max_features` field.

  See TF-IDF encoding of text features in Neptune ML.

- **`"none"`** – Using the none type causes no feature encoding to occur. The raw property values are parsed and saved instead.

  Use none only if you plan to perform your own custom feature encoding as part of custom model training.

The norm field

This field is required for numerical features. It specifies a normalization method to use on numeric values:

```
"norm": "min-max"
```

The following normalization methods are supported:

- **"min-max"** – Normalize each value by subtracting the minimum value from it and then dividing it by the difference between the maximum value and the minimum.
- **"standard"** – Normalize each value by dividing it by the sum of all the values.
- **"none"** – Don't normalize the numerical values during encoding.

See Numerical features in Neptune ML.

The language field

The language field specifies the language used in text property values. Its usage depends on the text encoding method:

- For `text_fasttext` encoding, this field is required, and must specify one of the following languages:
  - `en` (English)
  - `zh` (Chinese)
  - `hi` (Hindi)
  - `es` (Spanish)
  - `fr` (French)
- For `text_sbert` encoding, this field is not used, since SBERT encoding is multilingual.
- For `text_word2vec` encoding, this field is optional, since `text_word2vec` only supports English. If present, it must specify the name of the English language model:

  ```
  "language" : "en_core_web_lg"
  ```

- For `text_tfidf` encoding, this field is not used.

The max_length field

The `max_length` field is optional for `text_fasttext` features, where it specifies the maximum number of tokens in an input text feature that will be encoded. Input text that is longer than `max_length` is truncated. For example, setting max_length to 128 indicates that any tokens after the 128th in a text sequence will be ignored:

```
"max_length": 128
```

The separator field

This field is used optionally with `category`, `numerical` and `auto` features. It specifies a character that can be used to split a property value into multiple categorical values or numerical values:

```
"separator": ";"
```

Only use the `separator` field when the property stores multiple delimited values in a single string, such as `"Actor;Director"` or `"0.1;0.2"`.

See Categorical features, Numerical features, and Auto encoding.

The range field

This field is required for `bucket_numerical` features. It specifies the range of numerical values that are to be divided into buckets, in the format `[`*`lower-bound`*`,` *`upper-bound`*`]`:

```
"range" : [20, 100]
```

If a property value is smaller than the lower bound then it is assigned to the first bucket, or if it's larger than the upper bound, it's assigned to the last bucket.

See Bucket-numerical features in Neptune ML.

The bucket_cnt field

This field is required for `bucket_numerical` features. It specifies the number of buckets that the numerical range defined by the `range` parameter should be divided into:

```
"bucket_cnt": 10
```

See Bucket-numerical features in Neptune ML.

The slide_window_size field

This field is used optionally with `bucket_numerical` features to assign values to more than one bucket:

```
"slide_window_size": 5
```

The way a slide window works is that Neptune ML takes the window size **s** and transforms each numeric value **v** of a property into a range from `v - s/2` through `v + s/2`. The value is then assigned to every bucket that the range overlaps.

See Bucket-numerical features in Neptune ML.

The imputer field

This field is used optionally with `numerical` and `bucket_numerical` features to provide an imputation technique for filling in missing values:

```
"imputer": "mean"
```

The supported imputation techniques are:

- `"mean"`
- `"median"`
- `"most-frequent"`

If you don't include the imputer parameter, data preprocessing halts and exits when a missing value is encountered.

See Numerical features in Neptune ML and Bucket-numerical features in Neptune ML.

The max_features field

This field is used optionally by `text_tfidf` features to specify the maximum number of terms to encode:

```
"max_features": 100
```

A setting of 100 causes the TF-IDF vectorizer to encode only the 100 most common terms. The default value if you don't include `max_features` is 5,000.

See TF-IDF encoding of text features in Neptune ML.

The min_df field

This field is used optionally by `text_tfidf` features to specify the minimum document frequency of terms to encode:

```
"min_df": 5
```

A setting of 5 indicates that a term must appear in at least 5 different property values in order to be encoded.

The default value if you don't include the `min_df` parameter is 2.

See TF-IDF encoding of text features in Neptune ML.

The ngram_range field

This field is used optionally by `text_tfidf` features to specify what size sequences of words or tokens should be considered as potential individual terms to encode:

```
"ngram_range": [2, 4]
```

The value `[2, 4]` specifies that sequences of 2, 3 and 4 words should be considered as potential individual terms.

The default if you don't explicitly set `ngram_range` is `[1, 1]`, meaning that only single words or tokens are considered as terms to encode.

See TF-IDF encoding of text features in Neptune ML.

The datetime_parts field

This field is used optionally by `datetime` features to specify which parts of the datetime value to encode categorically:

```
"datetime_parts": ["weekday", "hour"]
```

If you don't include `datetime_parts`, by default Neptune ML encodes the year, month, weekday and hour parts of the datetime value. The value `["weekday", "hour"]` indicates that only the weekday and hour of datetime values should be encoded categorically in the feature.

If one of the parts does not have more than one unique value in the training set, it is not encoded.

See Datetime features in Neptune ML.

Examples of using parameters within additionalParams for tuning model-training configuration

The following examples demonstrate how to utilize the "additionalParams" feature in property-graph and RDF data models to configure various aspects of the model training process for a Neptune ML application. The examples cover a wide range of functionality, including specifying default split rates for training/validation/test data, defining node classification, regression, and link prediction tasks, as well as configuring different feature types such as numerical buckets, text embeddings, datetime, and categorical data. These detailed configurations allow you to tailor the machine learning pipeline to your specific data and modeling requirements, unlocking the full potential of the Neptune ML capabilities.

Contents

- Property-graph examples using additionalParams
  - Specifying a default split rate for model-training configuration
  - Specifying a node-classification task for model-training configuration
  - Specifying a multi-class node classification task for model-training configuration
  - Specifying a node regression task for model-training configuration
  - Specifying an edge-classification task for model-training configuration
  - Specifying a multi-class edge classification task for model-training configuration
  - Specifying an edge regression for model-training configuration
  - Specifying a link prediction task for model-training configuration
  - Specifying a numerical bucket feature
  - Specifying a Word2Vec feature
  - Specifying a FastText feature
  - Specifying a Sentence BERT feature
  - Specifying a TF-IDF feature
  - Specifying a datetime feature
  - Specifying a category feature
  - Specifying a numerical feature
  - Specifying an auto feature
- RDF examples using additionalParams
  - Specifying a default split rate for model-training configuration

- Specifying a node-classification task for model-training configuration
- Specifying a node regression task for model-training configuration
- Specifying a link prediction task for particular edges
- Specifying a link prediction task for all edges

Property-graph examples using additionalParams

Specifying a default split rate for model-training configuration

In the following example, the `split_rate` parameter sets the default split rate for model training. If no default split rate is specified, the training uses a value of [0.9, 0.1, 0.0]. You can override the default value on a per-target basis by specifying a `split_rate` for each target.

In the following example, the `default split_rate` field indicates that a split rate of `[0.7,0.1,0.2]` should be used unless overridden on a per-target basis:"

```
  "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "split_rate": [0.7,0.1,0.2],
    "targets": [
      (...)
    ],
    "features": [
      (...)
    ]
  }
}
```

Specifying a node-classification task for model-training configuration

To indicate which node property contains labeled examples for training purposes, add a node classification element to the `targets` array, using `"type" : "classification"`. Add a `split_rate` field if you want to override the default split rate.

In the following example, the node target indicates that the `genre` property of each `Movie` node should be treated as a node class label. The `split_rate` value overrides the default split rate:

```
  "additionalParams": {
  "neptune_ml": {
```

```
    "version": "v2.0",
    "targets": [
      {
        "node": "Movie",
        "property": "genre",
        "type": "classification",
        "split_rate": [0.7,0.1,0.2]
      }
    ],
    "features": [
      (...)
    ]
  }
}
```

Specifying a multi-class node classification task for model-training configuration

To indicate which node property contains multiple labeled examples for training purposes, add a node classification element to the targets array, using `"type" : "classification"`, and `separator` to specify a character that can be used to split a target property value into multiple categorical values. Add a `split_rate` field if you want to override the default split rate.

In the following example, the `node` target indicates that the `genre` property of each `Movie` node should be treated as a node class label. The `separator` field indicates that each genre property contains multiple semicolon-separated values:

```
  "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      {
        "node": "Movie",
        "property": "genre",
        "type": "classification",
        "separator": ";"
      }
    ],
    "features": [
      (...)
    ]
  }
}
```

Specifying a node regression task for model-training configuration

To indicate which node property contains labeled regressions for training purposes, add a node regression element to the targets array, using `"type" : "regression"`. Add a split_rate field if you want to override the default split rate.

The following node target indicates that the `rating` property of each `Movie` node should be treated as a node regression label:

```
    "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      {
        "node": "Movie",
        "property": "rating",
        "type" : "regression",
        "split_rate": [0.7,0.1,0.2]
      }
    ],
    "features": [
      ...
    ]
  }
}
```

Specifying an edge-classification task for model-training configuration

To indicate which edge property contains labeled examples for training purposes, add an edge element to the `targets` array, using `"type" : "regression"`. Add a split_rate field if you want to override the default split rate.

The following edge target indicates that the `metAtLocation` property of each `knows` edge should be treated as an edge class label:

```
  "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      {
        "edge": ["Person", "knows", "Person"],
        "property": "metAtLocation",
```

```
        "type": "classification"
      }
    ],
    "features": [
      (...)
    ]
  }
}
```

Specifying a multi-class edge classification task for model-training configuration

To indicate which edge property contains multiple labeled examples for training purposes, add an edge element to the `targets` array, using `"type" : "classification"`, and a `separator` field to specify a character used to split a target property value into multiple categorical values. Add a `split_rate` field if you want to override the default split rate.

The following edge target indicates that the `sentiment` property of each `repliedTo` edge should be treated as an edge class label. The separator field indicates that each sentiment property contains multile comma-separated values:

```
  "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      {
        "edge": ["Person", "repliedTo", "Message"],
        "property": "sentiment",
        "type": "classification",
        "separator": ","
      }
    ],
    "features": [
      (...)
    ]
  }
}
```

Specifying an edge regression for model-training configuration

To indicate which edge property contains labeled regression examples for training purposes, add an edge element to the `targets` array, using `"type" : "regression"`. Add a `split_rate` field if you want to override the default split rate.

The following edge target indicates that the `rating` property of each `reviewed` edge should be treated as an edge regression:

```
  "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      {
        "edge": ["Person", "reviewed", "Movie"],
        "property": "rating",
        "type" : "regression"
      }
    ],
    "features": [
      (...)
    ]
  }
}
```

Specifying a link prediction task for model-training configuration

To indicate which edges should be used for link prediction training purposes, add an edge element to the targets array using `"type" : "link_prediction"`. Add a `split_rate` field if you want to override the default split rate.

The following edge target indicates that `cites` edges should be used for link prediction:

```
  "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      {
        "edge": ["Article", "cites", "Article"],
        "type" : "link_prediction"
      }
    ],
    "features": [
      (...)
    ]
  }
}
```

Specifying a numerical bucket feature

You can specify a numerical data feature for a node property by adding `"type": "bucket_numerical"` to the `features` array.

The following node feature indicates that the `age` property of each `Person` node should be treated as a numerical bucket feature:

```
  "additionalParams": {
  "neptune_ml": {
    "targets": [
      ...
    ],
    "features": [
      {
        "node": "Person",
        "property": "age",
        "type": "bucket_numerical",
        "range": [1, 100],
        "bucket_cnt": 5,
        "slide_window_size": 3,
        "imputer": "median"
      }
    ]
  }
}
```

Specifying a `Word2Vec` feature

You can specify a `Word2Vec` feature for a node property by adding `"type": "text_word2vec"` to the `features` array.

The following node feature indicates that the `description` property of each `Movie` node should be treated as a `Word2Vec` feature:

```
"additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      ...
    ],
    "features": [
```

```
      {
        "node": "Movie",
        "property": "description",
        "type": "text_word2vec",
        "language": "en_core_web_lg"
      }
    ]
  }
}
```

Specifying a `FastText` feature

You can specify a `FastText` feature for a node property by adding `"type": "text_fasttext"` to the `features` array. The `language` field is required, and must specify one of the following languages codes:

- `en` (English)
- `zh` (Chinese)
- `hi` (Hindi)
- `es` (Spanish)
- `fr` (French)

Note that the `text_fasttext` encoding cannot handle more than one language at a time in a feature.

The following node feature indicates that the French `description` property of each `Movie` node should be treated as a `FastText` feature:

```
"additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      ...
    ],
    "features": [
      {
        "node": "Movie",
        "property": "description",
        "type": "text_fasttext",
        "language": "fr",
```

```
        "max_length": 1024
      }
    ]
  }
}
```

Specifying a `Sentence BERT` feature

You can specify a `Sentence BERT` feature for a node property by adding `"type": "text_sbert"` to the `features` array. You don't need to specify the language, since the method automatically encodes text features using a multilingual language model.

The following node feature indicates that the `description` property of each `Movie` node should be treated as a `Sentence BERT` feature:

```
"additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      ...
    ],
    "features": [
      {
        "node": "Movie",
        "property": "description",
        "type": "text_sbert128",
      }
    ]
  }
}
```

Specifying a `TF-IDF` feature

You can specify a `TF-IDF` feature for a node property by adding `"type": "text_tfidf"` to the `features` array.

The following node feature indicates that the `bio` property of each `Person` node should be treated as a `TF-IDF` feature:

```
  "additionalParams": {
  "neptune_ml": {
```

```
    "version": "v2.0",
    "targets": [
      ...
    ],
    "features": [
      {
        "node": "Movie",
        "property": "bio",
        "type": "text_tfidf",
        "ngram_range": [1, 2],
        "min_df": 5,
        "max_features": 1000
      }
    ]
  }
}
```

Specifying a `datetime` feature

The export process automatically infers `datetime` features for date properties. However, if you want to limit the `datetime_parts` used for a `datetime` feature, or override a feature specification so that a property that would normally be treated as an `auto` feature is explicitly treated as a `datetime` feature, you can do so by adding a `"type": "datetime"` to the features array.

The following node feature indicates that the `createdAt` property of each `Post` node should be treated as a `datetime` feature:

```
  "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      ...
    ],
    "features": [
      {
        "node": "Post",
        "property": "createdAt",
        "type": "datetime",
        "datetime_parts": ["month", "weekday", "hour"]
      }
    ]
```

```
  }
}
```

Specifying a `category` feature

The export process automatically infers `auto` features for string properties and numeric properties containing multiples values. For numeric properties containing single values, it infers `numerical` features. For date properties it infers `datetime` features.

If you want to override a feature specification so that a property is treated as a categorical feature, add a `"type": "category"` to the features array. If the property contains multiple values, include a `separator` field. For example:

```
  "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      ...
    ],
    "features": [
      {
        "node": "Post",
        "property": "tag",
        "type": "category",
        "separator": "|"
      }
    ]
  }
}
```

Specifying a `numerical` feature

The export process automatically infers `auto` features for string properties and numeric properties containing multiples values. For numeric properties containing single values, it infers `numerical` features. For date properties it infers `datetime` features.

If you want to override a feature specification so that a property is treated as a `numerical` feature, add `"type": "numerical"` to the features array. If the property contains multiple values, include a `separator` field. For example:

```
  "additionalParams": {
```

```
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      ...
    ],
    "features": [
      {
        "node": "Recording",
        "property": "duration",
        "type": "numerical",
        "separator": ","
      }
    ]
  }
}
```

Specifying an `auto` feature

The export process automatically infers `auto` features for string properties and numeric properties containing multiples values. For numeric properties containing single values, it infers `numerical` features. For date properties it infers `datetime` features.

If you want to override a feature specification so that a property is treated as an `auto` feature, add `"type": "auto"` to the features array. If the property contains multiple values, include a `separator` field. For example:

```
  "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      ...
    ],
    "features": [
      {
        "node": "User",
        "property": "role",
        "type": "auto",
        "separator": ","
      }
    ]
  }
}
```

RDF examples using `additionalParams`

Specifying a default split rate for model-training configuration

In the following example, the `split_rate` parameter sets the default split rate for model training. If no default split rate is specified, the training uses a value of [0.9, 0.1, 0.0]. You can override the default value on a per-target basis by specifying a `split_rate` for each target.

In the following example, the `default split_rate` field indicates that a split rate of `[0.7,0.1,0.2]` should be used unless overridden on a per-target basis:"

```
  "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "split_rate": [0.7,0.1,0.2],
    "targets": [
      (...)
    ]
  }
}
```

Specifying a node-classification task for model-training configuration

To indicate which node property contains labeled examples for training purposes, add a node classification element to the `targets` array, using `"type" : "classification"`. Add a node field to indicate the node type of target nodes. Add a `predicate` field to define which literal data is used as the target node feature of the target node. Add a `split_rate` field if you want to override the default split rate.

In the following example, the node target indicates that the `genre` property of each `Movie` node should be treated as a node class label. The `split_rate` value overrides the default split rate:

```
  "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      {
        "node": "http://aws.amazon.com/neptune/csv2rdf/class/Movie",
        "predicate": "http://aws.amazon.com/neptune/csv2rdf/datatypeProperty/genre",
        "type": "classification",
        "split_rate": [0.7,0.1,0.2]
```

```
      }
    ]
  }
}
```

Specifying a node regression task for model-training configuration

To indicate which node property contains labeled regressions for training purposes, add a node regression element to the targets array, using `"type" : "regression"`. Add a `node` field to indicate the node type of target nodes. Add a `predicate` field to define which literal data is used as the target node feature of the target node. Add a `split_rate` field if you want to override the default split rate.

The following `node` target indicates that the `rating` property of each `Movie` node should be treated as a node regression label:

```
    "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      {
        "node": "http://aws.amazon.com/neptune/csv2rdf/class/Movie",
        "predicate": "http://aws.amazon.com/neptune/csv2rdf/datatypeProperty/rating",
        "type": "regression",
        "split_rate": [0.7,0.1,0.2]
      }
    ]
  }
}
```

Specifying a link prediction task for particular edges

To indicate which edges should be used for link prediction training purposes, add an edge element to the targets array using `"type" : "link_prediction"`. Add `subject`, `predicate` and `object` fields to specify the edge type. Add a `split_rate` field if you want to override the default split rate.

The following `edge` target indicates that `directed` edges that connect `Directors` to `Movies` should be used for link prediction:

```
  "additionalParams": {
```

```
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      {
        "subject": "http://aws.amazon.com/neptune/csv2rdf/class/Director",
        "predicate": "http://aws.amazon.com/neptune/csv2rdf/datatypeProperty/directed",
        "object": "http://aws.amazon.com/neptune/csv2rdf/class/Movie",
        "type" : "link_prediction"
      }
    ]
  }
}
```

Specifying a link prediction task for all edges

To indicate that all edges should be used for link prediction training purposes, add an edge element to the targets array using `"type" : "link_prediction"`. Do not add `subject`, `predicate`, or `object` fields. Add a `split_rate` field if you want to override the default split rate.

```
  "additionalParams": {
  "neptune_ml": {
    "version": "v2.0",
    "targets": [
      {
        "type" : "link_prediction"
      }
    ]
  }
}
```

Processing the graph data exported from Neptune for training

The data-processing step takes the Neptune graph data created by the export process and creates the information that is used by the Deep Graph Library (DGL) during training. This includes performing various data mappings and transformations:

- Parsing nodes and edges to construct the graph- and ID-mapping files required by DGL.
- Converting node and edge properties into the node and edge features required by DGL.
- Splitting the data into training, validation, and test sets.

Managing the data-processing step for Neptune ML

After you have exported the data from Neptune that you want to use for model training, you can start a data-processing job using a command like the following:

AWS CLI

```
aws neptunedata start-ml-data-processing-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --input-data-s3-location "s3:////(S3 bucket name)/(path to your input folder)" \
  --id "(a job ID for the new job)" \
  --processed-data-s3-location "s3://(S3 bucket name)/(path to your output folder)"
 \
  --config-file-name "training-job-configuration.json"
```

For more information, see start-ml-data-processing-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_data_processing_job(
    inputDataS3Location='s3://(S3 bucket name)/(path to your input folder)',
```

```
    id='(a job ID for the new job)',
    processedDataS3Location='s3:///(S3 bucket name)/(path to your output folder)',
    configFileName='training-job-configuration.json'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/dataprocessing \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "inputDataS3Location" : "s3:///(S3 bucket name)/(path to your input folder)",
        "id" : "(a job ID for the new job)",
        "processedDataS3Location" : "s3:///(S3 bucket name)/(path to your output
 folder)",
        "configFileName" : "training-job-configuration.json"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/dataprocessing \
  -H 'Content-Type: application/json' \
  -d '{
        "inputDataS3Location" : "s3:///(S3 bucket name)/(path to your input folder)",
        "id" : "(a job ID for the new job)",
        "processedDataS3Location" : "s3:///(S3 bucket name)/(path to your output
 folder)",
        "configFileName" : "training-job-configuration.json"
      }'
```

The details of how to use this command are explained in The dataprocessing command, along with information about how to get the status of a running job, how to stop a running job, and how to list all running jobs.

Processing updated graph data for Neptune ML

You can also supply a `previousDataProcessingJobId` to the API to ensure that the new data processing job uses the same processing method as a previous job. This is required when you want to get predictions for updated graph data in Neptune, either by retraining the old model on the new data, or by recomputing the model artifacts on the new data.

You do this by using a command like this:

AWS CLI

```
aws neptunedata start-ml-data-processing-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --input-data-s3-location "s3://(Amazon S3 bucket name)/(path to your input
 folder)" \
  --id "(a job ID for the new job)" \
  --processed-data-s3-location "s3://(Amazon S3 bucket name)/(path to your output
 folder)" \
  --previous-data-processing-job-id "(the job ID of the previous data-processing
 job)"
```

For more information, see start-ml-data-processing-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_data_processing_job(
    inputDataS3Location='s3://(Amazon S3 bucket name)/(path to your input folder)',
    id='(a job ID for the new job)',
    processedDataS3Location='s3://(Amazon S3 bucket name)/(path to your output
 folder)',
```

```
    previousDataProcessingJobId='(the job ID of the previous data-processing job)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/dataprocessing \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "inputDataS3Location" : "s3:////(Amazon S3 bucket name)/(path to your input
 folder)",
        "id" : "(a job ID for the new job)",
        "processedDataS3Location" : "s3://(Amazon S3 bucket name)/(path to your
 output folder)",
        "previousDataProcessingJobId" : "(the job ID of the previous data-processing
 job)"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/dataprocessing \
  -H 'Content-Type: application/json' \
  -d '{
        "inputDataS3Location" : "s3://(Amazon S3 bucket name)/(path to your input
 folder)",
        "id" : "(a job ID for the new job)",
        "processedDataS3Location" : "s3://(Amazon S3 bucket name)/(path to your
 output folder)",
        "previousDataProcessingJobId" : "(the job ID of the previous data-processing
 job)"
```

```
    }'
```

Set the value of the `previousDataProcessingJobId` parameter to the job ID of the previous-data processing job that corresponds to the trained model.

> **Note**
>
> Node deletions in the updated graph are currently not supported. If nodes have been removed in an updated graph, you have to start a completely new data processing job rather than use `previousDataProcessingJobId`.

Feature encoding in Neptune ML

Property values come in different formats and data types. To achieve good performance in machine learning, it is essential to convert those values to numerical encodings known as *features*.

Neptune ML performs feature extraction and encoding as part of the data-export and data-processing steps, using feature-encoding techniques described here.

> **Note**
>
> If you plan to implement your own feature encoding in a custom model implementation, you can disable the automatic feature encoding in the data preprocessing stage by selecting none as the feature encoding type. No feature encoding then occurs on that node or edge property, and instead the raw property values are parsed and saved in a dictionary. Data preprocessing still creates the DGL graph from the exported dataset, but the constructed DGL graph doesn't have the pre-processed features for training.
> You should use this option only if you plan to perform your custom feature encoding as part of custom model training. See Custom models in Neptune ML for details.

Categorical features in Neptune ML

A property that can take one or more distinct values from a fixed list of possible values is a categorical feature. In Neptune ML, categorical features are encoded using one-hot encoding. The following example shows how the property name of different foods is one-hot encoded according to its category:

```
 Food        Veg.   Meat   Fruit    Encoding
---------    ----   ----   -----    --------
 Apple         0      0      1         001
 Chicken       0      1      0         010
 Broccoli      1      0      0         100
```

> **Note**
>
> The maximum number of categories in any categorical feature is 100. If a property has more than 100 categories of value, only the most common 99 of them are placed in distinct categories, and the rest are placed in a special category named OTHER.

Numerical features in Neptune ML

Any property whose values are real numbers can be encoded as a numerical feature in Neptune ML. Numerical features are encoded using floating-point numbers.

You can specify a data-normalization method to use when encoding numerical features, like this: `"norm": "normalization technique"`. The following normalization techniques are supported:

- **"none"** – Don't normalize the numerical values during encoding.
- **"min-max"** – Normalize each value by subtracting the minimum value from it and then dividing it by the difference between the maximum value and the minimum.
- **"standard"** – Normalize each value by dividing it by the sum of all the values.

Bucket-numerical features in Neptune ML

Rather than representing a numerical property using raw numbers, you can condense numerical values into categories. For example, you could divide people's ages into categories such as kids (0-20), young adults (20-40), middle-aged people (40-60) and elders (from 60 on). Using these numerical buckets, you would be transforming a numerical property into a kind of categorical feature.

In Neptune ML, you can cause a numerical property to be encoded as a bucket-numerical feature, you must provide two things:

- A numerical range in the form, `"range": [a, b]` , where a and b are integers.
- A bucket count, in the form `"bucket_cnt": c` , where c is the number of buckets, also an integer.

Neptune ML then calculates the size of each bucket as `( b - a ) / c` , and encodes each numeric value as the number of whatever bucket it falls into. Any value less than a is considered to belong in the first bucket, and any value greater than b is considered to belong in the last bucket.

You can also, optionally, make numeric values fall into more than one bucket, by specifying a slide-window size, like this: `"slide_window_size": s` , where s is a number. Neptune ML then transforms each numeric value v of the property into a range from `v - s/2` through `v + s/2` , and assigns the value v to every bucket that the range covers.

Finally, you can also optionally provide a way of filling in missing values for numerical features and bucket-numerical features. You do this using `"imputer": "imputation technique "`, where the imputation technique is one of `"mean"`, `"median"`, or `"most-frequent"`. If you don't specify an imputer, a missing value can cause processing to halt.

Text feature encoding in Neptune ML

For free-form text, Neptune ML can use several different models to convert the sequence of tokens in a property value string into a fixed-size real-value vector:

- `text_fasttext` – Uses fastText encoding. This is the recommended encoding for features that use one and only one of the five languages that fastText supports.
- `text_sbert` – Uses the Sentence BERT (SBERT) encoding models. This is the recommended encoding for text that `text_fasttext` does not support.
- `text_word2vec` – Uses the Word2Vec algorithms originally published by Google to encode text. Word2Vec only supports English.
- `text_tfidf` – Uses a term frequency–inverse document frequency (TF-IDF) vectorizer for encoding text. TF-IDF encoding supports statistical features that the other encodings do not.

fastText encoding of text property values in Neptune ML

Neptune ML can use the fastText models to convert text property values into fixed-size real-value vectors. This is the recommended encoding method for text property values in any one of the five languages that fastText supports:

- `en` (English)
- `zh` (Chinese)
- `hi` (Hindi)

- `es` (Spanish)
- `fr` (French)

Note that fastText cannot handle sentences containing words in more than one language.

The `text_fasttext` method can optionally take `max_length` field that specifies the maximum number of tokens in a text property value that will be encoded, after which the string is truncated. This can improve performance when text property values contain long strings, because if `max_length` is not specified, fastText encodes all the tokens regardless of the string length.

This example specifies that French movie titles are encoded using fastText:

```
{
    "file_name" : "nodes/movie.csv",
    "separator" : ",",
    "node" : ["~id", "movie"],
    "features" : [
      {
        "feature": ["title", "title", "text_fasttext"],
        "language": "fr",
        "max_length": 1024
      }
    ]
  }
```

Sentence BERT (SBERT) encoding of text features in Neptune ML

Neptune ML can convert the sequence of tokens in a string property value into a fixed-size real-value vector using Sentence BERT (SBERT) models. Neptune supports two SBERT methods: `text_sbert128`, which is the default if you just specify `text_sbert`, and `text_sbert512`. The difference between the two is the maximum length of a text property value string that is encoded. The `text_sbert128` encoding truncates text strings after encoding 128 tokens, while `text_sbert512` truncates text strings after encoding 512 tokens. As a result, `text_sbert512` requires more processing time than `text_sbert128`. Both methods are slower than `text_fasttext`.

SBERT encoding is multilingual, so there is no need to specify a language for the property value text you are encoding. SBERT supports many languages, and can encode a sentence that contains more than one language. If you are encoding property values containing text in a language or languages that fastText does not support, SBERT is the recommended encoding method.

The following example specifies that movie titles are encoded as SBERT up to a maximum of 128 tokens:

```
{
    "file_name" : "nodes/movie.csv",
    "separator" : ",",
    "node" : ["~id", "movie"],
    "features" : [
      { "feature": ["title", "title", "text_sbert128"] }
    ]
  }
```

Word2Vec encoding of text features in Neptune ML

Neptune ML can encode string property values as a Word2Vec feature (Word2Vec algorithms were originally published by Google). The `text_word2vec` method encodes the tokens in a string as a dense vector using one of the spaCy trained models. This only supports the English language using the en_core_web_lg model).

The following example specifies that movie titles are encoded using Word2Vec:

```
{
    "file_name" : "nodes/movie.csv",
    "separator" : ",",
    "node" : ["~id", "movie"],
    "features" : [
      {
        "feature": ["title", "title", "text_word2vec"],
        "language": "en_core_web_lg"
      }
    ]
  }
```

Note that the language field is optional, since the English `en_core_web_lg` model is the only one that Neptune supports.

TF-IDF encoding of text features in Neptune ML

Neptune ML can encode text property values as `text_tfidf` features. This encoding converts the sequence of words in the text into a numeric vector using a term frequency–inverse document frequency (TF-IDF) vectorizer, followed by a dimensionality-reduction operation.

TF-IDF (term frequency – inverse document frequency) is a numerical value intended to measure how important a word is in a document set. It is calculated by dividing the number of times a word appears in a given property value by the total number of such property values that it appears in.

For example, if the word "kiss" appears twice in a given movie title (say, "kiss kiss bang bang"), and "kiss" appears in the title of 4 movies in all, then the TF-IDF value of "kiss" in the "kiss kiss bang bang" title would be `2 / 4` .

The vector that is initially created has ***d*** dimensions, where ***d*** is the number of unique terms in all property values of that type. The dimensionality-reduction operation uses a random sparse projection to reduce that number to a maximum of 100. The vocabulary of a graph is then generated by merging all the `text_tfidf` features in it.

You can control the TF-IDF vectorizer in several ways:

- **`max_features`** – Using the `max_features` parameter, you can limit the number of terms in `text_tfidf` features to the most common ones. For example, if you set `max_features` to 100, only the top 100 most commonly used terms are included. The default value for `max_features` if you don't explicitly set it is 5,000.
- **`min_df`** – Using the `min_df` parameter, you can limit the number of terms in `text_tfidf` features to ones having at least a specified document frequency. For example, if you set `min_df` to 5, only terms that appear in at least 5 different property values are used. The default value for `min_df` if you don't explicitly set it is 2.
- **`ngram_range`** – The `ngram_range` parameter determines what combinations of words are treated as terms. For example, if you set `ngram_range` to `[2, 4]`, the following 6 terms would be found in the "kiss kiss bang bang" title:
  - *2-word terms*: "kiss kiss", "kiss bang", and "bang bang".
  - *3-word terms*: "kiss kiss bang" and "kiss bang bang".
  - *4-word terms*: "kiss kiss bang bang".

  The default setting for `ngram_range` is `[1, 1]`.

Datetime features in Neptune ML

Neptune ML can convert parts of `datetime` property values into categorical features by encoding them as one-hot arrays. Use the `datetime_parts` parameter to specify one or more of the

following parts to encode: `["year", "month", "weekday", "hour"]`. If you don't set `datetime_parts`, by default all four parts are encoded.

For example, if the range of datetime values spans the years 2010 through 2012, the four parts of the datetime entry `2011-04-22 01:16:34` are as follows:

- **year** – `[0, 1, 0]`.

  Since there are only 3 years in the span (2010, 2011, and 2012), the one-hot array has three entries, one for each year.
- **month** – `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]`.

  Here, the one-hot array has an entry for each month of the year.
- **weekday** – `[0, 0, 0, 0, 1, 0, 0]`.

  The ISO 8601 standard states that Monday is the first day of the week, and since April 22, 2011 was a Friday, the corresponding one-hot weekday array is hot in the fifth position.
- **hour** – `[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`.

  The hour 1 AM is set in a 24-member one-hot array.

Day of the month, minute, and second are not encoded categorically.

If the total `datetime` range in question only includes dates within a single year, no `year` array is encoded.

You can specify an imputation strategy to fill in missing `datetime` values, using the `imputer` parameter and one of the strategies available for numerical features.

Auto feature encoding in Neptune ML

Instead of manually specifying the feature encoding methods to use for the properties in your graph, you can set `auto` as a feature encoding method. Neptune ML then attempts to infer the best feature encoding for each property based on its underlying data type.

Here are some of the heuristics that Neptune ML uses in selecting the appropriate feature encodings:

- If the property has only numeric values and can be cast into numeric data types, then Neptune ML generally encodes it as a numeric value. However, if the number of unique values for the property is less than 10% of the total number of values and the cardinality of those unique values is less than 100, then Neptune ML uses a categorical encoding.
- If the property values can be cast to a `datetime` type, then Neptune ML encodes them as a `datetime` feature.
- If the property values can be coerced to booleans (1/0 or True/False), then Neptune ML uses category encoding.
- If the property is a string with more than 10% of its values unique, and the average number of tokens per value is greater than or equal to 3, the Neptune ML infers the property type to be text and automatically detects the language being used. If the language detected is one of the ones supported by fastText, namely English, Chinese, Hindi, Spanish and French, then Neptune ML uses `text_fasttext` to encode the text. Othewise, Neptune ML uses `text_sbert`.
- If the property is a string not classified as a text feature then Neptune ML presumes it to be a categorical feature and uses category encoding.
- If each node has its own unique value for a property that is inferred to be a category feature, Neptune ML drops the property from the training graph because it is probably an ID that would not be informative for learning.
- If the property is known to contain valid Neptune separators such as semicolons (";"), then Neptune ML can only treat the property as `MultiNumerical` or `MultiCategorical`.
  - Neptune ML first tries to encode the values as numeric features. if this succeeds, Neptune ML uses numerical encoding to create numeric vector features.
  - Otherwise, Neptune ML encodes the values as multi-categorical.
- If Neptune ML cannot infer the data type of a property's values, Neptune MLdrops the property from the training graph.

Editing a training data configuration file

The Neptune export process exports Neptune ML data from a Neptune DB cluster into an S3 bucket. It exports nodes and edges separately into a `nodes/` and an `edges/` folder. It also creates a JSON training data configuration file, named `training-data-configuration.json` by default. This file contains information about the schema of the graph, the types of its features, feature transformation and normalization operations, and the target feature for a classification or regression task.

There might be cases when you want to modify the configuration file directly. One such case is when you want to change the way features are processed or how the graph is constructed, without needing to rerun the export every time you want to modify the specification for the machine learning task you're solving.

To edit the training data configuration file

1. **Download the file to your local machine.**

   Unless you specified one or more named jobs in the `additionalParams/neptune_ml` parameter passed to the export process, the file will have the default name, which is `training-data-configuration.json`. You can use an AWS CLI command like this to download the file:

   ```
   aws s3 cp \
     s3:////(your Amazon S3 bucket)/(path to your export folder)/training-data-
   configuration.json \
     ./
   ```

2. **Edit the file using a text editor.**

3. **Upload the modified file.** Upload the modified file back to the same location in Amazon S3 from which you downloaded it, using use an AWS CLI command like this:

   ```
   aws s3 cp \
     training-data-configuration.json \
     s3://(your Amazon S3 bucket)/(path to your export folder)/training-data-
   configuration.json
   ```

Example of a JSON training data configuration file

Here is a sample training data configuration file that describes a graph for a node-classification task:

```
{
  "version" : "v2.0",
  "query_engine" : "gremlin",
  "graph" : [
    {
      "edges" : [
        {
          "file_name" : "edges/(movie)-included_in-(genre).csv",
          "separator" : ",",
          "source" : ["~from", "movie"],
          "relation" : ["", "included_in"],
          "dest" : [ "~to", "genre" ]
        },
        {
          "file_name" : "edges/(user)-rated-(movie).csv",
          "separator" : ",",
          "source" : ["~from", "movie"],
          "relation" : ["rating", "prefixname"], # [prefixname#value]
          "dest" : ["~to", "genre"],
          "features" : [
            {
              "feature" : ["rating", "rating", "numerical"],
              "norm" : "min-max"
            }
          ]
        }
      ],
      "nodes" : [
        {
          "file_name" : "nodes/genre.csv",
          "separator" : ",",
          "node" : ["~id", "genre"],
          "features" : [
            {
              "feature": ["name", "genre", "category"],
              "separator": ";"
            }
          ]
```

```
        },
        {
          "file_name" : "nodes/movie.csv",
          "separator" : ",",
          "node" : ["~id", "movie"],
          "features" : [
            {
              "feature": ["title", "title", "word2vec"],
              "language": ["en_core_web_lg"]
            }
          ]
        },
        {
          "file_name" : "nodes/user.csv",
          "separator" : ",",
          "node" : ["~id", "user"],
          "features" : [
            {
              "feature": ["age", "age", "numerical"],
              "norm" : "min-max",
              "imputation": "median",
            },
            {
              "feature": ["occupation", "occupation", "category"],
            }
          ],
          "labels" : [
            {
              "label": ["gender", "classification"],
              "split_rate" : [0.8, 0.2, 0.0]
            }
          ]
        }
      ]
    },
    "warnings" : [ ]
  ]
}
```

The structure of JSON training data configuration files

The training configuration file refers to CSV files saved by the export process in the `nodes/` and `edges/` folders.

Each file under `nodes/` stores information about nodes that have the same property-graph node label. Each column in a node file stores either the node ID or the node property. The first line of the file contains a header that specifies the `~id` or property name for each column.

Each file under `edges/` stores information about nodes that have the same property-graph edge label. Each column in a node file stores either the source node ID, the destination node ID, or the edge property. The first line of the file contains a header specifying the `~from`, `~to`, or property name for each column.

The training data configuration file has three top-level elements:

```
{
  "version" : "v2.0",
  "query_engine" : "gremlin",
  "graph" : [ ... ]
}
```

- `version` – (String) The version of configuration file being used.
- `query_engine` – (String) The query language used for exporting the graph data. Currently, only "gremlin" is valid.
- `graph` – (JSON array) lists one or more configuration objects that contain model parameters for each of the nodes and edges that will be used.

  The configuration objects in the graph array have the structure described in the next section.

Contents of a configuration object listed in the `graph` array

A configuration object in the `graph` array can contain three top-level nodes:

```
    {
      "edges"    : [ ... ],
      "nodes"    : [ ... ],
      "warnings" : [ ... ],
    }
```

- `edges` – (array of JSON objects) Each JSON object specifies a set of parameters to define how an edge in the graph will be treated during the model processing and training. This is only used with the Gremlin engine.

- `nodes` – (array of JSON objects) Each JSON object specifies a set of parameters to define how a node in the graph will be treated during the model processing and training. This is only used with the Gremlin engine.
- `warnings` – (array of JSON objects) Each object contains a warning generated during the data export process.

Contents of an edge configuration object listed in an `edges` array

An edge configuration object listed in an `edges` array can contain the following top-level fields:

```
    {
      "file_name" : "(path to a CSV file)",
      "separator" : "(separator character)",
      "source"    : ["(column label for starting node ID)", "(starting node type)"],
      "relation"  : ["(column label for the relationship name)", "(the prefix name
for the relationship name)"],
      "dest"      : ["(column label for ending node ID)", "(ending node type)"],
      "features"  : [(array of feature objects)],
      "labels"    : [(array of label objects)]
    }
```

- **`file_name`** – A string specifying the path to a CSV file that stores information about edges having the same property-graph label.

  The first line of that file contains a header line of column labels.

  The first two column labels are `~from` and `~to`. The first column (the `~from` column) stores the ID of the edge's starting node, and the second (the `~to` column) stores the ID of the edge's ending node.

  The remaining column labels in the header line specify, for each remaining column, the name of the edge property whose values have been exported into that column.
- **`separator`** – A string containing the delimiter that separates columns in that CSV file.
- **`source`** – A JSON array containing two strings that specify the starting node of the edge. The first string contains the header name of the column that the starting node ID is stored in. The second string specifies the node type.

- **`relation`** – A JSON array containing two strings that specify the edge's relation type. The first string contains the header name of the column that the relation name (`relname`) is stored in. The second string contains the prefix for the relation name (`prefixname`).

  The full relation type consists of the two strings combined, with a hyphen character between them, like this: *`prefixname`*`-`*`relname`*.

  If the first string is empty, all edges have the same relation type, namely the `prefixname` string.
- **`dest`** – A JSON array containing two strings that specify the ending node of the edge. The first string contains the header name of the column that the node ID is stored in. The second string specifies the node type.
- **`features`** – A JSON array of property-value feature objects. Each property-value feature object contains the following fields:
  - **feature** – A JSON array of three strings. The first string contains the header name of the column that contains the property value. The second string contains the feature name. The third string contains the feature type.
  - **norm** – (*Optional*) Specifies a normalization method to apply to the property values.
- **`labels`** – A JSON array of objects. Each of the objects defines a target feature of the edges, and specifies the proportions of the edges that the training and validation stages should take. Each object contains the following fields:
  - **label** – A JSON array of two strings. The first string contains the header name of the column that contains the target feature property value. The second string specifies one of the following target task types:
    - `"classification"` – An edge classification task. The property values provided in the column identified by the first string in the `label` array are treated as categorical values. For an edge classification task, the first string in the `label` array can't be empty.
    - `"regression"` – An edge regression task. The property values provided in the column identified by the first string in the `label` array are treated as numerical values. For an edge regression task, the first string in the `label` array can't be empty.
    - `"link_prediction"` – A link prediction task. No property values are required. For a link prediction task, the first string in the `label` array is ignored.
  - **`split_rate`** – A JSON array containing three numbers between zero and one that add up to one and that represent an estimate of the proportions of nodes that the training, validation,

and test stages will use, respectively. Either this field or the `custom_split_filenames` can be defined, but not both. See split_rate.

- **`custom_split_filenames`** – A JSON object that specifies the file names for the files that define the training, validation and test populations. Either this field or `split_rate` can be defined, but not both. See Custom train-validation-test proportions for more information.

Contents of a node configuration object listed in a `nodes` array

A node configuration object listed in a `nodes` array can contain the following fields:

```
{
  "file_name" : "(path to a CSV file)",
  "separator" : "(separator character)",
  "node"      : ["(column label for the node ID)", "(node type)"],
  "features"  : [(feature array)],
  "labels"    : [(label array)],
}
```

- **`file_name`** – A string specifying the path to a CSV file that stores information about nodes having the same property-graph label.

  The first line of that file contains a header line of column labels.

  The first column label is `~id`, and the first column (the `~id` column) stores the node ID.

  The remaining column labels in the header line specify, for each remaining column, the name of the node property whose values have been exported into that column.
- **`separator`** – A string containing the delimiter that separates columns in that CSV file.
- **`node`** – A JSON array containing two strings. The first string contains the header name of the column that stores node IDs. The second string specifies the node type in the graph, which corresponds to a property-graph label of the node.
- **`features`** – A JSON array of node feature objects. See Contents of a feature object listed in a `features` array for a node or edge.
- **`labels`** – A JSON array of node label objects. See Contents of a node label object listed in a node `labels` array.

Contents of a feature object listed in a `features` array for a node or edge

A node feature object listed in a node `features` array can contain the following top-level fields:

- **`feature`** – A JSON array of three strings. The first string contains the header name of the column that contains the property value for the feature. The second string contains the feature name.

  The third string contains the feature type. Valid feature types are listed in Possible values of the type field for features.
- **`norm`** – This field is required for numerical features. It specifies a normalization method to use on numeric values. Valid values are `"none"`, `"min-max"`, and "standard". See The norm field for details.
- **`language`** – The language field specifies the language being used in text property values. Its usage depends on the text encoding method:
  - For `text_fasttext` encoding, this field is required, and must specify one of the following languages:
    - `en` (English)
    - `zh` (Chinese)
    - `hi` (Hindi)
    - `es` (Spanish)
    - `fr` (French)

    However, `text_fasttext` cannot handle more than one language at a time.
  - For `text_sbert` encoding, this field is not used, since SBERT encoding is multilingual.
  - For `text_word2vec` encoding, this field is optional, since `text_word2vec` only supports English. If present, it must specify the name of the English language model:

    ```
    "language" : "en_core_web_lg"
    ```
  - For `tfidf` encoding, this field is not used.
- **`max_length`** – This field is optional for `text_fasttext` features, where it specifies the maximum number of tokens in an input text feature that will be encoded. Input text after `max_length` is reached is ignored. For example, setting max_length to 128 indicates that any tokens after the 128th in a text sequence are ignored.

- **`separator`** – This field is used optionally with `category`, `numerical` and `auto` features. It specifies a character that can be used to split a property value into multiple categorical values or numerical values.

  See The separator field.
- **`range`** – This field is required for `bucket_numerical` features. It specifies the range of numerical values that are to be divided into buckets.

  See The range field.
- **`bucket_cnt`** – This field is required for `bucket_numerical` features. It specifies the number of buckets that the numerical range defined by the `range` parameter should be divided into.

  See Bucket-numerical features in Neptune ML.
- **`slide_window_size`** – This field is used optionally with `bucket_numerical` features to assign values to more than one bucket.

  See The slide_window_size field.
- **`imputer`** – This field is used optionally with `numerical`, `bucket_numerical`, and `datetime` features to provide an imputation technique for filling in missing values. The supported imputation techniques are `"mean"`, `"median"`, and `"most_frequent"`.

  See The imputer field.
- **`max_features`** – This field is used optionally by `text_tfidf` features to specify the maximum number of terms to encode.

  See The max_features field.
- **`min_df`** – This field is used optionally by `text_tfidf` features to specify the minimum document frequency of terms to encode

  See The min_df field.
- **`ngram_range`** – This field is used optionally by `text_tfidf` features to specify a range of numbers of words or tokens to considered as potential individual terms to encode

  See The ngram_range field.
- **`datetime_parts`** – This field is used optionally by `datetime` features to specify which parts of the datetime value to encode categorically.

  See The datetime_parts field.

Contents of a node label object listed in a node `labels` array

A label object listed in a node `labels` array defines a node target feature and specifies the proportions of nodes that the training, validation, and test stages will use. Each object can contain the following fields:

```
    {
      "label"      : ["(column label for the target feature property value)", "(task
type)"],
      "split_rate" : [(training proportion), (validation proportion), (test
proportion)],
      "custom_split_filenames" : {"train": "(training file name)", "valid":
"(validation file name)", "test": "(test file name)"},
      "separator"  : "(separator character for node-classification category values)",
    }
```

- **`label`** – A JSON array containing two strings. The first string contains the header name of the column that stores the property values for the feature. The second string specifies the target task type, which can be:
  - `"classification"` – A node classification task. The property values in the specified column are used to create a categorical feature.
  - `"regression"` – A node regression task. The property values in the specified column are used to create a numerical feature.
- **`split_rate`** – A JSON array containing three numbers between zero and one that add up to one and represent an estimate of the proportions of nodes that the training, validation, and test stages will use, respectively. See split_rate.
- **`custom_split_filenames`** – A JSON object that specifies the file names for the files that define the training, validation and test populations. Either this field or `split_rate` can be defined, but not both. See Custom train-validation-test proportions for more information.
- **`separator`** – A string containing the delimiter that separates categorical feature values for a classification task.

> **Note**
>
> If no label object is provided for both edges and nodes, the task is automatically assumed to be link prediction, and edges are randomly split into 90% for training and 10% for validation.

Custom train-validation-test proportions

By default, the `split_rate` parameter is used by Neptune ML to split the graph randomly into training, validation and test populations using the proportions defined in this parameter. To have more precise control over which entities are used in these different populations, files can be created that explicitly define them, and then the training data configuration file can be edited to map these indexing files to the populations. This mapping is specified by a JSON object for the `custom_split_filesnames` key in the training configuration file. If this option is used, filenames must be provided for the `train` and `validation` keys, and is optional for the `test` key.

The formatting of these files should match the Gremlin data format. Specifically, for node-level tasks, each file should contain a column with the `~id` header that lists the node IDs, and for edge-level tasks, the files should specify `~from` and `~to` to indicate the source and destination nodes of the edges, respectively. These files need to be placed in the same Amazon S3 location as the exported data that is used for data processing (see: `outputS3Path`).

For property classification or regression tasks, these files can optionally define the labels for the machine-learning task. In that case the files need to have a property column with the same header name as is defined in the training data configuration file. If property labels are defined in both the exported node and edge files and the custom-split files, priority is given to the custom-split files.

Training a model using Neptune ML

After you have processed the data that you exported from Neptune for model training, you can start a model-training job using a command like the following:

AWS CLI

```
aws neptunedata start-ml-model-training-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-training job ID)" \
  --data-processing-job-id "(the data-processing job-id of a completed job)" \
  --train-model-s3-location "s3://(your S3 bucket)/neptune-model-graph-autotrainer"
```

For more information, see start-ml-model-training-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_model_training_job(
    id='(a unique model-training job ID)',
    dataProcessingJobId='(the data-processing job-id of a completed job)',
    trainModelS3Location='s3://(your S3 bucket)/neptune-model-graph-autotrainer'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltraining \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
```

```
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3://(your S3 bucket)/neptune-model-graph-
autotrainer"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltraining \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3://(your S3 bucket)/neptune-model-graph-
autotrainer"
      }'
```

The details of how to use this command are explained in The modeltraining command, along with information about how to get the status of a running job, how to stop a running job, and how to list all running jobs.

You can also supply a `previousModelTrainingJobId` to use information from a completed Neptune ML model training job to accelerate the hyperparameter search in a new training job. This is useful during model retraining on new graph data, as well as incremental training on the same graph data. Use a command like this one:

AWS CLI

```
aws neptunedata start-ml-model-training-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-training job ID)" \
  --data-processing-job-id "(the data-processing job-id of a completed job)" \
  --train-model-s3-location "s3://(your S3 bucket)/neptune-model-graph-autotrainer"
 \
```

```
  --previous-model-training-job-id "(the model-training job-id of a completed job)"
```

For more information, see start-ml-model-training-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_model_training_job(
    id='(a unique model-training job ID)',
    dataProcessingJobId='(the data-processing job-id of a completed job)',
    trainModelS3Location='s3:////(your S3 bucket)/neptune-model-graph-autotrainer',
    previousModelTrainingJobId='(the model-training job-id of a completed job)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltraining \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3://(your S3 bucket)/neptune-model-graph-
autotrainer",
        "previousModelTrainingJobId" : "(the model-training job-id of a completed
 job)"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltraining \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3://(your S3 bucket)/neptune-model-graph-
autotrainer",
        "previousModelTrainingJobId" : "(the model-training job-id of a completed
 job)"
      }'
```

You can train your own model implementation on the Neptune ML training infrastructure by supplying a `customModelTrainingParameters` object, like this:

AWS CLI

```
aws neptunedata start-ml-model-training-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-training job ID)" \
  --data-processing-job-id "(the data-processing job-id of a completed job)" \
  --train-model-s3-location "s3://(your Amazon S3 bucket)/neptune-model-graph-
autotrainer" \
  --model-name "custom" \
  --custom-model-training-parameters '{
    "sourceS3DirectoryPath": "s3://(your Amazon S3 bucket)/(path to your Python
 module)",
    "trainingEntryPointScript": "(your training script entry-point name in the
 Python module)",
    "transformEntryPointScript": "(your transform script entry-point name in the
 Python module)"
  }'
```

For more information, see start-ml-model-training-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_model_training_job(
    id='(a unique model-training job ID)',
    dataProcessingJobId='(the data-processing job-id of a completed job)',
    trainModelS3Location='s3://(your Amazon S3 bucket)/neptune-model-graph-
autotrainer',
    modelName='custom',
    customModelTrainingParameters={
        'sourceS3DirectoryPath': 's3://(your Amazon S3 bucket)/(path to your Python
 module)',
        'trainingEntryPointScript': '(your training script entry-point name in the
 Python module)',
        'transformEntryPointScript': '(your transform script entry-point name in the
 Python module)'
    }
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltraining \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3://(your Amazon S3 bucket)/neptune-model-graph-
autotrainer",
        "modelName": "custom",
```

```
        "customModelTrainingParameters" : {
          "sourceS3DirectoryPath": "s3:////(your Amazon S3 bucket)/(path to your
 Python module)",
          "trainingEntryPointScript": "(your training script entry-point name in the
 Python module)",
          "transformEntryPointScript": "(your transform script entry-point name in
 the Python module)"
        }
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltraining \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3://(your Amazon S3 bucket)/neptune-model-graph-
autotrainer",
        "modelName": "custom",
        "customModelTrainingParameters" : {
          "sourceS3DirectoryPath": "s3://(your Amazon S3 bucket)/(path to your
 Python module)",
          "trainingEntryPointScript": "(your training script entry-point name in the
 Python module)",
          "transformEntryPointScript": "(your transform script entry-point name in
 the Python module)"
        }
      }'
```

See The modeltraining command for more information, such as about how to get the status of a running job, how to stop a running job, and how to list all running jobs. See Custom models in Neptune ML for information about how to implement and use a custom model.

Topics

- Models and model training in Amazon Neptune ML
- Customizing model hyperparameter configurations in Neptune ML
- Model training best practices

Models and model training in Amazon Neptune ML

Neptune ML uses Graph Neural Networks (GNN) to create models for the various machine-learning tasks. Graph neural networks have been shown to obtain state-of-the-art results for graph machine learning tasks and are excellent at extracting informative patterns from graph structured data.

Graph neural networks (GNNs) in Neptune ML

Graph Neural Networks (GNNs) belong to a family of neural networks that compute node representations by taking into account the structure and features of nearby nodes. GNNs complement other traditional machine learning and neural network methods that are not well-suited for graph data.

GNNs are used to solve machine-learning tasks such as node classification and regression (predicting properties of nodes) and edge classification and regression (predicting properties of edges) or link prediction (predicting whether two nodes in the graph should be connected or not).

In general, using a GNN for a machine learning task involves two stages:

- An encoding stage, where the GNN computes a d-dimensional vector for each node in the graph. These vectors are also called *representations* or *embeddings*.
- A decoding stage, which makes predictions based on the encoded representations.

For node classification and regression, the node representations are used directly for the classification and regression tasks. For edge classification and regression, the node representations of the incident nodes on an edge are used as input for the classification or regression. For link prediction, an edge likelihood score is computed by using a pair of node representations and an edge type representation.

The Deep Graph Library (DGL) facilitates the efficient definition and training of GNNs for these tasks.

Different GNN models are unified under the formulation of message passing. In this view, the representation for a node in a graph is calculated using the node's neighbors' representations (the messages), together with the node's initial representation. In NeptuneML, the initial representation of a node is derived from the features extracted from its node properties, or is learnable and depends on the identity of the node.

Neptune ML also provides the option to concatenate node features and learnable node representations to serve as the original node representation.

For the various tasks in Neptune ML involving graphs with node properties, we use the Relational Graph Convolutional Network (R-GCN)) to perform the encoding stage. R-GCN is a GNN architecture that is well-suited for graphs that have multiple node and edge types (these are known as heterogeneous graphs).

The R-GCN network consists of a fixed number of layers, stacked one after the other. Each layer of the R-GCN uses its learnable model parameters to aggregate information from the immediate, 1-hop neighborhood of a node. Since subsequent layers use the previous layer's output representations as input, the radius of the graph neighborhood that influences a node's final embedding depends on the number of layers (`num-layer`), of the R-GCN network.

For example, this means that a 2-layer network uses information from nodes that are 2 hops away.

To learn more about GNNs, see A Comprehensive Survey on Graph Neural Networks. For more information about the Deep Graph Library (DGL), visit the DGL webpage. For a hands-on tutorial about using DGL with GNNs, see Learning graph neural networks with Deep Graph Library.

Training Graph Neural Networks

In machine learning, the process of getting a model to learn how to make good predictions for a task is called model training. This is usually performed by specifying a particular objective to optimize, as well as an algorithm to use to perform this optimization.

This process is employed in training a GNN to learn good representations for the downstream task as well. We create an objective function for that task that is minimized during model training. For example, for node classification, we use CrossEntropyLoss as the objective, which penalizes misclassifications, and for node regression we minimize MeanSquareError.

The objective is usually a loss function that takes the model predictions for a particular data point and compares them to the ground-truth value for that data point. It returns the loss value, which shows how far off the model's predictions are. The goal of the training process is to minimize the loss and ensure that model predictions are close to the ground-truth.

The optimization algorithm used in deep learning for the training process is usually a variant of gradient descent. In Neptune ML, we use Adam, which is an algorithm for first-order gradient-based optimization of stochastic objective functions based on adaptive estimates of lower-order moments.

While the model training process tries to ensure that the learned model parameters are close to the minima of the objective function, the overall performance of a model also depends

on the model's *hyperparameters*, which are model settings that aren't learned by the training algorithm. For example, the dimensionality of the learned node representation, `num-hidden`, is a hyperparameter that affects model performance. Therefore, it is common in machine learning to perform hyperparameter optimization (HPO) to choose the suitable hyperparameters.

Neptune ML uses a SageMaker AI hyperparameter tuning job to launch multiple instances of model training with different hyperparameter configurations to try to find the best model for a range of hyperparameters settings. See Customizing model hyperparameter configurations in Neptune ML.

Knowledge graph embedding models in Neptune ML

Knowledge graphs (KGs) are graphs that encode information about different entities (nodes) and their relations (edges). In Neptune ML, knowledge graph embedding models are applied by default for performing link prediction when the graph does not contain node properties, only relations with other nodes. Although, R-GCN models with learnable embeddings can also be used for these graphs by specifying the model type as `"rgcn"`, knowledge graph embedding models are simpler and are designed to be effective for learning representations for large scale knowledge graphs.

Knowledge graph embedding models are used in a link prediction task to predict the nodes or relations that complete a triple `(h, r, t)` where **h** is the source node, **r** is the relation type and **t** is the destination node.

The knowledge graph embedding models implemented in Neptune ML are `distmult`, `transE`, and `rotatE`. To learn more about knowledge graph embedding models, see DGL-KE.

Training custom models in Neptune ML

Neptune ML lets you define and implement custom models of your own, for particular scenarios. See Custom models in Neptune ML for information about how to implement a custom model and how to use Neptune ML infrastructure to train it.

Customizing model hyperparameter configurations in Neptune ML

When you start a Neptune ML model-training job, Neptune ML automatically uses the information inferred from the preceding data-processing job. It uses the information to generate hyperparameter configuration ranges that are used to create a SageMaker AI hyperparameter tuning job to train multiple models for your task. That way, you don't have to specify a long list of hyperparameter values for the models to be trained with. Instead, the model hyperparameter ranges and defaults are selected based on the task type, graph type, and the tuning-job settings.

However, you can also override the default hyperparameter configuration and provide custom hyperparameters by modifying a JSON configuration file that the data-processing job generates.

Using the Neptune ML modelTraining API, you can control several high level hyperparameter tuning job settings like `maxHPONumberOfTrainingJobs`, `maxHPOParallelTrainingJobs`, and `trainingInstanceType`. For more fine-grained control over the model hyperparameters, you can customize the `model-HPO-configuration.json` file that the data-processing job generates. The file is saved in the Amazon S3 location that you specified for processing-job output.

You can download the file, edit it to override the default hyperparameter configurations, and upload it back to the same Amazon S3 location. Do not change the name of the file, and be careful to follow these instructions as you edit.

To download the file from Amazon S3:

```
aws s3 cp \
  s3:////(bucket name)/(path to output folder)/model-HPO-configuration.json \
  ./
```

When you have finished editing, upload the file back to where it was:

```
aws s3 cp \
  model-HPO-configuration.json \
  s3://(bucket name)/(path to output folder)/model-HPO-configuration.json
```

Structure of the `model-HPO-configuration.json` file

The `model-HPO-configuration.json` file specifies the model to be trained, the machine learning `task_type` and the hyperparameters that should be varied or fixed for the various runs of model training.

The hyperparameters are categorized as belonging to various tiers that signify the precedence given to the hyperparameters when the hyperparameter tuning job is invoked:

- Tier-1 hyperparameters have the highest precedence. If you set `maxHPONumberOfTrainingJobs` to a value less than 10, only Tier-1 hyperparameters are tuned, and the rest take their default values.
- Tier-2 hyperparameters have lower precedence, so if you have more than 10 but less than 50 total training jobs for a tuning job, then both Tier-1 and Tier-2 hyperparameters are tuned.
- Tier 3 hyperparameters are tuned together with Tier-1 and Tier-2 only if you have more than 50 total training jobs.
- Finally, fixed hyperparameters are not tuned at all, and always take their default values.

Example of a `model-HPO-configuration.json` file

The following is a sample `model-HPO-configuration.json` file:

```
{
  "models": [
    {
      "model": "rgcn",
      "task_type": "node_class",
      "eval_metric": {
        "metric": "acc"
      },
      "eval_frequency": {
          "type":  "evaluate_every_epoch",
          "value":  1
      },
      "1-tier-param": [
        {
            "param": "num-hidden",
            "range": [16, 128],
            "type": "int",
            "inc_strategy": "power2"
        },
        {
          "param": "num-epochs",
          "range": [3,30],
          "inc_strategy": "linear",
          "inc_val": 1,
          "type": "int",
```

```
          "node_strategy": "perM"
        },
        {
          "param": "lr",
          "range": [0.001,0.01],
          "type": "float",
          "inc_strategy": "log"
        }
      ],
      "2-tier-param": [
        {
          "param": "dropout",
          "range": [0.0,0.5],
          "inc_strategy": "linear",
          "type": "float",
          "default": 0.3
        },
        {
          "param": "layer-norm",
          "type": "bool",
          "default": true
        }
      ],
      "3-tier-param": [
        {
          "param": "batch-size",
          "range": [128, 4096],
          "inc_strategy": "power2",
          "type": "int",
          "default": 1024
        },
        {
          "param": "fanout",
          "type": "int",
          "options": [[10, 30],[15, 30], [15, 30]],
          "default": [10, 15, 15]
        },
        {
          "param": "num-layer",
          "range": [1, 3],
          "inc_strategy": "linear",
          "inc_val": 1,
          "type": "int",
          "default": 2
```

```
        },
        {
          "param": "num-bases",
          "range": [0, 8],
          "inc_strategy": "linear",
          "inc_val": 2,
          "type": "int",
          "default": 0
        }
      ],
      "fixed-param": [
        {
          "param": "concat-node-embed",
          "type": "bool",
          "default": true
        },
        {
          "param": "use-self-loop",
          "type": "bool",
          "default": true
        },
        {
          "param": "low-mem",
          "type": "bool",
          "default": true
        },
        {
          "param": "l2norm",
          "type": "float",
          "default": 0
        }
      ]
    }
  ]
}
```

Elements of a `model-HPO-configuration.json` file

The file contains a JSON object with a single top-level array named `models` that contains a single model-configuration object. When customizing the file, make sure the `models` array only has one model-configuration object in it. If your file contains more than one model-configuration object, the tuning job will fail with a warning.

The model-configuration object contains the following top-level elements:

- **`model`** – (*String*) The model type to be trained (**do not modify**). Valid values are:
  - `"rgcn"` – This is the default for node classification and regression tasks, and for heterogeneous link prediction tasks.
  - `"transe"` – This is the default for KGE link prediction tasks.
  - `"distmult"` – This is an alternative model type for KGE link prediction tasks.
  - `"rotate"` – This is an alternative model type for KGE link prediction tasks.

  As a rule, don't directly modify the `model` value, because different model types often have substantially different applicable hyperparameters, which can result in a parsing error after the training job has started.

  To change the model type, use the `modelName` parameter in the modelTraining API rather than change it in the `model-HPO-configuration.json` file.

  A way to change the model type and make fine-grain hyperparameter changes is to copy the default model configuration template for the model that you want to use and paste that into the `model-HPO-configuration.json` file. There is a folder named `hpo-configuration-templates` in the same Amazon S3 location as the `model-HPO-configuration.json` file if the inferred task type supports multiple models. This folder contains all the default hyperparameter configurations for the other models that are applicable to the task.

  For example, if you want to change the model and hyperparameter configurations for a KGE link-prediction task from the default `transe` model to a `distmult` model, simply paste the contents of the `hpo-configuration-templates/distmult.json` file into the `model-HPO-configuration.json` file and then edit the hyperparameters as necessary.

  > **Note**
  >
  > If you set the `modelName` parameter in the `modelTraining` API and also change the `model` and hyperparameter specification in the `model-HPO-configuration.json` file, and these are different, the `model` value in the `model-HPO-configuration.json` file takes precedence, and the `modelName` value is ignored.

- **`task_type`** – (*String*) The machine learning task type inferred by or passed directly to the data-processing job (**do not modify**). Valid values are:

- `"node_class"`
- `"node_regression"`
- `"link_prediction"`

The data-processing job infers the task type by examining the exported dataset and the generated training-job configuration file for properties of the dataset.

This value should not be changed. If you want to train a different task, you need to run a new data-processing job. If the `task_type` value is not what you were expecting, you should check the inputs to your data-processing job to make sure that they are correct. This includes parameters to the `modelTraining` API, as well as in the training-job configuration file generated by the data-export process.

- **`eval_metric`** – (*String*) The evaluation metric should be used for evaluating the model performance and for selecting the best-performing model across HPO runs. Valid values are:
  - `"acc"` – Standard classification accuracy. This is the default for single-label classification tasks, unless imbalanced labels are found during data processing, in which case the default is `"F1"`.
  - `"acc_topk"` – The number of times the correct label is among the top **k** predictions. You can also set the value **k** by passing in `topk` as an extra key.
  - `"F1"` – The F1 score.
  - `"mse"` – Mean-squared error metric, for regression tasks.
  - `"mrr"` – Mean reciprocal rank metric.
  - `"precision"` – The model precision, calculated as the ratio of true positives to predicted positives: `= true-positives / (true-positives + false-positives)`.
  - `"recall"` – The model recall, calculated as the ratio of true positives to actual positives: `= true-positives / (true-positives + false-negatives)`.
  - `"roc_auc"` – The area under the ROC curve. This is the default for multi-label classification.

  For example, to change the metric to F1, change the `eval_metric` value as follows:

```
"  eval_metric": {
    "metric": "F1",
  },
```

  Or, to change the metric to a `topk` accuracy score, you would change `eval_metric` as follows:

```
"eval_metric": {
  "metric": "acc_topk",
  "topk": 2
},
```

- **`eval_frequency`** – (*Object*) Specifies how often during training the performance of the model on the validation set should be checked. Based on the validation performance, early stopping can then be initiated and the best model can be saved.

  The `eval_frequency` object contains two elements, namely `"type"` and `"value"`. For example:

  ```
  "eval_frequency": {
    "type":  "evaluate_every_pct",
    "value":  0.1
  },
  ```

  Valid `type` values are:

  - **`evaluate_every_pct`** – Specifies the percentage of training to be completed for each evaluation.

    For `evaluate_every_pct`, the `"value"` field contains a floating-point number between zero and one which expresses that percentage.

  - **`evaluate_every_batch`** – Specifies the number of training batches to be completed for each evaluation.

    For `evaluate_every_batch`, the `"value"` field contains an integer which expresses that batch count.

  - **`evaluate_every_epoch`** – Specifies the number of epochs per evaluation, where a new epoch starts at midnight.

    For `evaluate_every_epoch`, the `"value"` field contains an integer which expresses that epoch count.

  The default setting for `eval_frequency` is:

  ```
  "eval_frequency": {
    "type":  "evaluate_every_epoch",
  ```

```
    "value":  1
  },
```

- **`1-tier-param`** – (*Required*) An array of Tier-1 hyperparameters.

  If you don't want to tune any hyperparameters, you can set this to an empty array. This does not affect the total number of training jobs launched by the SageMaker AI hyperparameter tuning job. It just means that all training jobs, if there is more than 1 but less than 10, will run with the same set of hyperparameters.

  On the other hand, if you want to treat all your tunable hyperparameters with equal significance then you can put all the hyperparameters in this array.

- **`2-tier-param`** – (*Required*) An array of Tier-2 hyperparameters.

  These parameters are only tuned if `maxHPONumberOfTrainingJobs` has a value greater than 10. Otherwise, they are fixed to the default values.

  If you have a training budget of at most 10 training jobs or don't want Tier-2 hyperparameters for any other reason, but you want to tune all tunable hyperparameters, you can set this to an empty array.

- **`3-tier-param`** – (*Required*) An array of Tier-3 hyperparameters.

  These parameters are only tuned if `maxHPONumberOfTrainingJobs` has a value greater than 50. Otherwise, they are fixed to the default values.

  If you don't want Tier-3 hyperparameters, you can set this to an empty array.

- **`fixed-param`** – (*Required*) An array of fixed hyperparameters that take only their default values and do not vary in different training jobs.

  If you want to vary all hyperparameters, you can set this to an empty array and either set the value for `maxHPONumberOfTrainingJobs` large enough to vary all tiers or make all hyperparameters Tier-1.

The JSON object that represents each hyperparameter in `1-tier-param`, `2-tier-param`, `3-tier-param`, and `fixed-param` contains the following elements:

- **`param`** – (*String*) The name of the hyperparameter (**do not change**).

  See the list of valid hyperparameter names in Neptune ML.

- **`type`** – (*String*) The hyperparameter type (**do not change**).

  Valid types are: `bool`, `int`, and `float`.
- **`default`** – (*String*) The default value for the hyperparameter.

  You can set a new default value.

Tunable hyperparameters can also contain the following elements:

- **`range`** – (*Array*) The range for a continuous tunable hyperparameter.

  This should be an array with two values, namely the minimum and maximum of the range (`[min, max]`).
- **`options`** – (*Array*) The options for a categorical tunable hyperparameter.

  This array should contain all the options to consider:

  ```
  "options" : [value1, value2, ... valuen]
  ```
- **`inc_strategy`** – (*String*) The type of incremental change for continuous tunable hyperparameter ranges (**do not change**).

  Valid values are `log`, `linear`, and `power2`. This applies only when the range key is set.

  Modifying this may result in not using the full range of your hyperparameter for tuning.
- **`inc_val`** – (*Float*) The amount by which successive increments differ for continuous tunablehyperparameters (**do not change**).

  This applies only when the range key is set.

  Modifying this may result in not using the full range of your hyperparameter for tuning.
- **`node_strategy`** – (*String*) Indicates that the effective range for this hyperparameter should change based on the number of nodes in the graph (**do not change**).

  Valid values are `"perM"` (per million), `"per10M"` (per 10 million), and `"per100M"` (per 100 million).

  Rather than change this value, change the `range` instead.

- **`edge_strategy`** – (*String*) Indicates that the effective range for this hyperparameter should change based on the number of edges in the graph (**do not change**).

  Valid values are `"perM"` (per million), `"per10M"` (per 10 million), and `"per100M"` (per 100 million).

  Rather than change this value, change the `range` instead.

List of all the hyperparameters in Neptune ML

The following list contains all the hyperparameters that can be set anywhere in Neptune ML, for any model type and task. Because they are not all applicable to every model type, it is important that you only set hyperparameters in the `model-HPO-configuration.json` file that appear in the template for the model you're using.

- **`batch-size`** – The size of the batch of target nodes using in one forward pass. *Type*: `int`.

  Setting this to a much larger value can cause memory issues for training on GPU instances.
- **`concat-node-embed`** – Indicates whether to get the initial representation of a node by concatenating its processed features with learnable initial node embeddings in order to increase the expressivity of the model. *Type*: `bool`.
- **`dropout`** – The dropout probability applied to dropout layers. *Type*: `float`.
- **`edge-num-hidden`** – The hidden layer size or number of units for the edge feature module. Only used when `use-edge-features` is set to `True`. *Type*: float.
- **`enable-early-stop`** – Toggles whether or not to use the early stopping feature. *Type*: `bool`. *Default*: `true`.

  Use this Boolean parameter to turn off the early stop feature.
- **`fanout`** – The number of neighbors to sample for a target node during neighbor sampling. *Type*: `int`.

  This value is tightly coupled with `num-layers` and should always be in the same hyperparameter tier. This is because you can specify a fanout for each potential GNN layer.

  Because this hyperparameter can cause model performance to vary widely, it should be fixed or set as a Tier-2 or Tier-3 hyperparameter. Setting it to a large value can cause memory issues for training on GPU instance.

- **gamma** – The margin value in the score function. *Type*: `float`.

  This applies to KGE link-prediction models only.
- **l2norm** – The weight decay value used in the optimizer which imposes an L2 normalization penalty on the weights. *Type*: `bool`.
- **layer-norm** – Indicates whether to use layer normalization for `rgcn` models. *Type*: `bool`.
- **low-mem** – Indicates whether to use a low-memory implementation of the relation message passing function at the expense of speed. *Type*: `bool`.
- **lr** – The learning rate. *Type*: `float`.

  This should be set as a Tier-1 hyperparameter.
- **neg-share** – In link prediction, indicates whether positive sampled edges can share negative edge samples. *Type*: `bool`.
- **num-bases** – The number of bases for basis decomposition in a `rgcn` model. Using a value of `num-bases` that is less than the number of edge types in the graph acts as a regularizer for the `rgcn` model. *Type*: `int`.
- **num-epochs** – The number of epochs of training to run. *Type*: `int`.

  An epoch is a complete training pass through the graph.
- **num-hidden** – The hidden layer size or number of units. *Type*: `int`.

  This also sets the initial embedding size for featureless nodes.

  Setting this to a much larger value without reducing `batch-size` can cause out-of-memory issues for training on GPU instance.
- **num-layer** – The number of GNN layers in the model. *Type*: `int`.

  This value is tightly coupled with the fanout parameter and should come after fanout is set in the same hyperparameter tier.

  Because this can cause model performance to vary widely, it should be fixed or set as a Tier-2 or Tier-3 hyperparameter.
- **num-negs** – In link prediction, the number of negative samples per positive sample. *Type*: `int`.
- **per-feat-name-embed** – Indicates whether to embed each feature by independently transforming it before combining features. *Type*: `bool`.

When set to `true`, each feature per node is independently transformed to a fixed dimension size before all the transformed features for the node are concatenated and further transformed to the `num_hidden` dimension.

When set to `false`, the features are concatenated without any feature-specific transformations.

- **`regularization-coef`** – In link prediction, the coefficient of regularization loss. *Type*: `float`.
- **`rel-part`** – Indicates whether to use relation partition for KGE link prediction. *Type*: `bool`.
- **`sparse-lr`** – The learning rate for learnable-node embeddings. *Type*: `float`.

  Learnable initial node embeddings are used for nodes without features or when `concat-node-embed` is set. The parameters of the sparse learnable node embedding layer are trained using a separate optimizer which can have a separate learning rate.
- **`use-class-weight`** – Indicates whether to apply class weights for imbalanced classification tasks. If set to to `true`, the label counts are used to set a weight for each class label. *Type*: `bool`.
- **`use-edge-features`** – Indicates whether to use edge features during message passing. If set to `true`, a custom edge feature module is added to the RGCN layer for edge types that have features. *Type*: `bool`.
- **`use-self-loop`** – Indicates whether to include self loops in training a `rgcn` model. *Type*: `bool`.
- **`window-for-early-stop`** – Controls the number of latest validation scores to average to decide on an early stop. The default is 3. type=int. See also Early stopping of the model training process in Neptune ML. *Type*: `int`. *Default*: 3.

  See .

Customizing hyperparameters in Neptune ML

When you are editing the `model-HPO-configuration.json` file, the following are the most common kinds of changes to make:

- Edit the minimum and/or maximum values of `range` hyperparameters.
- Set a hyperparameter to a fixed value by moving it to the `fixed-param` section and setting its default value to the fixed value you want it to take.

- Change the priority of a hyperparameter by placing it in a particular tier, editing its range, and making sure that its default value is set appropriately.

Model training best practices

There are things you can do to improve the performance of Neptune ML models.

Choose the right node property

Not all the properties in your graph may be meaningful or relevant to your machine learning tasks. Any irrelevant properties should be excluded during data export.

Here are some best practices:

- Use domain experts to help evaluate the importance of features and the feasibility of using them for predictions.
- Remove the features that you determine are redundant or irrelevant to reduce noise in the data and unimportant correlations.
- Iterate as you build your model. Adjust the features, feature combinations, and tuning objectives as you go along.

Feature Processing in the Amazon Machine Learning Developer Guide provides additional guidelines for feature processing that are relevant to Neptune ML.

Handle outlier data points

An outlier is a data point that is significantly different from the remaining data. Data outliers can spoil or mislead the training process, resulting in longer training time or less accurate models. Unless they are truly important, you should eliminate outliers before exporting the data.

Remove duplicate nodes and edges

Graphs stored in Neptune may have duplicate nodes or edges. These redundant elements will introduce noise for ML model training. Eliminate duplicate nodes or edges before exporting the data.

Tune the graph structure

When the graph is exported, you can change the way features are processed and how the graph is constructed, to improve the model performance.

Here are some best practices:

- When an edge property has the meaning of categories of edges, it is worth turning it into edge types in some cases.
- The default normalization policy used for a numerical property is `min-max`, but in some cases other normalization policies work better. You can preprocess the property and change the normalization policy as explained in Elements of a `model-HPO-configuration.json` file.
- The export process automatically generates feature types based on property types. For example, it treats `String` properties as categorical features and `Float` and `Int` properties as numerical features. If you need to, you can modify the feature type after export (see Elements of a `model-HPO-configuration.json` file).

Tune the hyperparameter ranges and defaults

The data-processing operation infers hyperparameter configuration ranges from the graph. If the generated model hyperparameter ranges and defaults don't work well for your graph data, you can edit the HPO configuration file to create your own hyperparameter tuning strategy.

Here are some best practices:

- When the graph goes large, the default hidden dimension size may not be large enough to contain all the information. You can change the `num-hidden` hyperparameter to control the hidden dimension size.
- For knowledge graph embedding (KGE) models, you may want to change the specific model being used according to your graph structure and budget.

  `TrainsE` models have difficulty in dealing with one-to-many (1-N), many-to-one (N-1), and many-to-many (N-N) relations. `DistMult` models have difficulty in dealing with symmetric relations. `RotatE` is good at modeling all kinds of relations but is more expensive than `TrainsE` and `DistMult` during training.
- In some cases, when both node identification and node feature information are important, you should use `` `concat-node-embed` `` to tell the Neptune ML model to get the initial representation of a node by concatenating its features with its initial embeddings.
- When you are getting reasonably good performance over some hyperparameters, you can adjust the hyperparameter search space according to those results.

Early stopping of the model training process in Neptune ML

Early stopping can significantly reduce the model-training run time and associated costs without degrading model performance. It also prevent the model from overfitting on the training data.

Early stopping depends on regular measurements of validation-set performance. Initially, performance improves as training proceeds, but when the model starts overfitting, it starts to decline again. The early stopping feature identifies the point at which the model starts overfitting and halts model training at that point.

Neptune ML monitors the validation metric calls and compares the most recent validation metric to the average of validation metrics over the last **n** evaluations, where **n** is a number set using the `window-for-early-stop` parameter. As soon as the validation metric is worse than that average, Neptune ML stops the model training and saves the best model so far.

You can control early stopping using the following parameters:

- **`window-for-early-stop`** – The value of this parameter is an integer that specifies the number of recent validation scores to average when deciding on an early stop. The default value is 3.
- **`enable-early-stop`** – Use this Boolean parameter to turn off the early stop feature. By default, its value is `true`.

Early stopping of the HPO process in Neptune ML

The early stop feature in Neptune ML also stops training jobs that are not performing well compared to other training jobs, using the SageMaker AI HPO warm-start feature. This too can reduce costs and improve the quality of HPO.

See Run a warm start hyperparameter tuning job for a description of how this works.

Warm start provides the ability to pass information learned from previous training jobs to subsequent training jobs and provides two distinct benefits:

- First, the results of previous training jobs are used to select good combinations of hyperparameters to search over in the new tuning job.
- Second, it allows early stopping to access more model runs, which reduces tuning time.

This feature is enabled automatically in Neptune ML, and allows you strike a balance between model training time and performance. If you are satisfied with the performance of the current model, you can use that model. Otherwise, you run more HPOs that are warm-started with the results of previous runs so as to discover a better model.

Get professional support services

AWS offers professional support services to help you with problems in your machine learning on Neptune projects. If you get stuck, reach out to AWS support.

Use a trained model to generate new model artifacts

Using the Neptune ML model transform command, you can compute model artifacts like node embeddings on processed graph data using pre-trained model parameters.

Model transform for incremental inference

In the incremental model inference workflow, after you have processed the updated graph data that you exported from Neptune you can start a model transform job using a command like the following:

AWS CLI

```
aws neptunedata start-ml-model-transform-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-transform job ID)" \
  --data-processing-job-id "(the data-processing job-id of a completed job)" \
  --ml-model-training-job-id "(the ML model training job-id)" \
  --model-transform-output-s3-location "s3://(your S3 bucket)/neptune-model-
transform/"
```

For more information, see start-ml-model-transform-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_model_transform_job(
    id='(a unique model-transform job ID)',
    dataProcessingJobId='(the data-processing job-id of a completed job)',
    mlModelTrainingJobId='(the ML model training job-id)',
    modelTransformOutputS3Location='s3://(your S3 bucket)/neptune-model-transform/'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltransform \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-transform job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "mlModelTrainingJobId": "(the ML model training job-id)",
        "modelTransformOutputS3Location" : "s3:///(your S3 bucket)/neptune-model-
transform/"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltransform \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-transform job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "mlModelTrainingJobId": "(the ML model training job-id)",
        "modelTransformOutputS3Location" : "s3:///(your S3 bucket)/neptune-model-
transform/"
      }'
```

You can then pass the ID of this job to the create-endpoints API call to create a new endpoint or update an existing one with the new model artifacts generated by this job. This allows the new or updated endpoint to provide model predictions for the updated graph data.

Model transform for any training job

You can also supply a `trainingJobName` parameter to generate model artifacts for any of the SageMaker AI training jobs launched during Neptune ML model training. Since a Neptune ML model training job can potentially launch many SageMaker AI training jobs, this gives you the flexibility to create an inference endpoint based on any of those SageMaker AI training jobs.

For example:

AWS CLI

```
aws neptunedata start-ml-model-transform-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-transform job ID)" \
  --training-job-name "(name of a completed SageMaker training job)" \
  --model-transform-output-s3-location "s3:////(your S3 bucket)/neptune-model-
transform/"
```

For more information, see start-ml-model-transform-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_model_transform_job(
    id='(a unique model-transform job ID)',
    trainingJobName='(name of a completed SageMaker training job)',
    modelTransformOutputS3Location='s3://(your S3 bucket)/neptune-model-transform/'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltransform \
```

```
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-transform job ID)",
        "trainingJobName" : "(name of a completed SageMaker training job)",
        "modelTransformOutputS3Location" : "s3:///(your S3 bucket)/neptune-model-
transform/"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltransform \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-transform job ID)",
        "trainingJobName" : "(name of a completed SageMaker training job)",
        "modelTransformOutputS3Location" : "s3:///(your S3 bucket)/neptune-model-
transform/"
      }'
```

If the original training job was for a user-provided custom model, you must include a `customModelTransformParameters` object when invoking a model transform. See Custom models in Neptune ML for information about how to implement and use a custom model.

> **Note**
>
> The `modeltransform` command always runs the model transform on the best SageMaker AI training job for that training.

See The modeltransform command for more information about model transform jobs.

Artifacts produced by model training in Neptune ML

After model training, Neptune ML uses the best trained model parameters to generate model artifacts that are necessary for launching the inference endpoint and providing model predictions. These artifacts are packaged by the training job and stored in the Amazon S3 output location of the best SageMaker AI training job.

The following sections describe what is included in the model artifacts for the various tasks, and how the model transform command uses a pre-existing trained model to generate artifacts even on new graph data.

Artifacts generated for different tasks

The content of the model artifacts generated by the training process depends on the target machine learning task:

- **Node classification and regression** – For node property prediction, the artifacts include model parameters, node embeddings from the GNN encoder, model predictions for nodes in the training graph, and some configuration files for the inference endpoint. In node classification and node regression tasks, model predictions are pre-computed for nodes present during training to reduce query latency.
- **Edge classification and regression** – For edge property prediction, the artifacts also include model parameters and node embeddings. The parameters of the model decoder are especially important for inference because we compute the edge classification or edge regression predictions by applying the model decoder to the embeddings of the source and destination vertex of an edge.
- **Link prediction** – For link prediction, in addition to the artifacts generated for edge property prediction, the DGL graph is also included as an artifact because link prediction requires the training graph to perform predictions. The objective of link prediction is to predict the destination vertices that are likely to combine with a source vertex to form an edge of a particular type in the graph. In order to do this, the node embedding of the source vertex and a learned representation for the edge type are combined with the node embeddings of all possible destination vertices to produce an edge likelihood score for each of the destination vertices. The scores are then sorted to rank the potential destination vertices and return the top candidates.

For each of the task types, the Graph Neural Network model weights from DGL are saved in the model artifact. This allows Neptune ML to compute fresh model outputs as the graph changes

(*inductive* inference), in addition to using pre-computed predictions and embeddings (*transductive* inference) to reduce latency.

Generating new model artifacts

The model artifacts generated after model training in Neptune ML are directly tied to the training process. This means that the pre-computed embeddings and predictions only exist for entities that were in the original training graph. Although inductive inference mode for Neptune ML endpoints can compute predictions for new entities in real-time, you may want to generate batch predictions on new entities without querying an endpoint.

In order to get batch model predictions for new entities that have been added to the graph, new model artifacts need to be recomputed for the new graph data. This is accomplished using the `modeltransform` command. You use the `modeltransform` command when you only want batch predictions without setting up an endpoint, or when you want all the predictions generated so that you can write them back to the graph.

Since model training implicitly performs a model transform at the end of the training process, model artifacts are always recomputed on the training graph data by a training job. However, the `modeltransform` command can also compute model artifacts on graph data that was not used for training a model. In order to this, the new graph data must be processed using the same feature encodings as the original graph data and must adhere to the same graph schema.

You can accomplish this by first creating a new data processing job that is a clone of the data processing job run on the original training graph data, and running it on the new graph data (see Processing updated graph data for Neptune ML). Then, call the `modeltransform` command with the new `dataProcessingJobId` and the old `modelTrainingJobId` to recompute the model artifacts on the updated graph data.

For node property prediction, the node embeddings and predictions are recomputed on the new graph data, even for nodes that were present in the original training graph.

For edge property prediction and link prediction, the node embeddings are also recomputed and similarly override any existing node embeddings. To recompute the node embeddings, Neptune ML applies the learned GNN encoder from the previous trained model to the nodes of the new graph data with their new features.

For nodes that do not have features, the learned initial representations from the original model training are re-used. For new nodes that do not have features and were not present in the original

training graph, Neptune ML initializes their representation asthe average of the learned initial node representations of that node type present in the original training graph. This can cause some performance drop in model predictions if you have many new nodes that do not have features, since they will all be initialized to the average initial embedding for that node type.

If your model is trained with `concat-node-embed` set to true, then the initial node representations are created by concatenating the node features with the learnable initial representation. Thus, for the updated graph, the initial node representation of new nodes also uses the average initial node embeddings, concatenated with new node features.

Additionally, node deletions are currently not supported. If nodes have been removed in the updated graph, you have to retrain the model on the updated graph data.

Recomputing the model artifacts re-uses the learned model parameters on a new graph, and should only be done when the new graph is very similar to the old graph. If your new graph is not sufficiently similar, you need to retrain the model to obtain similar model performance on the new graph data. What constitutes sufficiently similar depends on the structure of your graph data, but as a rule of thumb you should retrain your model if your new data is more than 10-20% different from the original training graph data.

For graphs where all the nodes have features, the higher end of the threshold (20% different) applies but for graphs where many nodes do not have features and the new nodes added to the graph don't have properties, then the lower end (10% different) may be even be too high.

See The modeltransform command for more information about model transform jobs.

Custom models in Neptune ML

> **Note**
>
> Neptune ML custom model support relies on an older version of Python 3. To create and run custom GNN models with up-to-date dependencies use GraphStorm on SageMaker. Real-time inductive inference is not currently supported for custom models.

Neptune ML lets you define your own custom model implementations using Python. You can train and deploy custom models using Neptune ML infrastructure very much as you do for the built-in models, and use them to obtain predictions through graph queries.

You can start implementing a custom model of your own in Python by following the Neptune ML toolkit examples, and by using the model components provided in the Neptune ML toolkit. The following sections provide more details.

Contents

- Overview of custom models in Neptune ML
  - When to use a custom model in Neptune ML
  - Workflow for developing and using a custom model in Neptune ML
- Custom model development in Neptune ML
  - Custom model training script development in Neptune ML
  - Custom model transform script development in Neptune ML
  - Custom model-hpo-configuration.json file in Neptune ML
  - Local testing of your custom model implementation in Neptune ML

Overview of custom models in Neptune ML

When to use a custom model in Neptune ML

Neptune ML's built-in models handle all the standard tasks supported by Neptune ML, but there may be cases where you want to have more granular control over the model for a particular task, or need to customize the model training process. For example, a custom model is appropriate in the following situations:

- Feature encoding for text features of very large text models need to be run on GPU.
- You want to use your own custom Graph Neural Network (GNN) model developed in Deep Graph Library (DGL).
- You want to use tabular models or ensemble models for node classification and regression.

Workflow for developing and using a custom model in Neptune ML

Custom model support in Neptune ML is designed to integrate seamlessly into existing Neptune ML workflows. It works by running custom code in your source module on Neptune ML's infrastructure to train the model. Just as is the case for a built-in mode, Neptune ML automatically launches a SageMaker AI HyperParameter tuning job and selects the best model according to the evaluation metric. It then uses the implementation provided in your source module to generate model artifacts for deployment.

Data export, training configuration, and data preprocessing is the same for a custom model as for a built-in one.

After data preprocessing is when you can iteratively and interactively develop and test your custom model implementation using Python. When your model is production-ready, you can upload the resulting Python module to Amazon S3 like this:

```
aws s3 cp --recursive (source path to module) s3://(bucket name)/(destination path for
 your module)
```

Then, you can use the normal default or the incremental data workflow to deploy the model to production, with a few differences.

For model training using a custom model, you must provide a `customModelTrainingParameters` JSON object to the Neptune ML model training API to

ensure that your custom code is used. The fields in the `customModelTrainingParameters` object are as follows:

- **`sourceS3DirectoryPath`** – (*Required*) The path to the Amazon S3 location where the Python module implementing your model is located. This must point to a valid existing Amazon S3 location that contains, at a minimum, a training script, a transform script, and a `model-hpo-configuration.json` file.
- **`trainingEntryPointScript`** – (*Optional*) The name of the entry point in your module of a script that performs model training and takes hyperparameters as command-line arguments, including fixed hyperparameters.

  Default: `training.py`.
- **`transformEntryPointScript`** – (*Optional*) The name of the entry point in your module of a script that should be run after the best model from the hyperparameter search has been identified, to compute the model artifacts necessary for model deployment. It should be able to run with no command-line arguments.

  Default: `transform.py`.

For example:

AWS CLI

```
aws neptunedata start-ml-model-training-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-training job ID)" \
  --data-processing-job-id "(the data-processing job-id of a completed job)" \
  --train-model-s3-location "s3://(your Amazon S3 bucket)/neptune-model-graph-
autotrainer" \
  --model-name "custom" \
  --custom-model-training-parameters '{
    "sourceS3DirectoryPath": "s3://(your Amazon S3 bucket)/(path to your Python
 module)",
    "trainingEntryPointScript": "(your training script entry-point name in the
 Python module)",
    "transformEntryPointScript": "(your transform script entry-point name in the
 Python module)"
  }'
```

For more information, see start-ml-model-training-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_model_training_job(
    id='(a unique model-training job ID)',
    dataProcessingJobId='(the data-processing job-id of a completed job)',
    trainModelS3Location='s3:////(your Amazon S3 bucket)/neptune-model-graph-
autotrainer',
    modelName='custom',
    customModelTrainingParameters={
        'sourceS3DirectoryPath': 's3://(your Amazon S3 bucket)/(path to your Python
 module)',
        'trainingEntryPointScript': '(your training script entry-point name in the
 Python module)',
        'transformEntryPointScript': '(your transform script entry-point name in the
 Python module)'
    }
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltraining \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3://(your Amazon S3 bucket)/neptune-model-graph-
autotrainer",
        "modelName": "custom",
        "customModelTrainingParameters" : {
```

```
          "sourceS3DirectoryPath": "s3:////(your Amazon S3 bucket)/(path to your
 Python module)",
          "trainingEntryPointScript": "(your training script entry-point name in the
 Python module)",
          "transformEntryPointScript": "(your transform script entry-point name in
 the Python module)"
        }
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltraining \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3://(your Amazon S3 bucket)/neptune-model-graph-
autotrainer",
        "modelName": "custom",
        "customModelTrainingParameters" : {
          "sourceS3DirectoryPath": "s3://(your Amazon S3 bucket)/(path to your
 Python module)",
          "trainingEntryPointScript": "(your training script entry-point name in the
 Python module)",
          "transformEntryPointScript": "(your transform script entry-point name in
 the Python module)"
        }
      }'
```

Similarly, to enable a custom model transform, you must provide a `customModelTransformParameters` JSON object to the Neptune ML model transform API, with field values that are compatible with the saved model parameters from the training job. The `customModelTransformParameters` object contains these fields:

- **`sourceS3DirectoryPath`** – (*Required*) The path to the Amazon S3 location where the Python module implementing your model is located. This must point to a valid existing Amazon S3 location that contains, at a minimum, a training script, a transform script, and a `model-hpo-configuration.json` file.
- **`transformEntryPointScript`** – (*Optional*) The name of the entry point in your module of a script that should be run after the best model from the hyperparameter search has been identified, to compute the model artifacts necessary for model deployment. It should be able to run with no command-line arguments.

  Default: `transform.py`.

For example:

AWS CLI

```
aws neptunedata start-ml-model-transform-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-transform job ID)" \
  --training-job-name "(name of a completed SageMaker training job)" \
  --model-transform-output-s3-location "s3://(your Amazon S3 bucket)/neptune-model-
transform/" \
  --custom-model-transform-parameters '{
    "sourceS3DirectoryPath": "s3://(your Amazon S3 bucket)/(path to your Python
 module)",
    "transformEntryPointScript": "(your transform script entry-point name in the
 Python module)"
  }'
```

For more information, see start-ml-model-transform-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)
```

```
response = client.start_ml_model_transform_job(
    id='(a unique model-transform job ID)',
    trainingJobName='(name of a completed SageMaker training job)',
    modelTransformOutputS3Location='s3://(your Amazon S3 bucket)/neptune-model-
transform/',
    customModelTransformParameters={
        'sourceS3DirectoryPath': 's3://(your Amazon S3 bucket)/(path to your Python
 module)',
        'transformEntryPointScript': '(your transform script entry-point name in the
 Python module)'
    }
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltransform \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-transform job ID)",
        "trainingJobName" : "(name of a completed SageMaker training job)",
        "modelTransformOutputS3Location" : "s3://(your Amazon S3 bucket)/neptune-
model-transform/",
        "customModelTransformParameters" : {
          "sourceS3DirectoryPath": "s3://(your Amazon S3 bucket)/(path to your
 Python module)",
          "transformEntryPointScript": "(your transform script entry-point name in
 the Python module)"
        }
      }'
```

Note

This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltransform \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-transform job ID)",
        "trainingJobName" : "(name of a completed SageMaker training job)",
        "modelTransformOutputS3Location" : "s3:////(your Amazon S3 bucket)/neptune-
model-transform/",
        "customModelTransformParameters" : {
          "sourceS3DirectoryPath": "s3://(your Amazon S3 bucket)/(path to your
 Python module)",
          "transformEntryPointScript": "(your transform script entry-point name in
 the Python module)"
        }
      }'
```

Custom model development in Neptune ML

A good way to start custom model development is by following Neptune ML toolkit examples to structure and write your training module. The Neptune ML toolkit also implements modularized graph ML model components in the modelzoo that you can stack and use to create your custom model.

In addition, the toolkit provides utility functions that help you generate the necessary artifacts during model training and model transform. You can import this Python package in your custom implementation. Any functions or modules provided in the toolkit are also available in the Neptune ML training environment.

If your Python module has additional external dependencies, you can include these additional dependencies by creating a `requirements.txt` file in your module's directory. The packages listed in the `requirements.txt` file will then be installed before your training script is run.

At a minimum, the Python module that implements your custom model needs to contain the following:

- A training script entry point
- A transform script entry point
- A `model-hpo-configuration.json` file

Custom model training script development in Neptune ML

Your custom model training script should be an executable Python script like the Neptune ML toolkit's `train.py` example. It must accept hyperparameter names and values as command-line arguments. During model training, the hyperparameter names are obtained from the `model-hpo-configuration.json` file. The hyperparameter values either fall within the valid hyperparameter range if the hyperparameter is tunable, or take the default hyperparameter value if it is not tunable.

Your training script is run on a SageMaker AI training instance using a syntax like this:

```
python3 (script entry point) --(1st parameter) (1st value) --(2nd parameter) (2nd
 value) (...)
```

For all tasks, the Neptune ML AutoTrainer sends several required parameters to your training script in addition to the hyperparameters that you specify, and your script must be able to handle these additional parameters in order to work properly.

These additional required parameters vary somewhat by task:

For node classification or node regression

- **`task`** – The task type used internally by Neptune ML. For node classification this is `node_class`, and for node regression it is `node_regression`.
- **`model`** – The model name used internally by Neptune ML, which is `custom` in this case.
- **`name`** – The name of the task used internally by Neptune ML, which is `node_class-custom` for node classification in this case, and `node_regression-custom` for node regression.
- **`target_ntype`** – The name of the node type for classification or regression.
- **`property`** – The name of the node property for classification or regression.

For link prediction

- **`task`** – The task type used internally by Neptune ML. For link prediction, this is `link_predict`.
- **`model`** – The model name used internally by Neptune ML, which is `custom` in this case.
- **`name`** – The name of the task used internally by Neptune ML, which is `link_predict-custom` in this case.

For edge classification or edge regression

- **`task`** – The task type used internally by Neptune ML. For edge classification this is `edge_class`, and for edge regression it is `edge_regression`.
- **`model`** – The model name used internally by Neptune ML, which is `custom` in this case.
- **`name`** – The name of the task used internally by Neptune ML, which is `edge_class-custom` for edge classification in this case, and `edge_regression-custom` for edge regression.
- **`target_etype`** – The name of the edge type for classification or regression.
- **`property`** – The name of the edge property for classification or regression.

Your script should save the model parameters, as well as any other artifacts that will be needed to at the end of training.

You can use Neptune ML toolkit utility functions to determine the location of the processed graph data, the location where the model parameters should be saved, and what GPU devices are available on the training instance. See the train.py sample training script for examples of how to use these utility functions.

Custom model transform script development in Neptune ML

A transform script is needed to take advantage of the Neptune ML incremental workflow for model inference on evolving graphs without retraining the model. Even if all the artifacts necessary for model deployment are generated by the training script, you still need to provide a transform script if you want to generate updated models without retraining the model.

> **Note**
>
> Real-time inductive inference is not currently supported for custom models.

Your custom model transform script should be an executable Python script like the Neptune ML toolkit's transform.py example script. Because this script is invoked during model training with no command line arguments, any command line arguments that the script does accept must have defaults.

The script runs on a SageMaker AI training instance with a syntax like this:

```
python3 (your transform script entry point)
```

Your transform script will need various pieces of information, such as:

- The location of the processed graph data.
- The location where the model parameters are saved and where new model artifacts should be saved.
- The devices available on the instance.
- The hyperparameters that generated the best model.

These inputs are obtained using Neptune ML utility functions that your script can call. See the toolkit's sample transform.py script for examples of how to do that.

The script should save the node embeddings, node ID mappings, and any other artifacts necessary for model deployment for each task. See the model artifacts documentation for more information about the model artifacts required for different Neptune ML tasks.

Custom `model-hpo-configuration.json` file in Neptune ML

The `model-hpo-configuration.json` file defines hyperparameters for your custom model. It is in the same format as the `model-hpo-configuration.json` file used with the Neptune ML built-in models, and takes precedence over the version that is auto-generated by Neptune ML and uploaded to the location of your processed data.

When you add a new hyperparameter to your model, you must also add an entry for the hyperparameter in this file so that the hyperparameter is passed to your training script.

You must provide a range for a hyperparameter if you want it to be tunable, and set it as a `tier-1`, `tier-2`, or `tier-3` param. The hyperparameter will be tuned if the total number of training jobs configured allow for tuning hyperparameters in its tier. For a non-tunable parameter, you must provide a default value and add the hyperparameter to the `fixed-param` section of the file. See the toolkit's sample sample `model-hpo-configuration.json` file for an example of how to do that.

You must also provide the metric definition that the SageMaker AI HyperParameter Optimization job will use to evaluate the candidate models trained. To do this, you add an `eval_metric` JSON object to the `model-hpo-configuration.json` file like this:

```
"eval_metric": {
  "tuning_objective": {
      "MetricName": "(metric_name)",
      "Type": "Maximize"
  },
  "metric_definitions": [
    {
      "Name": "(metric_name)",
      "Regex": "(metric regular expression)"
    }
  ]
},
```

The `metric_definitions` array in the `eval_metric` object lists metric definition objects for each metric that you want SageMaker AI to extract from the training instance. Each metric definition object has a `Name` key that lets you provide a name for the metric (such as "accuracy", "f1", and so on) The `Regex` key lets you provide a regular expression string that matches how that particular metric is printed in the training logs. See the SageMaker AI HyperParameter Tuning page for more details on how to define metrics.

The `tuning_objective` object in `eval_metric` then allows you to specify which of the metrics in `metric_definitions` should be used as the evaluation metric that serves as the objective metric for hyperparameter optimization. The value for the `MetricName` must match the value of a `Name` in one of the definitions in `metric_definitions`. The value for `Type` should be either "Maximize" or "Minimize" depending on whether the metric should be interpreted as greater-is-better (like "accuracy") or less-is-better (like "mean-squared-error".

Errors in this section of the `model-hpo-configuration.json` file can result in failures of the Neptune ML model training API job, because the SageMaker AI HyperParameter Tuning job will not be able to select the best model.

Local testing of your custom model implementation in Neptune ML

You can use the Neptune ML toolkit Conda environment to run your code locally in order to test and validate your model. If you're developing on a Neptune Notebook instance, then this Conda environment will be pre-installed on the Neptune Notebook instance. If you're developing on a different instance, then you need to follow the local setup instructions in the Neptune ML toolkit.

The Conda environment accurately reproduces the environment where your model will run when you call the model training API. All of the example training scripts and transform scripts allow you to pass a command line `--local` flag to run the scripts in a local environment for easy debugging. This is a good practice while developing your own model because it allows you to interactively and iteratively test your model implementation. During model training in the Neptune ML production training environment, this parameter is omitted.

Creating an inference endpoint to query

An inference endpoint lets you query one specific model that the model-training process constructed. The endpoint attaches to the best-performing model of a given type that the training process was able to generate. The endpoint is then able to accept Gremlin queries from Neptune and return that model's predictions for inputs in the queries. After you have created an inference endpoint, it stays active until you delete it.

Managing inference endpoints for Neptune ML

After you have completed model training on data that you exported from Neptune, you can create an inference endpoint using a command like the following:

AWS CLI

```
aws neptunedata create-ml-endpoint \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique ID for the new endpoint)" \
  --ml-model-training-job-id "(the model-training job-id of a completed job)"
```

For more information, see create-ml-endpoint in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.create_ml_endpoint(
    id='(a unique ID for the new endpoint)',
    mlModelTrainingJobId='(the model-training job-id of a completed job)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/endpoints \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique ID for the new endpoint)",
        "mlModelTrainingJobId": "(the model-training job-id of a completed job)"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/endpoints \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique ID for the new endpoint)",
        "mlModelTrainingJobId": "(the model-training job-id of a completed job)"
      }'
```

You can also create an inference endpoint from a model created by a completed model transform job, in much the same way:

AWS CLI

```
aws neptunedata create-ml-endpoint \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique ID for the new endpoint)" \
  --ml-model-transform-job-id "(the model-transform job-id of a completed job)"
```

For more information, see create-ml-endpoint in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.create_ml_endpoint(
    id='(a unique ID for the new endpoint)',
    mlModelTransformJobId='(the model-transform job-id of a completed job)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/endpoints \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique ID for the new endpoint)",
        "mlModelTransformJobId": "(the model-transform job-id of a completed job)"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/endpoints \
  -H 'Content-Type: application/json' \
```

```
-d '{
      "id" : "(a unique ID for the new endpoint)",
      "mlModelTransformJobId": "(the model-transform job-id of a completed job)"
    }'
```

The details of how to use these commands are explained in The endpoints command, along with information about how to get the status of an endpoint, how to delete an endpoint, and how to list all inference endpoints.

Inference queries in Neptune ML

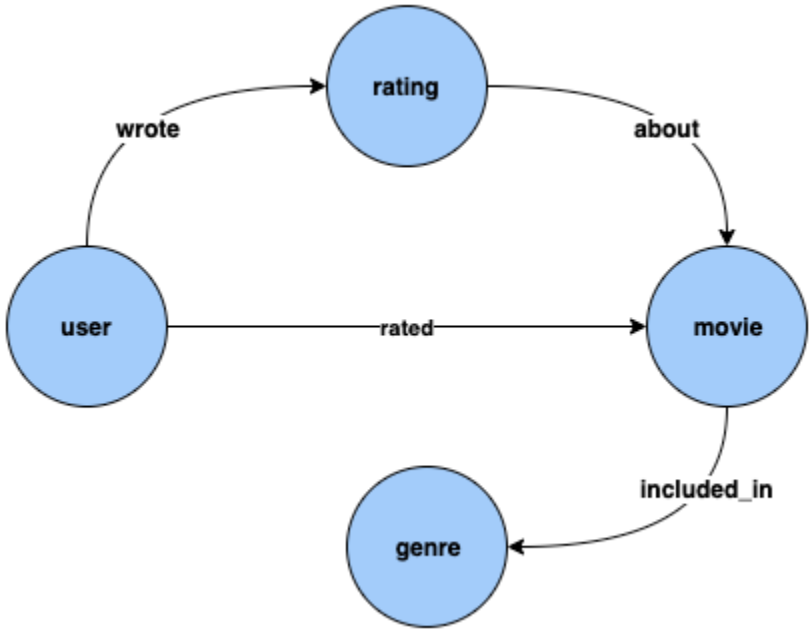
You can use either Gremlin or SPARQL to query a Neptune ML inference endpoint. Real-time inductive inference, however, is currently only supported for Gremlin queries.

Gremlin inference queries in Neptune ML

As described in Neptune ML capabilities, Neptune ML supports training models that can do the following kinds of inference tasks:

- **Node classification** – Predicts the categorical feature of a vertex property.
- **Node regression** – Predicts a numerical property of a vertex.
- **Edge classification** – Predicts the categorical feature of an edge property.
- **Edge regression** – Predicts a numerical property of an edge.
- **Link prediction** – Predicts destination nodes given a source node and outgoing edge, or source nodes given a destination node and incoming edge.

We can illustrate these different tasks with examples that use the MovieLens 100k dataset provided by GroupLens Research. This dataset consists of movies, users, and ratings of the movies by the users, from which we've created a property graph like this:



Node classification: In the dataset above, `Genre` is a vertex type which is connected to vertex type `Movie` by edge `included_in`. However, if we tweak the dataset to make `Genre` a categorical feature for vertex type `Movie`, then the problem of inferring `Genre` for new movies added to our knowledge graph can be solved using node classification models.

Node regression: If we consider the vertex type `Rating`, which has properties like `timestamp` and `score`, then the problem of inferring the numerical value `Score` for a `Rating` can be solved using node regression models.

Edge classification: Similarly, for a `Rated` edge, if we have a property `Scale` that can have one of the values, `Love`, `Like`, `Dislike`, `Neutral`, `Hate`, then the problem of inferring `Scale` for the `Rated` edge for new movies/ratings can be solved using edge classification models.

Edge regression: Similarly, for the same `Rated` edge, if we have a property `Score` that holds a numerical value for the rating, then this can be inferred from edge regression models.

Link prediction: Problems like, find the top ten users who are most likely to rate a given movie, or find the top ten Movies that a given user is most likely to rate, falls under link prediction.

Topics

- Neptune ML predicates used in Gremlin inference queries
- Gremlin node classification queries in Neptune ML
- Gremlin node regression queries in Neptune ML
- Gremlin edge classification queries in Neptune ML
- Gremlin edge regression queries in Neptune ML
- Gremlin link prediction queries using link-prediction models in Neptune ML
- List of exceptions for Neptune ML Gremlin inference queries

Neptune ML predicates used in Gremlin inference queries

`Neptune#ml.deterministic`

This predicate is an option for inductive inference queries — that is, for queries that include the `Neptune#ml.inductiveInference` predicate.

When using inductive inference, the Neptune engine creates the appropriate subgraph to evaluate the trained GNN model, and the requirements of this subgraph depend on parameters of the final model. Specifically, the `num-layer` parameter determines the number of traversal hops from the target nodes or edges, and the `fanouts` parameter specifies how many neighbors to sample at each hop (see HPO parameters).

By default, inductive inference queries run in non-deterministic mode, in which Neptune builds the neighborhood randomly. When making predictions, this normal random-neighbor sampling sometimes result in different predictions.

When you include `Neptune#ml.deterministic` in an inductive inference query, the Neptune engine attempts to sample neighbors in a deterministic way so that multiple invocations of the same query return the same results every time. The results can't be guaranteed to be completely deterministic, however, because changes to the underlying graph and artifacts of distributed systems can still introduce fluctuations.

You include the `Neptune#ml.deterministic` predicate in a query like this:

```
.with("Neptune#ml.deterministic")
```

If the `Neptune#ml.deterministic` predicate is included in a query that doesn't also include `Neptune#ml.inductiveInference`, it is simply ignored.

`Neptune#ml.disableInductiveInferenceMetadataCache`

This predicate is an option for inductive inference queries — that is, for queries that include the `Neptune#ml.inductiveInference` predicate.

For inductive inference queries, Neptune uses a metadata file stored in Amazon S3 to decide the number of hops and the fanout while building the neighborhood. Neptune normally caches this model metadata to avoid fetching the file from Amazon S3 repeatedly. Caching can be disabled by including the `Neptune#ml.disableInductiveInferenceMetadataCache` predicate in the query. Although it may be slower for Neptune to fetch the metadata directly from Amazon S3, it is useful when the SageMaker AI endpoint has been updated after retraining or transformation and the cache is stale.

You include the `Neptune#ml.disableInductiveInferenceMetadataCache` predicate in a query like this:

```
.with("Neptune#ml.disableInductiveInferenceMetadataCache")
```

Here is how a sample query might look in a Jupyter notebook:

```
%%gremlin
g.with("Neptune#ml.endpoint", "ep1")
```

```
.with("Neptune#ml.iamRoleArn", "arn:aws:iam::123456789012:role/NeptuneMLRole")
.with("Neptune#ml.disableInductiveInferenceMetadataCache")
.V('101').properties("rating")
.with("Neptune#ml.regression")
.with("Neptune#ml.inductiveInference")
```

`Neptune#ml.endpoint`

The `Neptune#ml.endpoint` predicate is used in a `with()` step to specify the inference endpoint, if necessary:

```
.with("Neptune#ml.endpoint", "the model's SageMaker AI inference endpoint")
```

You can identify the endpoint either by its `id` or its URL. For example:

```
.with( "Neptune#ml.endpoint", "node-classification-movie-lens-endpoint" )
```

Or:

```
.with( "Neptune#ml.endpoint", "https://runtime.sagemaker.us-east-1.amazonaws.com/
endpoints/node-classification-movie-lens-endpoint/invocations" )
```

> **Note**
>
> If you set the `neptune_ml_endpoint` parameter in your Neptune DB cluster parameter group to the endpoint `id` or URL, you don't need to include the `Neptune#ml.endpoint` predicate in each query.

`Neptune#ml.iamRoleArn`

`Neptune#ml.iamRoleArn` is used in a `with()` step to specify the ARN of the SageMaker AI execution IAM role, if necessary:

```
.with("Neptune#ml.iamRoleArn", "the ARN for the SageMaker AI execution IAM role")
```

For information about how to create the SageMaker AI execution IAM role, see Create a custom NeptuneSageMakerIAMRole role.

> **Note**
>
> If you set the `neptune_ml_iam_role` parameter in your Neptune DB cluster parameter group to the ARN of your SageMaker AI execution IAM role, you don't need to include the `Neptune#ml.iamRoleArn` predicate in each query.

Neptune#ml.inductiveInference

Transductive inference is enabled by default in Gremlin. To make a real-time inductive inference query, include the `Neptune#ml.inductiveInference` predicate like this:

```
.with("Neptune#ml.inductiveInference")
```

If your graph is dynamic, inductive inference is often the best choice, but if your graph is static, transductive inference is faster and more efficient.

Neptune#ml.limit

The `Neptune#ml.limit` predicate optionally limits the number of results returned per entity:

```
.with( "Neptune#ml.limit", 2 )
```

By default, the limit is 1, and the maximum number that can be set is 100.

Neptune#ml.threshold

The `Neptune#ml.threshold` predicate optionally establishes a cutoff threshold for result scores:

```
.with( "Neptune#ml.threshold", 0.5D )
```

This lets you discard all results with scores below the specified threshold.

Neptune#ml.classification

The `Neptune#ml.classification` predicate is attached to the `properties()` step to establish that the properties need to be fetched from the SageMaker AI endpoint of the node classification model:

```
.properties( "property key of the node classification
model" ).with( "Neptune#ml.classification" )
```

Neptune#ml.regression

The `Neptune#ml.regression` predicate is attached to the `properties()` step to establish that the properties need to be fetched from the SageMaker AI endpoint of the node regression model:

```
.properties( "property key of the node regression
model" ).with( "Neptune#ml.regression" )
```

Neptune#ml.prediction

The `Neptune#ml.prediction` predicate is attached to `in()` and `out()` steps to establish that this a link-prediction query:

```
.in("edge label of the link prediction
model").with("Neptune#ml.prediction").hasLabel("target node label")
```

Neptune#ml.score

The `Neptune#ml.score` predicate is used in Gremlin node or edge classification queries to fetch a machine-learning confidence Score. The `Neptune#ml.score` predicate should be passed together with the query predicate in the `properties()` step to obtain an ML confidence score for node or edge classification queries.

You can find a node classification example with other node classification examples, and an edge classification example in the edge classification section.

Gremlin node classification queries in Neptune ML

For Gremlin node classification in Neptune ML:

- The model is trained on one property of the vertices. The set of unique values of this property are referred to as a set of node classes, or simply, classes.
- The node class or categorical property value of a vertex's property can be inferred from the node classification model. This is useful where this property is not already attached to the vertex.
- In order to fetch one or more classes from a node classification model, you need to use the `with()` step with the predicate `Neptune#ml.classification` to configure the `properties()` step. The output format is similar to what you would expect if those were vertex properties.

> **Note**
>
> Node classification only works with string property values. That means that numerical property values such as `0` or `1` are not supported, although the string equivalents `"0"` and `"1"` are. Similarly, the Boolean property values `true` and `false` don't work, but `"true"` and `"false"` do.

Here is a sample node classification query:

```
g.with( "Neptune#ml.endpoint","node-classification-movie-lens-endpoint" )
 .with( "Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role" )
 .with( "Neptune#ml.limit", 2 )
 .with( "Neptune#ml.threshold", 0.5D )
 .V( "movie_1", "movie_2", "movie_3" )
 .properties("genre").with("Neptune#ml.classification")
```

The output of this query would look something like the following:

```
==>vp[genre->Action]
==>vp[genre->Crime]
==>vp[genre->Comedy]
```

In the query above, the `V()` and `properties()` steps are used as follows:

The `V()` step contains the set of vertices for which you want to fetch the classes from the node-classification model:

```
 .V( "movie_1", "movie_2", "movie_3" )
```

The `properties()` step contains the key on which the model was trained, and has `.with("Neptune#ml.classification")` to indicate that this is a node classification ML inference query.

Multiple property keys are not currently supported in a `properties().with("Neptune#ml.classification")` step. For example, the following query results in an exception:

```
g.with("Neptune#ml.endpoint", "node-classification-movie-lens-endpoint")
```

```
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V( "movie_1", "movie_2", "movie_3" )
 .properties("genre", "other_label").with("Neptune#ml.classification")
```

For the specific error message, see the list of Neptune ML exceptions.

A `properties().with("Neptune#ml.classification")` step can be used in combination with any of the following steps:

- `value()`
- `value().is()`
- `hasValue()`
- `has(value,"")`
- `key()`
- `key().is()`
- `hasKey()`
- `has(key,"")`
- `path()`

Other node-classification queries

If both the inference endpoint and the corresponding IAM role have been saved in your DB cluster parameter group, a node-classification query can be as simple as this:

```
g.V("movie_1", "movie_2",
 "movie_3").properties("genre").with("Neptune#ml.classification")
```

You can mix vertex properties and classes in a query using the `union()` step:

```
g.with("Neptune#ml.endpoint","node-classification-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V( "movie_1", "movie_2", "movie_3" )
 .union(
   properties("genre").with("Neptune#ml.classification"),
   properties("genre")
 )
```

You can also make an unbounded query such as this:

```
g.with("Neptune#ml.endpoint","node-classification-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V()
 .properties("genre").with("Neptune#ml.classification")
```

You can retrieve the node classes together with vertices using the `select()` step together with the `as()` step:

```
g.with("Neptune#ml.endpoint","node-classification-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V( "movie_1", "movie_2", "movie_3" ).as("vertex")
 .properties("genre").with("Neptune#ml.classification").as("properties")
 .select("vertex","properties")
```

You can also filter on node classes, as illustrated in these examples:

```
g.with("Neptune#ml.endpoint", "node-classification-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V( "movie_1", "movie_2", "movie_3" )
 .properties("genre").with("Neptune#ml.classification")
 .has(value, "Horror")

g.with("Neptune#ml.endpoint","node-classification-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V( "movie_1", "movie_2", "movie_3" )
 .properties("genre").with("Neptune#ml.classification")
 .has(value, P.eq("Action"))

g.with("Neptune#ml.endpoint","node-classification-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V( "movie_1", "movie_2", "movie_3" )
 .properties("genre").with("Neptune#ml.classification")
 .has(value, P.within("Action", "Horror"))
```

You can get a node classification confidence score using the `Neptune#ml.score` predicate:

```
 g.with("Neptune#ml.endpoint","node-classification-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V( "movie_1", "movie_2", "movie_3" )
 .properties("genre", "Neptune#ml.score").with("Neptune#ml.classification")
```

The response would look like this:

```
==>vp[genre->Action]
==>vp[Neptune#ml.score->0.01234567]
==>vp[genre->Crime]
==>vp[Neptune#ml.score->0.543210]
==>vp[genre->Comedy]
==>vp[Neptune#ml.score->0.10101]
```

Using inductive inference in a node classification query

Supposing you were to add a new node to an existing graph, in a Jupyter notebook, like this:

```
%%gremlin
g.addV('label1').property(id,'101').as('newV')
 .V('1').as('oldV1')
 .V('2').as('oldV2')
 .addE('eLabel1').from('newV').to('oldV1')
 .addE('eLabel2').from('oldV2').to('newV')
```

You could then use an inductive inference query to get a genre and confidence score that reflected the new node:

```
%%gremlin
g.with("Neptune#ml.endpoint", "nc-ep")
 .with("Neptune#ml.iamRoleArn", "arn:aws:iam::123456789012:role/NeptuneMLRole")
 .V('101').properties("genre", "Neptune#ml.score")
 .with("Neptune#ml.classification")
 .with("Neptune#ml.inductiveInference")
```

If you ran the query several times, however, you might get somewhat different results:

```
# First time
==>vp[genre->Action]
==>vp[Neptune#ml.score->0.12345678]

# Second time
==>vp[genre->Action]
==>vp[Neptune#ml.score->0.21365921]
```

You could make the same query deterministic:

```
%%gremlin
g.with("Neptune#ml.endpoint", "nc-ep")
 .with("Neptune#ml.iamRoleArn", "arn:aws:iam::123456789012:role/NeptuneMLRole")
 .V('101').properties("genre", "Neptune#ml.score")
 .with("Neptune#ml.classification")
 .with("Neptune#ml.inductiveInference")
 .with("Neptune#ml.deterministic")
```

In that case, the results would be roughly the same every time:

```
# First time
==>vp[genre->Action]
==>vp[Neptune#ml.score->0.12345678]
# Second time
==>vp[genre->Action]
==>vp[Neptune#ml.score->0.12345678]
```

Gremlin node regression queries in Neptune ML

Node regression is similar to node classification, except that the value inferred from the regression model for each node is numeric. You can use the same Gremlin queries for node regression as for node classification except for the following differences:

- Again, in Neptune ML, nodes refer to vertices.
- The `properties()` step takes the form, `properties().with("Neptune#ml.regression")`instead of `properties().with("Neptune#ml.classification")`.
- The `"Neptune#ml.limit"` and `"Neptune#ml.threshold"` predicates are not applicable.
- When you filter on the value, you have to specify a numeric value.

Here is a sample vertex classification query:

```
g.with("Neptune#ml.endpoint","node-regression-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn", "arn:aws:iam::0123456789:role/sagemaker-role")
 .V("movie_1","movie_2","movie_3")
 .properties("revenue").with("Neptune#ml.regression")
```

You can filter on the value inferred using a regression model, as illustrated in the following examples:

```
g.with("Neptune#ml.endpoint","node-regression-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V("movie_1","movie_2","movie_3")
 .properties("revenue").with("Neptune#ml.regression")
 .value().is(P.gte(1600000))

g.with("Neptune#ml.endpoint","node-regression-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V("movie_1","movie_2","movie_3")
 .properties("revenue").with("Neptune#ml.regression")
 .hasValue(P.lte(1600000D))
```

Using inductive inference in a node regression query

Supposing you were to add a new node to an existing graph, in a Jupyter notebook, like this:

```
%%gremlin
g.addV('label1').property(id,'101').as('newV')
 .V('1').as('oldV1')
 .V('2').as('oldV2')
 .addE('eLabel1').from('newV').to('oldV1')
 .addE('eLabel2').from('oldV2').to('newV')
```

You could then use an inductive inference query to get a rating that took into account the new node:

```
%%gremlin
g.with("Neptune#ml.endpoint", "nr-ep")
 .with("Neptune#ml.iamRoleArn", "arn:aws:iam::123456789012:role/NeptuneMLRole")
 .V('101').properties("rating")
 .with("Neptune#ml.regression")
 .with("Neptune#ml.inductiveInference")
```

Because the query is not deterministic, it might return somewhat different results if you run it several times, based on the neighborhood:

```
# First time
==>vp[rating->9.1]

# Second time
==>vp[rating->8.9]
```

If you need more consistent results, you could make the query deterministic:

```
%%gremlin
g.with("Neptune#ml.endpoint", "nc-ep")
 .with("Neptune#ml.iamRoleArn", "arn:aws:iam::123456789012:role/NeptuneMLRole")
 .V('101').properties("rating")
 .with("Neptune#ml.regression")
 .with("Neptune#ml.inductiveInference")
 .with("Neptune#ml.deterministic")
```

Now the results will be roughly the same every time:

```
# First time
==>vp[rating->9.1]

# Second time
==>vp[rating->9.1]
```

Gremlin edge classification queries in Neptune ML

For Gremlin edge classification in Neptune ML:

- The model is trained on one property of the edges. The set of unique values of this property is referred to as a set of classes.
- The class or categorical property value of an edge can be inferred from the edge classification model, which is useful when this property is not already attached to the edge.
- In order to fetch one or more classes from an edge classification model, you need to use the `with()` step with the predicate, `"Neptune#ml.classification"` to configure the `properties()` step. The output format is similar to what you would expect if those were edge properties.

> **Note**
>
> Edge classification only works with string property values. That means that numerical property values such as `0` or `1` are not supported, although the string equivalents `"0"` and `"1"` are. Similarly, the Boolean property values `true` and `false` don't work, but `"true"` and `"false"` do.

Here is an example of an edge classification query that requests a confidence score using the `Neptune#ml.score` predicate:

```
g.with("Neptune#ml.endpoint","edge-classification-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .E("relationship_1","relationship_2","relationship_3")
 .properties("knows_by", "Neptune#ml.score").with("Neptune#ml.classification")
```

The response would look like this:

```
==>p[knows_by->"Family"]
==>p[Neptune#ml.score->0.01234567]
==>p[knows_by->"Friends"]
==>p[Neptune#ml.score->0.543210]
==>p[knows_by->"Colleagues"]
==>p[Neptune#ml.score->0.10101]
```

Syntax of a Gremlin edge classification query

For a simple graph where `User` is the head and tail node, and `Relationship` is the edge that connects them, an example edge classification query is:

```
g.with("Neptune#ml.endpoint","edge-classification-social-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .E("relationship_1","relationship_2","relationship_3")
 .properties("knows_by").with("Neptune#ml.classification")
```

The output of this query would look something like the following:

```
==>p[knows_by->"Family"]
==>p[knows_by->"Friends"]
==>p[knows_by->"Colleagues"]
```

In the query above, the `E()` and `properties()` steps are used as follows:

- The `E()` step contains the set of edges for which you want to fetch the classes from the edge-classification model:

  ```
  .E("relationship_1","relationship_2","relationship_3")
  ```

- The `properties()` step contains the key on which the model was trained, and has `.with("Neptune#ml.classification")` to indicate that this is an edge classification ML inference query.

Multiple property keys are not currently supported in a `properties().with("Neptune#ml.classification")` step. For example, the following query results in an exception being thrown:

```
g.with("Neptune#ml.endpoint","edge-classification-social-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .E("relationship_1","relationship_2","relationship_3")
 .properties("knows_by", "other_label").with("Neptune#ml.classification")
```

For specific error messages, see List of exceptions for Neptune ML Gremlin inference queries.

A `properties().with("Neptune#ml.classification")` step can be used in combination with any of the following steps:

- `value()`
- `value().is()`
- `hasValue()`
- `has(value,"")`
- `key()`
- `key().is()`
- `hasKey()`
- `has(key,"")`
- `path()`

Using inductive inference in an edge classification query

Supposing you were to add a new edge to an existing graph, in a Jupyter notebook, like this:

```
%%gremlin
g.V('1').as('fromV')
.V('2').as('toV')
```

```
.addE('eLabel1').from('fromV').to('toV').property(id, 'e101')
```

You could then use an inductive inference query to get a scale that took into account the new edge:

```
%%gremlin
g.with("Neptune#ml.endpoint", "ec-ep")
 .with("Neptune#ml.iamRoleArn", "arn:aws:iam::123456789012:role/NeptuneMLRole")
 .E('e101').properties("scale", "Neptune#ml.score")
 .with("Neptune#ml.classification")
 .with("Neptune#ml.inductiveInference")
```

Because the query is not deterministic, the results would vary somewhat if you run it multiple times, based on the random neighborhood:

```
# First time
==>vp[scale->Like]
==>vp[Neptune#ml.score->0.12345678]

# Second time
==>vp[scale->Like]
==>vp[Neptune#ml.score->0.21365921]
```

If you need more consistent results, you could make the query deterministic:

```
%%gremlin
g.with("Neptune#ml.endpoint", "ec-ep")
 .with("Neptune#ml.iamRoleArn", "arn:aws:iam::123456789012:role/NeptuneMLRole")
 .E('e101').properties("scale", "Neptune#ml.score")
 .with("Neptune#ml.classification")
 .with("Neptune#ml.inductiveInference")
 .with("Neptune#ml.deterministic")
```

Now the results will be more or less the same every time you run the query:

```
# First time
==>vp[scale->Like]
==>vp[Neptune#ml.score->0.12345678]

# Second time
==>vp[scale->Like]
==>vp[Neptune#ml.score->0.12345678]
```

Gremlin edge regression queries in Neptune ML

Edge regression is similar to edge classification, except that the value inferred from the ML model is numeric. For edge regression, Neptune ML supports the same queries as for classification.

Key points to note are:

- You need to use the ML predicate `"Neptune#ml.regression"` to configure the `properties()` step for this use-case.
- The `"Neptune#ml.limit"` and `"Neptune#ml.threshold"` predicates are not applicable in this use-case.
- For filtering on the value, you need to specify the value as numerical.

Syntax of a Gremlin edge regression query

For a simple graph where `User` is the head node, `Movie` is the tail node, and `Rated` is the edge that connects them, here is an example edge regression query that finds the numeric rating value, referred to as score here, for the edge `Rated`:

```
g.with("Neptune#ml.endpoint","edge-regression-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .E("rating_1","rating_2","rating_3")
 .properties("score").with("Neptune#ml.regression")
```

You can also filter on a value inferred from the ML regression model. For the existing `Rated` edges (from `User` to `Movie`) identified by `"rating_1"`, `"rating_2"`, and `"rating_3"`, where the edge property `Score` is not present for these ratings, you can use a query like following to infer `Score` for the edges where it is greater than or equal to 9:

```
g.with("Neptune#ml.endpoint","edge-regression-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .E("rating_1","rating_2","rating_3")
 .properties("score").with("Neptune#ml.regression")
 .value().is(P.gte(9))
```

Using inductive inference in an edge regression query

Supposing you were to add a new edge to an existing graph, in a Jupyter notebook, like this:

```
%%gremlin
g.V('1').as('fromV')
.V('2').as('toV')
.addE('eLabel1').from('fromV').to('toV').property(id, 'e101')
```

You could then use an inductive inference query to get a score that took into account the new edge:

```
%%gremlin
g.with("Neptune#ml.endpoint", "er-ep")
 .with("Neptune#ml.iamRoleArn", "arn:aws:iam::123456789012:role/NeptuneMLRole")
 .E('e101').properties("score")
 .with("Neptune#ml.regression")
 .with("Neptune#ml.inductiveInference")
```

Because the query is not deterministic, the results would vary somewhat if you run it multiple times, based on the random neighborhood:

```
# First time
==>ep[score->96]

# Second time
==>ep[score->91]
```

If you need more consistent results, you could make the query deterministic:

```
%%gremlin
g.with("Neptune#ml.endpoint", "er-ep")
 .with("Neptune#ml.iamRoleArn", "arn:aws:iam::123456789012:role/NeptuneMLRole")
 .E('e101').properties("score")
 .with("Neptune#ml.regression")
 .with("Neptune#ml.inductiveInference")
 .with("Neptune#ml.deterministic")
```

Now the results will be more or less the same every time you run the query:

```
# First time
==>ep[score->96]

# Second time
```

```
==>ep[score->96]
```

Gremlin link prediction queries using link-prediction models in Neptune ML

Link-prediction models can solve problems such as the following:

- **Head-node prediction**: Given a vertex and an edge type, what vertices is that vertex likely to link from?
- **Tail-node prediction**: Given a vertex and an edge label, what vertices is that vertex likely to link to?

> **Note**
>
> Edge prediction is not yet supported in Neptune ML.

For the examples below, consider a simple graph with the vertices `User` and `Movie` that are linked by the edge `Rated`.

Here is a sample head-node prediction query, used to predict the top five users most likely to rate the movies, `"movie_1"`, `"movie_2"`, and `"movie_3"`:

```
g.with("Neptune#ml.endpoint","node-prediction-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .with("Neptune#ml.limit", 5)
 .V("movie_1", "movie_2", "movie_3")
 .in("rated").with("Neptune#ml.prediction").hasLabel("user")
```

Here is a similar one for tail-node prediction, used to predict the top five movies that user `"user_1"` is likely to rate:

```
g.with("Neptune#ml.endpoint","node-prediction-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V("user_1")
 .out("rated").with("Neptune#ml.prediction").hasLabel("movie")
```

Both the edge label and the predicted vertex label are required. If either is omitted, an exception is thrown. For example, the following query without a predicted vertex label throws an exception:

```
g.with("Neptune#ml.endpoint","node-prediction-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V("user_1")
 .out("rated").with("Neptune#ml.prediction")
```

Similarly, the following query without an edge label throws an exception:

```
g.with("Neptune#ml.endpoint","node-prediction-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V("user_1")
 .out().with("Neptune#ml.prediction").hasLabel("movie")
```

For the specific error messages that these exceptions return, see the list of Neptune ML exceptions.

Other link-prediction queries

You can use the `select()` step with the `as()` step to output the predicted vertices together with the input vertices:

```
g.with("Neptune#ml.endpoint","node-prediction-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V("movie_1").as("source")
 .in("rated").with("Neptune#ml.prediction").hasLabel("user").as("target")
 .select("source","target")

g.with("Neptune#ml.endpoint","node-prediction-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V("user_1").as("source")
 .out("rated").with("Neptune#ml.prediction").hasLabel("movie").as("target")
 .select("source","target")
```

You can make unbounded queries, like these:

```
g.with("Neptune#ml.endpoint","node-prediction-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V("user_1")
 .out("rated").with("Neptune#ml.prediction").hasLabel("movie")

g.with("Neptune#ml.endpoint","node-prediction-movie-lens-endpoint")
 .with("Neptune#ml.iamRoleArn","arn:aws:iam::0123456789:role/sagemaker-role")
 .V("movie_1")
```

```
 .in("rated").with("Neptune#ml.prediction").hasLabel("user")
```

Using inductive inference in a link prediction query

Supposing you were to add a new node to an existing graph, in a Jupyter notebook, like this:

```
%%gremlin
g.addV('label1').property(id,'101').as('newV1')
 .addV('label2').property(id,'102').as('newV2')
 .V('1').as('oldV1')
 .V('2').as('oldV2')
 .addE('eLabel1').from('newV1').to('oldV1')
 .addE('eLabel2').from('oldV2').to('newV2')
```

You could then use an inductive inference query to predict the head node, taking into account the new node:

```
%%gremlin
g.with("Neptune#ml.endpoint", "lp-ep")
 .with("Neptune#ml.iamRoleArn", "arn:aws:iam::123456789012:role/NeptuneMLRole")
 .V('101').out("eLabel1")
 .with("Neptune#ml.prediction")
 .with("Neptune#ml.inductiveInference")
 .hasLabel("label2")
```

Result:

```
==>V[2]
```

Similarly, you could use an inductive inference query to predict the tail node, taking into account the new node:

```
%%gremlin
g.with("Neptune#ml.endpoint", "lp-ep")
 .with("Neptune#ml.iamRoleArn", "arn:aws:iam::123456789012:role/NeptuneMLRole")
 .V('102').in("eLabel2")
 .with("Neptune#ml.prediction")
 .with("Neptune#ml.inductiveInference")
 .hasLabel("label1")
```

Result:

```
==>V[1]
```

List of exceptions for Neptune ML Gremlin inference queries

This is a comprehensive list of exceptions that can occur when executing Neptune ML Gremlin inference queries. These exceptions cover a range of issues, from problems with the specified IAM role or endpoint, to unsupported Gremlin steps and limitations on the number of ML inference queries per query. Each entry includes a detailed message describing the issue.

- **`BadRequestException`** – The credentials for the supplied role cannot be loaded.

  Message: `Unable to load credentials for role: the specified IAM Role ARN.`
- **`BadRequestException`** – The specified IAM role is not authorized to invoke the SageMaker AI endpoint.

  Message: `User: the specified IAM Role ARN is not authorized to perform: sagemaker:InvokeEndpoint on resource: the specified endpoint.`
- **`BadRequestException`** – The specified endpoint does not exist.

  Message: `Endpoint the specified endpoint not found.`
- **`InternalFailureException`** – Unable to fetch Neptune ML real-time inductive inference metadata from Amazon S3.

  Message: `Unable to fetch Neptune ML - Real-Time Inductive Inference metadata from S3. Check the permissions of the S3 bucket or if the Neptune instance can connect to S3.`
- **`InternalFailureException`** – Neptune ML cannot find the metadata file for real-time inductive inference in Amazon S3.

  Message: `Neptune ML cannot find the metadata file for Real-Time Inductive Inference in S3.`
- **`InvalidParameterException`** – The specified endpoint is not syntactically valid.

  Message: `Invalid endpoint provided for external service query.`
- **`InvalidParameterException`** – The specified SageMaker execution IAM Role ARN is not syntactically valid.

  Message: `Invalid IAM role ARN provided for external service query.`

- **`InvalidParameterException`** – Multiple property keys are specified in the `properties()` step in a query.

  Message: `ML inference queries are currently supported for one property key.`
- **`InvalidParameterException`** – Multiple edge labels are specified in a query.

  Message: `ML inference are currently supported only with one edge label.`
- **`InvalidParameterException`** – Multiple vertex label constraints are specified in a query.

  Message: `ML inference are currently supported only with one vertex label constraint.`
- **`InvalidParameterException`** – Both `Neptune#ml.classification` and `Neptune#ml.regression` predicates are present in the same query.

  Message: `Both regression and classification ML predicates cannot be specified in the query.`
- **`InvalidParameterException`** – More than one edge label was specified in the `in()` or `out()` step in a link-prediction query.

  Message: `ML inference are currently supported only with one edge label.`
- **`InvalidParameterException`** – More than one property key was specified with Neptune#ml.score.

  Message: `Neptune ML inference queries are currently supported for one property key and one Neptune#ml.score property key.`
- **`MissingParameterException`** – The endpoint was not specified in the query or as a DB cluster parameter.

  Message: `No endpoint provided for external service query.`
- **MissingParameterException** – The SageMaker AI execution IAM role was not specified in the query or as a DB cluster parameter.

  Message: `No IAM role ARN provided for external service query.`
- **`MissingParameterException`** – The property key is missing from the `properties()` step in a query.

  Message: `Property key needs to be specified using properties() step for ML inference queries.`

- **`MissingParameterException`** – No edge label was specified in the `in()` or `out()` step of a link-prediction query.

  Message: `Edge label needs to be specified while using in() or out() step for ML inference queries.`
- **`MissingParameterException`** – No property key was specified with Neptune#ml.score.

  Message: `Property key needs to be specified along with Neptune#ml.score property key while using the properties() step for Neptune ML inference queries.`
- **`UnsupportedOperationException`** – The `both()` step is used in a link-prediction query.

  Message: `ML inference queries are currently not supported with both() step.`
- **`UnsupportedOperationException`** – No predicted vertex label was specified in the `has()` step with the `in()` or `out()` step in a link-prediction query.

  Message: `Predicted vertex label needs to be specified using has() step for ML inference queries.`
- **`UnsupportedOperationException`** – Gremlin ML inductive inference queries are not currently supported with unoptimized steps.

  Message: `Neptune ML - Real-Time Inductive Inference queries are currently not supported with Gremlin steps which are not optimized for Neptune. Check the Neptune User Guide for a list of Neptune-optimized steps.`
- **`UnsupportedOperationException`** – Neptune ML inference queries are not currently supported inside a `repeat` step.

  Message: `Neptune ML inference queries are currently not supported inside a repeat step.`
- **`UnsupportedOperationException`** – No more than one Neptune ML inference query is currently supported per Gremlin query.

  Message: `Neptune ML inference queries are currently supported only with one ML inference query per gremlin query.`

SPARQL inference queries in Neptune ML

Neptune ML maps the RDF graph into a property graph to model the ML Task. Currently, it supports the following use-cases:

- **Object classification** – Predicts the categorical feature of an object.
- **Object regression** – Predicts a numerical property of an object.
- **Object prediction** – Predicts an object given a subject and a relationship.
- **Subject prediction** – Predicts a subject given an object and a relationship.

> **Note**
>
> Neptune ML does not support subject classification and regression use cases with SPARQL.

Neptune ML predicates used in SPARQL inference queries

The following predicates are used with SPARQL inference:

`neptune-ml:timeout` predicate

Specifies the timeout for connection with the remote server. Should not be confused with the query request timeout, which is the maximum amount of time the server can take to satisfy a request.

Note that if the query timeout occurs before the service timeout specified by the `neptune-ml:timeout` predicate occurs, the service connection is canceled too.

`neptune-ml:outputClass` predicate

The `neptune-ml:outputClass` predicate is only used to define the class of the predicted object for object prediction or predicted subject for subect prediction.

`neptune-ml:outputScore` predicate

The `neptune-ml:outputScore` predicate is a positive number that represents the likelihood that the output of a machine learning model is correct.

`neptune-ml:modelType` predicate

The `neptune-ml:modelType` predicate specifies the type of machine learning model being trained:

- `OBJECT_CLASSIFICATION`
- `OBJECT_REGRESSION`
- `OBJECT_PREDICTION`
- `SUBJECT_PREDICTION`

`neptune-ml:input` predicate

The `neptune-ml:input` predicate refers to the list of URIs used as inputs for Neptune ML.

`neptune-ml:output` predicate

The `neptune-ml:output` predicate refers to the list of binding sets where Neptune ML returns results.

`neptune-ml:predicate` predicate

The `neptune-ml:predicate` predicate is used differently depending on the task being performed:

- For **object or subject prediction**: defines the type of predicate (the edge or relationship type).
- For **object classification and regression**: defines the literal (property) we want to predict.

`neptune-ml:batchSize` predicate

The `neptune-ml:batchSize` specifies the input size for the remote service call.

SPARQL object classification examples

For SPARQL object classification in Neptune ML, the model is trained on one of the predicate values. This is useful where that predicate is not already present with a given subject.

Only categorical predicate values can be inferred using the object classification model.

The following query seeks to predict the <http://www.example.org/team> predicate value for all the inputs of type `foaf:Person`:

```
SELECT * WHERE { ?input a foaf:Person .
  SERVICE neptune-ml:inference {
    neptune-ml:config neptune-ml:modelType 'OBJECT_CLASSIFICATION' ;
                      neptune-ml:input ?input ;
                      neptune-ml:predicate <http://www.example.org/team> ;
                      neptune-ml:output ?output .
  }
}
```

This query can be customized as follows:

```
SELECT * WHERE { ?input a foaf:Person .
  SERVICE neptune-ml:inference {
    neptune-ml:config neptune-ml:endpoint 'node-prediction-account-balance-endpoint' ;
                      neptune-ml:iamRoleArn 'arn:aws:iam::0123456789:role/sagemaker-
role' ;

                      neptune-ml:batchSize "40"^^xsd:integer ;
                      neptune-ml:timeout "1000"^^xsd:integer ;

                      neptune-ml:modelType 'OBJECT_CLASSIFICATION' ;
                      neptune-ml:input ?input ;
                      neptune-ml:predicate <http://www.example.org/team> ;
                      neptune-ml:output ?output .
  }
}
```

SPARQL object regression examples

Object regression is similar to object classification, except that a numerical predicate value inferred from the regression model for each node. You can use the same SPARQL queries for object regression as for object classification with the exception that `the Neptune#ml.limit` and `Neptune#ml.threshold` predicates are not applicable.

The following query seeks to predict the <http://www.example.org/accountbalance> predicate value for all the inputs of type `foaf:Person`:

```
SELECT * WHERE { ?input a foaf:Person .
  SERVICE neptune-ml:inference {
    neptune-ml:config neptune-ml:modelType 'OBJECT_REGRESSION' ;
                      neptune-ml:input ?input ;
```

```
                      neptune-ml:predicate <http://www.example.org/accountbalance> ;
                      neptune-ml:output ?output .
  }
}
```

This query can be customized as follows:

```
SELECT * WHERE { ?input a foaf:Person .
  SERVICE neptune-ml:inference {
    neptune-ml:config neptune-ml:endpoint 'node-prediction-account-balance-endpoint' ;
                      neptune-ml:iamRoleArn 'arn:aws:iam::0123456789:role/sagemaker-
role' ;

                      neptune-ml:batchSize "40"^^xsd:integer ;
                      neptune-ml:timeout "1000"^^xsd:integer ;

                      neptune-ml:modelType 'OBJECT_REGRESSION' ;
                      neptune-ml:input ?input ;
                      neptune-ml:predicate <http://www.example.org/accountbalance> ;
                      neptune-ml:output ?output .
  }
}
```

SPARQL object prediction example

Object prediction predicts the object value for a given subject and predicate.

The following object-prediction query seeks to predict what movie the input of type `foaf:Person` would like:

```
?x a foaf:Person .
?x   <http://www.example.org/likes> ?m .
?m a <http://www.example.org/movie> .

## Query
SELECT * WHERE { ?input a foaf:Person .
  SERVICE neptune-ml:inference {
    neptune-ml:config neptune-ml:modelType 'OBJECT_PREDICTION' ;
                      neptune-ml:input ?input ;
                      neptune-ml:predicate <http://www.example.org/likes> ;
                      neptune-ml:output ?output ;
                      neptune-ml:outputClass <http://www.example.org/movie> .
```

```
  }
}
```

The query itself could be customized as follows:

```
SELECT * WHERE { ?input a foaf:Person .
  SERVICE neptune-ml:inference {
    neptune-ml:config neptune-ml:endpoint 'node-prediction-user-movie-prediction-
endpoint' ;
                      neptune-ml:iamRoleArn 'arn:aws:iam::0123456789:role/sagemaker-
role' ;

                      neptune-ml:limit "5"^^xsd:integer ;
                      neptune-ml:batchSize "40"^^xsd:integer ;
                      neptune-ml:threshold "0.1"^^xsd:double ;
                      neptune-ml:timeout "1000"^^xsd:integer ;
                      neptune-ml:outputScore ?score ;

                      neptune-ml:modelType 'OBJECT_PREDICTION' ;
                      neptune-ml:input ?input ;
                      neptune-ml:predicate <http://www.example.org/likes> ;
                      neptune-ml:output ?output ;
                      neptune-ml:outputClass <http://www.example.org/movie> .
  }
}
```

SPARQL subject prediction example

Subject prediction predicts the subject given a predicate and an object.

For example, the following query predicts who (of type `foaf:User`) will watch a given movie:

```
SELECT * WHERE { ?input (a foaf:Movie) .
    SERVICE neptune-ml:inference {
        neptune-ml:config neptune-ml:modelType 'SUBJECT_PREDICTION' ;
                          neptune-ml:input ?input ;
                          neptune-ml:predicate <http://aws.amazon.com/neptune/csv2rdf/
object_Property/rated> ;
                          neptune-ml:output ?output ;
                          neptune-ml:outputClass <http://aws.amazon.com/neptune/
csv2rdf/class/User> ;        }
}
```

List of exceptions for Neptune ML SPARQL inference queries

- **BadRequestException** – *Message*: The SERVICE http://aws.amazon.com/neptune/vocab/v01/services/ml#inference expects at least 1 value for the parameter *(parameter name)*, found zero.
- **BadRequestException** – *Message*: The SERVICE http://aws.amazon.com/neptune/vocab/v01/services/ml#inference expects at most 1 value for the parameter *(parameter name)*, found *(a number)* values.
- **BadRequestException** – *Message*: Invalid predicate *(predicate name)* provided for external service http://aws.amazon.com/neptune/vocab/v01/services/ml#inference query.
- **BadRequestException** – *Message*: The SERVICE http://aws.amazon.com/neptune/vocab/v01/services/ml#inference expects the predicate *(predicate name)* to be defined.
- **BadRequestException** – *Message*: The SERVICE http://aws.amazon.com/neptune/vocab/v01/services/ml#inference expects the value of (parameter) *(parameter name)* to be a variable, found: *(type)*"
- **BadRequestException** – *Message*: The SERVICE http://aws.amazon.com/neptune/vocab/v01/services/ml#inference expects the input *(parameter name)* to be a constant, found: *(type)*.
- **BadRequestException** – *Message*: The SERVICE http://aws.amazon.com/neptune/vocab/v01/services/ml#inference is expected to return only 1 value.
- **BadRequestException** – *Message*: "The SERVICE http://aws.amazon.com/neptune/vocab/v01/services/ml#inference only allows StatementPatternNodes.
- **BadRequestException** – *Message*: The SERVICE http://aws.amazon.com/neptune/vocab/v01/services/ml#inference does not allow the predicate *(predicate name)*.
- **BadRequestException** – *Message*: The SERVICE http://aws.amazon.com/neptune/vocab/v01/services/ml#inference predicates cannot be variables, found: *(type)*.
- **BadRequestException** – *Message*: The SERVICE http://aws.amazon.com/neptune/vocab/v01/services/ml#inference predicates are expected to be part of the namespace *(namespace name)*, found: *(namespace name)*.

Neptune ML management API reference

Contents

- Data processing using the dataprocessing command
  - Creating a data-processing job using the Neptune ML dataprocessing command
  - Getting the status of a data-processing job using the Neptune ML dataprocessing command
  - Stopping a data-processing job using the Neptune ML dataprocessing command
  - Listing active data-processing jobs using the Neptune ML dataprocessing command
- Model training using the modeltraining command
  - Creating a model-training job using the Neptune ML modeltraining command
  - Getting the status of a model-training job using the Neptune ML modeltraining command
  - Stopping a model-training job using the Neptune ML modeltraining command
  - Listing active model-training jobs using the Neptune ML modeltraining command
- Model transform using the modeltransform command
  - Creating a model-transform job using the Neptune ML modeltransform command
  - Getting the status of a model-transform job using the Neptune ML modeltransform command
  - Stopping a model-transform job using the Neptune ML modeltransform command
  - Listing active model-transform jobs using the Neptune ML modeltransform command
- Managing inference endpoints using the endpoints command
  - Creating an inference endpoint using the Neptune ML endpoints command
  - Getting the status of an inference endpoint using the Neptune ML endpoints command
  - Deleting an instance endpoint using the Neptune ML endpoints command
  - Listing inference endpoints using the Neptune ML endpoints command
- Neptune ML management API errors and exceptions

Data processing using the `dataprocessing` command

You use the Neptune ML `dataprocessing` command to create a data processing job, check its status, stop it, or list all active data-processing jobs.

Creating a data-processing job using the Neptune ML `dataprocessing` command

A typical Neptune ML `dataprocessing` command for creating a new job looks like this:

AWS CLI

```
aws neptunedata start-ml-data-processing-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --input-data-s3-location "s3:////(S3 bucket name)/(path to your input folder)" \
  --id "(a job ID for the new job)" \
  --processed-data-s3-location "s3://(S3 bucket name)/(path to your output folder)"
```

For more information, see start-ml-data-processing-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_data_processing_job(
    inputDataS3Location='s3://(S3 bucket name)/(path to your input folder)',
    id='(a job ID for the new job)',
    processedDataS3Location='s3://(S3 bucket name)/(path to your output folder)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/dataprocessing \
  --region us-east-1 \
  --service neptune-db \
```

```
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "inputDataS3Location" : "s3:////(S3 bucket name)/(path to your input folder)",
        "id" : "(a job ID for the new job)",
        "processedDataS3Location" : "s3://(S3 bucket name)/(path to your output
 folder)"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/dataprocessing \
  -H 'Content-Type: application/json' \
  -d '{
        "inputDataS3Location" : "s3:////(S3 bucket name)/(path to your input folder)",
        "id" : "(a job ID for the new job)",
        "processedDataS3Location" : "s3://(S3 bucket name)/(path to your output
 folder)"
      }'
```

A command to initiate incremental re-processing looks like this:

AWS CLI

```
aws neptunedata start-ml-data-processing-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --input-data-s3-location "s3://(S3 bucket name)/(path to your input folder)" \
  --id "(a job ID for this job)" \
  --processed-data-s3-location "s3://(S3 bucket name)/(path to your output folder)"
 \
  --previous-data-processing-job-id "(the job ID of a previously completed job to
 update)"
```

For more information, see start-ml-data-processing-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_data_processing_job(
    inputDataS3Location='s3://(S3 bucket name)/(path to your input folder)',
    id='(a job ID for this job)',
    processedDataS3Location='s3://(S3 bucket name)/(path to your output folder)',
    previousDataProcessingJobId='(the job ID of a previously completed job to
 update)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/dataprocessing \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "inputDataS3Location" : "s3://(S3 bucket name)/(path to your input folder)",
        "id" : "(a job ID for this job)",
        "processedDataS3Location" : "s3://(S3 bucket name)/(path to your output
 folder)",
        "previousDataProcessingJobId" : "(the job ID of a previously completed job
 to update)"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/dataprocessing \
  -H 'Content-Type: application/json' \
  -d '{
        "inputDataS3Location" : "s3:////(S3 bucket name)/(path to your input folder)",
        "id" : "(a job ID for this job)",
        "processedDataS3Location" : "s3:////(S3 bucket name)/(path to your output
 folder)",
        "previousDataProcessingJobId" : "(the job ID of a previously completed job
 to update)"
      }'
```

Parameters for `dataprocessing` job creation

- **`id`** – (*Optional*) A unique identifier for the new job.

  Type: string. *Default*: An autogenerated UUID.
- **`previousDataProcessingJobId`** – (*Optional*) The job ID of a completed data processing job run on an earlier version of the data.

  Type: string. *Default*: *none*.

  Note: Use this for incremental data processing, to update the model when graph data has changed (but not when data has been deleted).
- **`inputDataS3Location`** – (*Required*) The URI of the Amazon S3 location where you want SageMaker AI to download the data needed to run the data processing job.

  Type: string.
- **`processedDataS3Location`** – (*Required*) The URI of the Amazon S3 location where you want SageMaker AI to save the results of a data processing job.

  Type: string.
- **`sagemakerIamRoleArn`** – (*Optional*) The ARN of an IAM role for SageMaker AI execution.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.
- **`neptuneIamRoleArn`** – (*Optional*) The Amazon Resource Name (ARN) of an IAM role that SageMaker AI can assume to perform tasks on your behalf.

Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.

- **`processingInstanceType`** – (*Optional*) The type of ML instance used during data processing. Its memory should be large enough to hold the processed dataset.

  Type: string. *Default*: the smallest `ml.r5` type whose memory is ten times larger than the size of the exported graph data on disk.

  Note: Neptune ML can select the instance type automatically. See Selecting an instance for data processing.

- **`processingInstanceVolumeSizeInGB`** – (*Optional*) The disk volume size of the processing instance. Both input data and processed data are stored on disk, so the volume size must be large enough to hold both data sets.

  Type: integer. *Default*: `0`.

  Note: If not specified or 0, Neptune ML chooses the volume size automatically based on the data size.

- **`processingTimeOutInSeconds`** – (*Optional*) Timeout in seconds for the data processing job.

  Type: integer. *Default*: `86,400` (1 day).

- **`modelType`** – (*Optional*) One of the two model types that Neptune ML currently supports: heterogeneous graph models (`heterogeneous`), and knowledge graph (`kge`).

  Type: string. *Default*: *none*.

  Note: If not specified, Neptune ML chooses the model type automatically based on the data.

- **`configFileName`** – (*Optional*) A data specification file that describes how to load the exported graph data for training. The file is automatically generated by the Neptune export toolkit.

  Type: string. *Default*: `training-data-configuration.json`.

- **`subnets`** – (*Optional*) The IDs of the subnets in the Neptune VPC.

  Type: list of strings. *Default*: *none*.

- **`securityGroupIds`** – (*Optional*) The VPC security group IDs.

  Type: list of strings. *Default*: *none*.

- **`volumeEncryptionKMSKey`** – (*Optional*) The AWS Key Management Service (AWS KMS) key that SageMaker AI uses to encrypt data on the storage volume attached to the ML compute instances that run the processing job.

  Type: string. *Default*: *none*.
- **`enableInterContainerTrafficEncryption`** – (*Optional*) Enable or disable inter-container traffic encryption in training or hyper-parameter tuning jobs.

  Type: boolean. *Default*: *True*.

  > **Note**
  >
  > The `enableInterContainerTrafficEncryption` parameter is only available in engine release 1.2.0.2.R3.

- **`s3OutputEncryptionKMSKey`** – (*Optional*) The AWS Key Management Service (AWS KMS) key that SageMaker AI uses to encrypt the output of the training job.

  Type: string. *Default*: *none*.

Getting the status of a data-processing job using the Neptune ML `dataprocessing` command

A sample Neptune ML `dataprocessing` command for the status of a job looks like this:

AWS CLI

```
aws neptunedata get-ml-data-processing-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(the job ID)"
```

For more information, see get-ml-data-processing-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
```

```
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_ml_data_processing_job(
    id='(the job ID)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/dataprocessing/(the job ID) \
  --region us-east-1 \
  --service neptune-db \
  -X GET
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -s \
  "https://your-neptune-endpoint:port/ml/dataprocessing/(the job ID)" \
  | python -m json.tool
```

Parameters for `dataprocessing` job status

- **`id`** – (*Required*) The unique identifier of the data-processing job.

  Type: string.

- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role that provides Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.

Stopping a data-processing job using the Neptune ML `dataprocessing` command

A sample Neptune ML `dataprocessing` command for stopping a job looks like this:

AWS CLI

```
aws neptunedata cancel-ml-data-processing-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(the job ID)"
```

To also clean up Amazon S3 artifacts:

```
aws neptunedata cancel-ml-data-processing-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(the job ID)" \
  --clean
```

For more information, see cancel-ml-data-processing-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.cancel_ml_data_processing_job(
    id='(the job ID)',
    clean=True
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/dataprocessing/(the job ID) \
  --region us-east-1 \
```

```
  --service neptune-db \
  -X DELETE
```

To also clean up Amazon S3 artifacts:

```
awscurl "https://your-neptune-endpoint:port/ml/dataprocessing/(the job ID)?
clean=true" \
  --region us-east-1 \
  --service neptune-db \
  -X DELETE
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -s \
  -X DELETE "https://your-neptune-endpoint:port/ml/dataprocessing/(the job ID)"
```

Or this:

```
curl -s \
  -X DELETE "https://your-neptune-endpoint:port/ml/dataprocessing/(the job ID)?
clean=true"
```

Parameters for `dataprocessing` stop job

- **`id`** – (*Required*) The unique identifier of the data-processing job.

  Type: string.
- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role that provides Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.
- **`clean`** – (*Optional*) This flag specifies that all Amazon S3 artifacts should be deleted when the job is stopped.

Type: Boolean. *Default*: FALSE.

Listing active data-processing jobs using the Neptune ML `dataprocessing` command

A sample Neptune ML `dataprocessing` command for listing active jobs looks like this:

AWS CLI

```
aws neptunedata list-ml-data-processing-jobs \
  --endpoint-url https://your-neptune-endpoint:port
```

To limit the number of results:

```
aws neptunedata list-ml-data-processing-jobs \
  --endpoint-url https://your-neptune-endpoint:port \
  --max-items 3
```

For more information, see list-ml-data-processing-jobs in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.list_ml_data_processing_jobs(
    maxItems=3
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/dataprocessing \
```

```
  --region us-east-1 \
  --service neptune-db \
  -X GET
```

To limit the number of results:

```
awscurl "https://your-neptune-endpoint:port/ml/dataprocessing?maxItems=3" \
  --region us-east-1 \
  --service neptune-db \
  -X GET
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -s "https://your-neptune-endpoint:port/ml/dataprocessing"
```

Or this:

```
curl -s "https://your-neptune-endpoint:port/ml/dataprocessing?maxItems=3"
```

Parameters for `dataprocessing` list jobs

- **`maxItems`** – (*Optional*) The maximum number of items to return.

  Type: integer. *Default*: `10`. *Maximum allowed value*: `1024`.
- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role that provides Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.

Model training using the `modeltraining` command

You use the Neptune ML `modeltraining` command to create a model training job, check its status, stop it, or list all active model-training jobs.

Creating a model-training job using the Neptune ML `modeltraining` command

A Neptune ML `modeltraining` command for creating a completely new job looks like this:

AWS CLI

```
aws neptunedata start-ml-model-training-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-training job ID)" \
  --data-processing-job-id "(the data-processing job-id of a completed job)" \
  --train-model-s3-location "s3://(your S3 bucket)/neptune-model-graph-autotrainer"
```

For more information, see start-ml-model-training-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_model_training_job(
    id='(a unique model-training job ID)',
    dataProcessingJobId='(the data-processing job-id of a completed job)',
    trainModelS3Location='s3://(your S3 bucket)/neptune-model-graph-autotrainer'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltraining \
  --region us-east-1 \
  --service neptune-db \
```

```
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3:///(your S3 bucket)/neptune-model-graph-
autotrainer"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltraining \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3:///(your S3 bucket)/neptune-model-graph-
autotrainer"
      }'
```

A Neptune ML `modeltraining` command for creating an update job for incremental model training looks like this:

AWS CLI

```
aws neptunedata start-ml-model-training-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-training job ID)" \
  --data-processing-job-id "(the data-processing job-id of a completed job)" \
  --train-model-s3-location "s3:///(your S3 bucket)/neptune-model-graph-autotrainer"
 \
  --previous-model-training-job-id "(the job ID of a completed model-training job to
 update)"
```

For more information, see start-ml-model-training-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_model_training_job(
    id='(a unique model-training job ID)',
    dataProcessingJobId='(the data-processing job-id of a completed job)',
    trainModelS3Location='s3://(your S3 bucket)/neptune-model-graph-autotrainer',
    previousModelTrainingJobId='(the job ID of a completed model-training job to
 update)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltraining \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3://(your S3 bucket)/neptune-model-graph-
autotrainer",
        "previousModelTrainingJobId" : "(the job ID of a completed model-training
 job to update)"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltraining \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3://(your S3 bucket)/neptune-model-graph-
autotrainer",
        "previousModelTrainingJobId" : "(the job ID of a completed model-training
 job to update)"
      }'
```

A Neptune ML `modeltraining` command for creating a new job with user provided custom model implementation looks like:

AWS CLI

```
aws neptunedata start-ml-model-training-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-training job ID)" \
  --data-processing-job-id "(the data-processing job-id of a completed job)" \
  --train-model-s3-location "s3://(your Amazon S3 bucket)/neptune-model-graph-
autotrainer" \
  --model-name "custom" \
  --custom-model-training-parameters '{
    "sourceS3DirectoryPath": "s3://(your Amazon S3 bucket)/(path to your Python
 module)",
    "trainingEntryPointScript": "(your training script entry-point name in the
 Python module)",
    "transformEntryPointScript": "(your transform script entry-point name in the
 Python module)"
  }'
```

For more information, see start-ml-model-training-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_model_training_job(
    id='(a unique model-training job ID)',
    dataProcessingJobId='(the data-processing job-id of a completed job)',
    trainModelS3Location='s3://(your Amazon S3 bucket)/neptune-model-graph-
autotrainer',
    modelName='custom',
    customModelTrainingParameters={
        'sourceS3DirectoryPath': 's3://(your Amazon S3 bucket)/(path to your Python
 module)',
        'trainingEntryPointScript': '(your training script entry-point name in the
 Python module)',
        'transformEntryPointScript': '(your transform script entry-point name in the
 Python module)'
    }
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltraining \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3://(your Amazon S3 bucket)/neptune-model-graph-
autotrainer",
        "modelName": "custom",
```

```
        "customModelTrainingParameters" : {
          "sourceS3DirectoryPath": "s3:////(your Amazon S3 bucket)/(path to your
 Python module)",
          "trainingEntryPointScript": "(your training script entry-point name in the
 Python module)",
          "transformEntryPointScript": "(your transform script entry-point name in
 the Python module)"
        }
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltraining \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-training job ID)",
        "dataProcessingJobId" : "(the data-processing job-id of a completed job)",
        "trainModelS3Location" : "s3://(your Amazon S3 bucket)/neptune-model-graph-
autotrainer",
        "modelName": "custom",
        "customModelTrainingParameters" : {
          "sourceS3DirectoryPath": "s3://(your Amazon S3 bucket)/(path to your
 Python module)",
          "trainingEntryPointScript": "(your training script entry-point name in the
 Python module)",
          "transformEntryPointScript": "(your transform script entry-point name in
 the Python module)"
        }
      }'
```

Parameters for `modeltraining` job creation

- **`id`** – (*Optional*) A unique identifier for the new job.

Type: string. *Default*: An autogenerated UUID.

- **`dataProcessingJobId`** – (*Required*) The job Id of the completed data-processing job that has created the data that the training will work with.

  Type: string.

- **`trainModelS3Location`** – (*Required*) The location in Amazon S3 where the model artifacts are to be stored.

  Type: string.

- **`previousModelTrainingJobId`** – (*Optional*) The job ID of a completed model-training job that you want to update incrementally based on updated data.

  Type: string. *Default*: *none*.

- **`sagemakerIamRoleArn`** – (*Optional*) The ARN of an IAM role for SageMaker AI execution.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.

- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role that provides Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.

- **`modelName`** – (*Optional*) The model type for training. By default the ML model is automatically based on the `modelType` used in data processing, but you can specify a different model type here.

  Type: string. *Default*: `rgcn` for heterogeneous graphs and `kge` for knowledge graphs. *Valid values*: For heterogeneous graphs: `rgcn`. For `kge` graphs: `transe`, `distmult`, or `rotate`. For a custom model implementation: `custom`.

- **`baseProcessingInstanceType`** – (*Optional*) The type of ML instance used in preparing and managing training of ML models.

  Type: string. *Note*: This is a CPU instance chosen based on memory requirements for processing the training data and model. See Selecting an instance for model training and model transform.

- **`trainingInstanceType`** – (*Optional*) The type of ML instance used for model training. All Neptune ML models support CPU, GPU, and multiGPU training.

  Type: string. *Default*: `ml.p3.2xlarge`.

Note: Choosing the right instance type for training depends on the task type, graph size, and your budget. See Selecting an instance for model training and model transform.

- **`trainingInstanceVolumeSizeInGB`** – (*Optional*) The disk volume size of the training instance. Both input data and the output model are stored on disk, so the volume size must be large enough to hold both data sets.

  Type: integer. *Default*: `0`.

  Note: If not specified or 0, Neptune ML selects a disk volume size based on the recommendation generated in the data processing step. See Selecting an instance for model training and model transform.

- **`trainingTimeOutInSeconds`** – (*Optional*) Timeout in seconds for the training job.

  Type: integer. *Default*: `86,400` (1 day).

- **`maxHPONumberOfTrainingJobs`** – Maximum total number of training jobs to start for the hyperparameter tuning job.

  Type: integer. *Default*: 2.

  Note: Neptune ML automatically tunes the hyper-parameters of the machine learning model. To obtain a model that performs well, use at least 10 jobs (in other words, set `maxHPONumberOfTrainingJobs` to 10). In general, the more tuning runs, the better the results.

- **`maxHPOParallelTrainingJobs`** – Maximum number of parallel training jobs to start for the hyperparameter tuning job.

  Type: integer. *Default*: 2.

  Note: The number of parallel jobs you can run is limited by the available resources on your training instance.

- **`subnets`** – (*Optional*) The IDs of the subnets in the Neptune VPC.

  Type: list of strings. *Default*: *none*.

- **`securityGroupIds`** – (*Optional*) The VPC security group IDs.

  Type: list of strings. *Default*: *none*.

- **`volumeEncryptionKMSKey`** – (*Optional*) The AWS Key Management Service (AWS KMS) key that SageMaker AI uses to encrypt data on the storage volume attached to the ML compute instances that run the training job.

  Type: string. *Default*: *none*.

- **`s3OutputEncryptionKMSKey`** – (*Optional*) The AWS Key Management Service (AWS KMS) key that SageMaker AI uses to encrypt the output of the processing job.

  Type: string. *Default*: *none*.

- **`enableInterContainerTrafficEncryption`** – (*Optional*) Enable or disable inter-container traffic encryption in training or hyper-parameter tuning jobs.

  Type: boolean. *Default*: *True*.

  > **Note**
  >
  > The `enableInterContainerTrafficEncryption` parameter is only available in engine release 1.2.0.2.R3.

- **`enableManagedSpotTraining`** – (*Optional*) Optimizes the cost of training machine learning models by using Amazon Elastic Compute Cloud spot instances. For more information, see Managed Spot Training in Amazon SageMaker.

  Type: Boolean. *Default*: *False*.

- **`customModelTrainingParameters`** – (*Optional*) The configuration for custom model training. This is a JSON object with the following fields:

  - **`sourceS3DirectoryPath`** – (*Required*) The path to the Amazon S3 location where the Python module implementing your model is located. This must point to a valid existing Amazon S3 location that contains, at a minimum, a training script, a transform script, and a `model-hpo-configuration.json` file.

  - **`trainingEntryPointScript`** – (*Optional*) The name of the entry point in your module of a script that performs model training and takes hyperparameters as command-line arguments, including fixed hyperparameters.

    Default: `training.py`.

  - **`transformEntryPointScript`** – (*Optional*) The name of the entry point in your module of a script that should be run after the best model from the hyperparameter search has been

identified, to compute the model artifacts necessary for model deployment. It should be able to run with no command-line arguments.

Default: `transform.py`.

- **`maxWaitTime`** – (*Optional*) The maximum time to wait, in seconds, when performing model training using spot instances. Should be greater than `trainingTimeOutInSeconds`.

  Type: integer.

Getting the status of a model-training job using the Neptune ML `modeltraining` command

A sample Neptune ML `modeltraining` command for the status of a job looks like this:

AWS CLI

```
aws neptunedata get-ml-model-training-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(the job ID)"
```

For more information, see get-ml-model-training-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_ml_model_training_job(
    id='(the job ID)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltraining/(the job ID) \
  --region us-east-1 \
  --service neptune-db \
  -X GET
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl -s \
  "https://your-neptune-endpoint:port/ml/modeltraining/(the job ID)" \
  | python -m json.tool
```

Parameters for `modeltraining` job status

- **`id`** – (*Required*) The unique identifier of the model-training job.

  Type: string.
- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role that provides Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.

Stopping a model-training job using the Neptune ML `modeltraining` command

A sample Neptune ML `modeltraining` command for stopping a job looks like this:

AWS CLI

```
aws neptunedata cancel-ml-model-training-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(the job ID)"
```

To also clean up Amazon S3 artifacts:

```
aws neptunedata cancel-ml-model-training-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(the job ID)" \
  --clean
```

For more information, see cancel-ml-model-training-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.cancel_ml_model_training_job(
    id='(the job ID)',
    clean=True
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltraining/(the job ID) \
  --region us-east-1 \
  --service neptune-db \
  -X DELETE
```

To also clean up Amazon S3 artifacts:

```
awscurl "https://your-neptune-endpoint:port/ml/modeltraining/(the job ID)?
clean=true" \
  --region us-east-1 \
  --service neptune-db \
  -X DELETE
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -s \
  -X DELETE "https://your-neptune-endpoint:port/ml/modeltraining/(the job ID)"
```

Or this:

```
curl -s \
  -X DELETE "https://your-neptune-endpoint:port/ml/modeltraining/(the job ID)?
clean=true"
```

Parameters for `modeltraining` stop job

- **`id`** – (*Required*) The unique identifier of the model-training job.

  Type: string.
- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role that provides Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.
- **`clean`** – (*Optional*) This flag specifies that all Amazon S3 artifacts should be deleted when the job is stopped.

  Type: Boolean. *Default*: `FALSE`.

Listing active model-training jobs using the Neptune ML `modeltraining` command

A sample Neptune ML `modeltraining` command for listing active jobs looks like this:

AWS CLI

```
aws neptunedata list-ml-model-training-jobs \
  --endpoint-url https://your-neptune-endpoint:port
```

To limit the number of results:

```
aws neptunedata list-ml-model-training-jobs \
  --endpoint-url https://your-neptune-endpoint:port \
  --max-items 3
```

For more information, see list-ml-model-training-jobs in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.list_ml_model_training_jobs(
    maxItems=3
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltraining \
  --region us-east-1 \
  --service neptune-db \
  -X GET
```

To limit the number of results:

```
awscurl "https://your-neptune-endpoint:port/ml/modeltraining?maxItems=3" \
  --region us-east-1 \
  --service neptune-db \
```

```
  -X GET
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -s "https://your-neptune-endpoint:port/ml/modeltraining" | python -m json.tool
```

Or this:

```
curl -s "https://your-neptune-endpoint:port/ml/modeltraining?maxItems=3" | python -m
 json.tool
```

Parameters for `modeltraining` list jobs

- **`maxItems`** – (*Optional*) The maximum number of items to return.

  Type: integer. *Default*: `10`. *Maximum allowed value*: `1024`.
- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role that provides Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.

Model transform using the `modeltransform` command

You use the Neptune ML `modeltransform` command to create a model transform job, check its status, stop it, or list all active model-transform jobs.

Creating a model-transform job using the Neptune ML `modeltransform` command

A Neptune ML `modeltransform` command for creating an incremental transform job, without model retraining, looks like this:

AWS CLI

```
aws neptunedata start-ml-model-transform-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-transform job ID)" \
  --data-processing-job-id "(the job-id of a completed data-processing job)" \
  --ml-model-training-job-id "(the job-id of a completed model-training job)" \
  --model-transform-output-s3-location "s3://(your S3 bucket)/neptune-model-
transform"
```

For more information, see start-ml-model-transform-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_model_transform_job(
    id='(a unique model-transform job ID)',
    dataProcessingJobId='(the job-id of a completed data-processing job)',
    mlModelTrainingJobId='(the job-id of a completed model-training job)',
    modelTransformOutputS3Location='s3://(your S3 bucket)/neptune-model-transform'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltransform \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-transform job ID)",
        "dataProcessingJobId" : "(the job-id of a completed data-processing job)",
        "mlModelTrainingJobId" : "(the job-id of a completed model-training job)",
        "modelTransformOutputS3Location" : "s3:////(your S3 bucket)/neptune-model-
transform"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltransform \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-transform job ID)",
        "dataProcessingJobId" : "(the job-id of a completed data-processing job)",
        "mlModelTrainingJobId" : "(the job-id of a completed model-training job)",
        "modelTransformOutputS3Location" : "s3:////(your S3 bucket)/neptune-model-
transform"
      }'
```

A Neptune ML `modeltransform` command for creating a job from a completed SageMaker AI training job looks like this:

AWS CLI

```
aws neptunedata start-ml-model-transform-job \
```

```
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-transform job ID)" \
  --training-job-name "(name of a completed SageMaker training job)" \
  --model-transform-output-s3-location "s3://(your S3 bucket)/neptune-model-
transform"
```

For more information, see start-ml-model-transform-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_model_transform_job(
    id='(a unique model-transform job ID)',
    trainingJobName='(name of a completed SageMaker training job)',
    modelTransformOutputS3Location='s3://(your S3 bucket)/neptune-model-transform'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltransform \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-transform job ID)",
        "trainingJobName" : "(name of a completed SageMaker training job)",
        "modelTransformOutputS3Location" : "s3://(your S3 bucket)/neptune-model-
transform"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltransform \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-transform job ID)",
        "trainingJobName" : "(name of a completed SageMaker training job)",
        "modelTransformOutputS3Location" : "s3:////(your S3 bucket)/neptune-model-
transform"
      }'
```

A Neptune ML `modeltransform` command for creating a job that uses a custom model implementation looks like:

AWS CLI

```
aws neptunedata start-ml-model-transform-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique model-transform job ID)" \
  --training-job-name "(name of a completed SageMaker training job)" \
  --model-transform-output-s3-location "s3://(your Amazon S3 bucket)/neptune-model-
transform/" \
  --custom-model-transform-parameters '{
    "sourceS3DirectoryPath": "s3://(your Amazon S3 bucket)/(path to your Python
 module)",
    "transformEntryPointScript": "(your transform script entry-point name in the
 Python module)"
  }'
```

For more information, see start-ml-model-transform-job in the AWS CLI Command Reference.

SDK

```
import boto3
```

```
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.start_ml_model_transform_job(
    id='(a unique model-transform job ID)',
    trainingJobName='(name of a completed SageMaker training job)',
    modelTransformOutputS3Location='s3:///(your Amazon S3 bucket)/neptune-model-
transform/',
    customModelTransformParameters={
        'sourceS3DirectoryPath': 's3:///(your Amazon S3 bucket)/(path to your Python
 module)',
        'transformEntryPointScript': '(your transform script entry-point name in the
 Python module)'
    }
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltransform \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-transform job ID)",
        "trainingJobName" : "(name of a completed SageMaker training job)",
        "modelTransformOutputS3Location" : "s3:///(your Amazon S3 bucket)/neptune-
model-transform/",
        "customModelTransformParameters" : {
          "sourceS3DirectoryPath": "s3:///(your Amazon S3 bucket)/(path to your
 Python module)",
          "transformEntryPointScript": "(your transform script entry-point name in
 the Python module)"
        }
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/modeltransform \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique model-transform job ID)",
        "trainingJobName" : "(name of a completed SageMaker training job)",
        "modelTransformOutputS3Location" : "s3:////(your Amazon S3 bucket)/neptune-
model-transform/",
        "customModelTransformParameters" : {
          "sourceS3DirectoryPath": "s3://(your Amazon S3 bucket)/(path to your
 Python module)",
          "transformEntryPointScript": "(your transform script entry-point name in
 the Python module)"
        }
      }'
```

Parameters for `modeltransform` job creation

- **`id`** – (*Optional*) A unique identifier for the new job.

  Type: string. *Default*: An autogenerated UUID.

- **`dataProcessingJobId`** – The job Id of a completed data-processing job.

  Type: string.

  Note: You must include either both `dataProcessingJobId` and `mlModelTrainingJobId`, or `trainingJobName`.

- **`mlModelTrainingJobId`** – The job Id of a completed model-training job.

  Type: string.

Note: You must include either both `dataProcessingJobId` and `mlModelTrainingJobId`, or `trainingJobName`.

- **`trainingJobName`** – The name of a completed SageMaker AI training job.

  Type: string.

  Note: You must include either both the `dataProcessingJobId` and the `mlModelTrainingJobId` parameters, or the `trainingJobName` parameter.

- **`sagemakerIamRoleArn`** – (*Optional*) The ARN of an IAM role for SageMaker AI execution.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.

- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role that provides Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.

- **`customModelTransformParameters`** – (*Optional*) Configuration information for a model transform using a custom model. The `customModelTransformParameters` object contains the following fields, which must have values compatible with the saved model parameters from the training job:

  - **`sourceS3DirectoryPath`** – (*Required*) The path to the Amazon S3 location where the Python module implementing your model is located. This must point to a valid existing Amazon S3 location that contains, at a minimum, a training script, a transform script, and a `model-hpo-configuration.json` file.

  - **`transformEntryPointScript`** – (*Optional*) The name of the entry point in your module of a script that should be run after the best model from the hyperparameter search has been identified, to compute the model artifacts necessary for model deployment. It should be able to run with no command-line arguments.

    Default: `transform.py`.

- **`baseProcessingInstanceType`** – (*Optional*) The type of ML instance used in preparing and managing training of ML models.

  Type: string. *Note*: This is a CPU instance chosen based on memory requirements for processing the transform data and model. See Selecting an instance for model training and model transform.

- **`baseProcessingInstanceVolumeSizeInGB`** – (*Optional*) The disk volume size of the training instance. Both input data and the output model are stored on disk, so the volume size must be large enough to hold both data sets.

  Type: integer. *Default*: `0`.

  Note: If not specified or 0, Neptune ML selects a disk volume size based on the recommendation generated in the data processing step. See Selecting an instance for model training and model transform.

- **`subnets`** – (*Optional*) The IDs of the subnets in the Neptune VPC.

  Type: list of strings. *Default*: *none*.

- **`securityGroupIds`** – (*Optional*) The VPC security group IDs.

  Type: list of strings. *Default*: *none*.

- **`volumeEncryptionKMSKey`** – (*Optional*) The AWS Key Management Service (AWS KMS) key that SageMaker AI uses to encrypt data on the storage volume attached to the ML compute instances that run the transform job.

  Type: string. *Default*: *none*.

- **`enableInterContainerTrafficEncryption`** – (*Optional*) Enable or disable inter-container traffic encryption in training or hyper-parameter tuning jobs.

  Type: boolean. *Default*: *True*.

  > **Note**
  >
  > The `enableInterContainerTrafficEncryption` parameter is only available in engine release 1.2.0.2.R3.

- **`s3OutputEncryptionKMSKey`** – (*Optional*) The AWS Key Management Service (AWS KMS) key that SageMaker AI uses to encrypt the output of the processing job.

  Type: string. *Default*: *none*.

Getting the status of a model-transform job using the Neptune ML `modeltransform` command

A sample Neptune ML `modeltransform` command for the status of a job looks like this:

AWS CLI

```
aws neptunedata get-ml-model-transform-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(the job ID)"
```

For more information, see get-ml-model-transform-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_ml_model_transform_job(
    id='(the job ID)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltransform/(the job ID) \
  --region us-east-1 \
  --service neptune-db \
  -X GET
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -s \
  "https://your-neptune-endpoint:port/ml/modeltransform/(the job ID)" \
  | python -m json.tool
```

Parameters for `modeltransform` job status

- **`id`** – (*Required*) The unique identifier of the model-transform job.

  Type: string.
- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role that provides Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.

Stopping a model-transform job using the Neptune ML `modeltransform` command

A sample Neptune ML `modeltransform` command for stopping a job looks like this:

AWS CLI

```
aws neptunedata cancel-ml-model-transform-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(the job ID)"
```

To also clean up Amazon S3 artifacts:

```
aws neptunedata cancel-ml-model-transform-job \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(the job ID)" \
  --clean
```

For more information, see cancel-ml-model-transform-job in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config
```

```
client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.cancel_ml_model_transform_job(
    id='(the job ID)',
    clean=True
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltransform/(the job ID) \
  --region us-east-1 \
  --service neptune-db \
  -X DELETE
```

To also clean up Amazon S3 artifacts:

```
awscurl "https://your-neptune-endpoint:port/ml/modeltransform/(the job ID)?
clean=true" \
  --region us-east-1 \
  --service neptune-db \
  -X DELETE
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl -s \
  -X DELETE "https://your-neptune-endpoint:port/ml/modeltransform/(the job ID)"
```

Or this:

```
curl -s \
  -X DELETE "https://your-neptune-endpoint:port/ml/modeltransform/(the job ID)?
clean=true"
```

Parameters for `modeltransform` stop job

- **`id`** – (*Required*) The unique identifier of the model-transform job.

  Type: string.
- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role that provides Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.
- **`clean`** – (*Optional*) This flag specifies that all Amazon S3 artifacts should be deleted when the job is stopped.

  Type: Boolean. *Default*: FALSE.

Listing active model-transform jobs using the Neptune ML `modeltransform` command

A sample Neptune ML `modeltransform` command for listing active jobs looks like this:

AWS CLI

```
aws neptunedata list-ml-model-transform-jobs \
  --endpoint-url https://your-neptune-endpoint:port
```

To limit the number of results:

```
aws neptunedata list-ml-model-transform-jobs \
  --endpoint-url https://your-neptune-endpoint:port \
  --max-items 3
```

For more information, see list-ml-model-transform-jobs in the AWS CLI Command Reference.

SDK

```
import boto3
```

```
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.list_ml_model_transform_jobs(
    maxItems=3
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/modeltransform \
  --region us-east-1 \
  --service neptune-db \
  -X GET
```

To limit the number of results:

```
awscurl "https://your-neptune-endpoint:port/ml/modeltransform?maxItems=3" \
  --region us-east-1 \
  --service neptune-db \
  -X GET
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl -s "https://your-neptune-endpoint:port/ml/modeltransform" | python -m json.tool
```

Or this:

```
curl -s "https://your-neptune-endpoint:port/ml/modeltransform?maxItems=3" | python -
m json.tool
```

Parameters for `modeltransform` list jobs

- **`maxItems`** – (*Optional*) The maximum number of items to return.

  Type: integer. *Default*: `10`. *Maximum allowed value*: `1024`.
- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role that provides Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will occur.

Managing inference endpoints using the `endpoints` command

You use the Neptune ML `endpoints` command to create an inference endpoint, check its status, delete it, or list existing inference endpoints.

Creating an inference endpoint using the Neptune ML `endpoints` command

A Neptune ML `endpoints` command for creating an inference endpoint from a model created by a training job looks like this:

AWS CLI

```
aws neptunedata create-ml-endpoint \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique ID for the new endpoint)" \
  --ml-model-training-job-id "(the model-training job-id of a completed job)"
```

For more information, see create-ml-endpoint in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.create_ml_endpoint(
    id='(a unique ID for the new endpoint)',
    mlModelTrainingJobId='(the model-training job-id of a completed job)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/endpoints \
  --region us-east-1 \
```

```
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique ID for the new endpoint)",
        "mlModelTrainingJobId": "(the model-training job-id of a completed job)"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/endpoints \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique ID for the new endpoint)",
        "mlModelTrainingJobId": "(the model-training job-id of a completed job)"
      }'
```

A Neptune ML `endpoints` command for updating an existing inference endpoint from a model created by a training job looks like this:

AWS CLI

```
aws neptunedata create-ml-endpoint \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique ID for the new endpoint)" \
  --update \
  --ml-model-training-job-id "(the model-training job-id of a completed job)"
```

For more information, see create-ml-endpoint in the AWS CLI Command Reference.

SDK

```
import boto3
```

```
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.create_ml_endpoint(
    id='(a unique ID for the new endpoint)',
    update=True,
    mlModelTrainingJobId='(the model-training job-id of a completed job)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/endpoints \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique ID for the new endpoint)",
        "update" : "true",
        "mlModelTrainingJobId": "(the model-training job-id of a completed job)"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/endpoints \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique ID for the new endpoint)",
```

```
        "update" : "true",
        "mlModelTrainingJobId": "(the model-training job-id of a completed job)"
      }'
```

A Neptune ML `endpoints` command for creating an inference endpoint from a model created by a model-transform job looks like this:

AWS CLI

```
aws neptunedata create-ml-endpoint \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique ID for the new endpoint)" \
  --ml-model-transform-job-id "(the model-transform job-id of a completed job)"
```

For more information, see create-ml-endpoint in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.create_ml_endpoint(
    id='(a unique ID for the new endpoint)',
    mlModelTransformJobId='(the model-transform job-id of a completed job)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/endpoints \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
```

```
        "id" : "(a unique ID for the new endpoint)",
        "mlModelTransformJobId": "(the model-transform job-id of a completed job)"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/endpoints \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique ID for the new endpoint)",
        "mlModelTransformJobId": "(the model-transform job-id of a completed job)"
      }'
```

A Neptune ML `endpoints` command for updating an existing inference endpoint from a model created by a model-transform job looks like this:

AWS CLI

```
aws neptunedata create-ml-endpoint \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(a unique ID for the new endpoint)" \
  --update \
  --ml-model-transform-job-id "(the model-transform job-id of a completed job)"
```

For more information, see create-ml-endpoint in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
```

```
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.create_ml_endpoint(
    id='(a unique ID for the new endpoint)',
    update=True,
    mlModelTransformJobId='(the model-transform job-id of a completed job)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/endpoints \
  --region us-east-1 \
  --service neptune-db \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique ID for the new endpoint)",
        "update" : "true",
        "mlModelTransformJobId": "(the model-transform job-id of a completed job)"
      }'
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl \
  -X POST https://your-neptune-endpoint:port/ml/endpoints \
  -H 'Content-Type: application/json' \
  -d '{
        "id" : "(a unique ID for the new endpoint)",
        "update" : "true",
        "mlModelTransformJobId": "(the model-transform job-id of a completed job)"
      }'
```

Parameters for `endpoints` inference endpoint creation

- **`id`** – (*Optional*) A unique identifier for the new inference endpoint.

  Type: string. *Default*: An autogenerated timestamped name.
- **`mlModelTrainingJobId`** – The job Id of the completed model-training job that has created the model that the inference endpoint will point to.

  Type: string.

  Note: You must supply either the `mlModelTrainingJobId` or the `mlModelTransformJobId`.
- **`mlModelTransformJobId`** – The job Id of the completed model-transform job.

  Type: string.

  Note: You must supply either the `mlModelTrainingJobId` or the `mlModelTransformJobId`.
- **`update`** – (*Optional*) If present, this parameter indicates that this is an update request.

  Type: Boolean. *Default*: `false`

  Note: You must supply either the `mlModelTrainingJobId` or the `mlModelTransformJobId`.
- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role providing Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will be thrown.
- **`modelName`** – (*Optional*) Model type for training. By default the ML model is automatically based on the `modelType` used in data processing, but you can specify a different model type here.

  Type: string. *Default*: `rgcn` for heterogeneous graphs and `kge` for knowledge graphs. *Valid values*: For heterogeneous graphs: `rgcn`. For knowledge graphs: `kge`, `transe`, `distmult`, or `rotate`.
- **`instanceType`** – (*Optional*) The type of ML instance used for online servicing.

  Type: string. *Default*: `ml.m5.xlarge`.

  Note: Choosing the ML instance for an inference endpoint depends on the task type, the graph size, and your budget. See Selecting an instance for an inference endpoint.

- **`instanceCount`** – (*Optional*) The minimum number of Amazon EC2 instances to deploy to an endpoint for prediction.

  Type: integer. *Default*: 1.
- **`volumeEncryptionKMSKey`** – (*Optional*) The AWS Key Management Service (AWS KMS) key that SageMaker AI uses to encrypt data on the storage volume attached to the ML compute instance(s) that run the endpoints.

  Type: string. *Default*: *none*.

Getting the status of an inference endpoint using the Neptune ML `endpoints` command

A sample Neptune ML `endpoints` command for the status of an instance endpoint looks like this:

AWS CLI

```
aws neptunedata get-ml-endpoint \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(the inference endpoint ID)"
```

For more information, see get-ml-endpoint in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.get_ml_endpoint(
    id='(the inference endpoint ID)'
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/endpoints/(the inference endpoint ID)
 \
  --region us-east-1 \
  --service neptune-db \
  -X GET
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl -s \
  "https://your-neptune-endpoint:port/ml/endpoints/(the inference endpoint ID)" \
  | python -m json.tool
```

Parameters for `endpoints` instance-endpoint status

- **`id`** – (*Required*) The unique identifier of the inference endpoint.

  Type: string.
- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role providing Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will be thrown.

Deleting an instance endpoint using the Neptune ML `endpoints` command

A sample Neptune ML `endpoints` command for deleting an instance endpoint looks like this:

AWS CLI

```
aws neptunedata delete-ml-endpoint \
  --endpoint-url https://your-neptune-endpoint:port \
```

```
  --id "(the inference endpoint ID)"
```

To also clean up related artifacts:

```
aws neptunedata delete-ml-endpoint \
  --endpoint-url https://your-neptune-endpoint:port \
  --id "(the inference endpoint ID)" \
  --clean
```

For more information, see delete-ml-endpoint in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.delete_ml_endpoint(
    id='(the inference endpoint ID)',
    clean=True
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/endpoints/(the inference endpoint ID)
 \
  --region us-east-1 \
  --service neptune-db \
  -X DELETE
```

To also clean up related artifacts:

```
awscurl "https://your-neptune-endpoint:port/ml/endpoints/(the inference endpoint
 ID)?clean=true" \
  --region us-east-1 \
```

```
  --service neptune-db \
  -X DELETE
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -s \
  -X DELETE "https://your-neptune-endpoint:port/ml/endpoints/(the inference endpoint
 ID)"
```

Or this:

```
curl -s \
  -X DELETE "https://your-neptune-endpoint:port/ml/endpoints/(the inference endpoint
 ID)?clean=true"
```

Parameters for `endpoints` deleting an inference endpoint

- **`id`** – (*Required*) The unique identifier of the inference endpoint.

  Type: string.
- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role providing Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will be thrown.
- **`clean`** – (*Optional*) Indicates that all artifacts related to this endpoint should also be deleted.

  Type: Boolean. *Default*: `FALSE`.

Listing inference endpoints using the Neptune ML `endpoints` command

A Neptune ML `endpoints` command for listing inference endpoints looks like this:

AWS CLI

```
aws neptunedata list-ml-endpoints \
  --endpoint-url https://your-neptune-endpoint:port
```

To limit the number of results:

```
aws neptunedata list-ml-endpoints \
  --endpoint-url https://your-neptune-endpoint:port \
  --max-items 3
```

For more information, see list-ml-endpoints in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-neptune-endpoint:port',
    config=Config(read_timeout=None, retries={'total_max_attempts': 1})
)

response = client.list_ml_endpoints(
    maxItems=3
)

print(response)
```

awscurl

```
awscurl https://your-neptune-endpoint:port/ml/endpoints \
  --region us-east-1 \
  --service neptune-db \
  -X GET
```

To limit the number of results:

```
awscurl "https://your-neptune-endpoint:port/ml/endpoints?maxItems=3" \
  --region us-east-1 \
  --service neptune-db \
```

```
  -X GET
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace *us-east-1* with the Region of your Neptune cluster.

curl

```
curl -s "https://your-neptune-endpoint:port/ml/endpoints" \
  | python -m json.tool
```

Or this:

```
curl -s "https://your-neptune-endpoint:port/ml/endpoints?maxItems=3" \
  | python -m json.tool
```

Parameters for `dataprocessing` list inference endpoints

- **`maxItems`** – (*Optional*) The maximum number of items to return.

  Type: integer. *Default*: `10`. *Maximum allowed value*: `1024`.
- **`neptuneIamRoleArn`** – (*Optional*) The ARN of an IAM role providing Neptune access to SageMaker AI and Amazon S3 resources.

  Type: string. *Note*: This must be listed in your DB cluster parameter group or an error will be thrown.

Neptune ML management API errors and exceptions

All Neptune ML management API exceptions return a 400 HTTP code. After receiving any of these exceptions, the command that generated the exception should not be retried.

- **`MissingParameterException`** – Error message:

  `Required credentials are missing. Please add IAM role to the cluster or pass as a parameter to this request.`
- **`InvalidParameterException`** – Error messages:
  - `Invalid ML instance type.`
  - `Invalid ID provided. ID can be 1-48 alphanumeric characters.`
  - `Invalid ID provided. Must contain only letters, digits, or hyphens.`
  - `Invalid ID provided. Please check whether a resource with the given ID exists.`
  - `Another resource with same ID already exists. Please use a new ID.`
  - `Failed to stop the job because it has already completed or failed.`
- **`BadRequestException`** – Error messages:
  - `Invalid S3 URL or incorrect S3 permissions. Please check your S3 configuration.`
  - `Provided ModelTraining job has not completed.`
  - `Provided SageMaker AI Training job has not completed.`
  - `Provided MLDataProcessing job is not completed.`
  - `Provided MLModelTraining job doesn't exist.`
  - `Provided ModelTransformJob doesn't exist.`
  - `Unable to find SageMaker AI resource. Please check your input.`

Neptune ML limits

- The types of inference currently supported are node classification, node regression, edge classification, edge regression and link prediction (see Neptune ML capabilities).
- The maximum graph size that Neptune ML can support depends on the amount of memory and storage required during data preparation, model training, and inference.
  - The maximum size of memory of a SageMaker AI data-processing instance is 768 GB. As a result, the data-processing stage fails if it needs more than 768 GB of memory.
  - The maximum size of memory of a SageMaker AI training instance is 732 GB. As a result, the training stage fails if it needs more than 732 GB of memory.
- The maximum size of an inference payload for a SageMaker AI endpoint is 6 MiB. As a result, inductive inference fails if the subgraph payload exceeds this size.
- Neptune ML is currently available only in Regions where Neptune and the other services it depends on (such as AWS Lambda, Amazon API Gateway and Amazon SageMaker AI) are all supported.

  There are differences in China (Beijing) and China (Ningxia) having to do with the default use of IAM authentication, as is explained here along with other differences.
- The link prediction inference endpoints launched by Neptune ML currently can only predict possible links with nodes that were present in the graph during training.

  For example, consider a graph with `User` and `Movie` vertices and `Rated` edges. Using a corresponding Neptune ML link-prediction recommendation model, you can add a new user to the graph and have the model predict movies for them, but the model can only recommend movies that were present during model training. Although the `User` node embedding is calculated in real-time using its local subgraph and the GNN model, and can therefore change with time as users rate movies, it's compared to the static, pre-computed movie embeddings for the final recommendation.
- The KGE models supported by Neptune ML only work for link prediction tasks, and the representations are specific to vertices and edge types present in the graph during training. This means that all vertices and edge types referred to in an inference query must have been present in the graph during training. Predictions for new edge types or vertices cannot be made without retraining the model.

SageMaker AI resource limitations

Depending on your activities and resource usage over time, you may encounter error messages saying that you've exceeded your quota (ResourceLimitExceeded). and you need to scale up your SageMaker AI resources, follow the steps in the Request a service quota increase for SageMaker resources procedure on this page to request a quota increase from AWS Support.

SageMaker AI resource names correspond to Neptune ML stages as follows:

- The SageMaker AI `ProcessingJob` is used by Neptune data processing, model training, and model transform jobs.
- The SageMaker AI `HyperParameterTuningJob` is used by Neptune model training jobs.
- The SageMaker AI `TrainingJob` is used by Neptune model training jobs.
- The SageMaker AI `Endpoint` is used by Neptune inference endpoints.

Monitoring Amazon Neptune resources

Amazon Neptune supports various methods for monitoring database performance and usage:

- **Instance status** – Check the health of a Neptune cluster's graph database engine, find out what version of the engine is installed, and obtain other instance-related information using the instance status API.
- **Graph summary API** – The graph summary API lets you quickly get a high-level understanding of your graph data size and content.

> **Note**
>
> Because the graph summary API relies on DFE statistics, it's only available when statistics are enabled, which is not the case on T3 and T4g instance types.

- **Amazon CloudWatch** – Neptune automatically sends metrics to CloudWatch and also supports CloudWatch Alarms. For more information, see the section called "Using CloudWatch".
- **Audit log files** – View, download, or watch database log files using the Neptune console. For more information, see the section called "Audit Logs with Neptune".
- **Publishing logs to Amazon CloudWatch Logs** – You can configure a Neptune DB cluster to publish audit log data to a log group in Amazon CloudWatch Logs. With CloudWatch Logs, you can perform real-time analysis of the log data, use CloudWatch to create alarms and view metrics, and use CloudWatch Logs to store your log records in highly durable storage. See Neptune CloudWatch Logs.
- **AWS CloudTrail** – Neptune supports API logging using CloudTrail. For more information, see the section called "Logging Neptune API Calls with AWS CloudTrail".
- **Events** – View Neptune events in the console, AWS CLI, or API, and subscribe to event notifications using Amazon SNS. For more information, see the section called "Events".
- **Tagging** – Use tags to add metadata to your Neptune resources and track usage based on tags. For more information, see the section called "Tagging Neptune resources".

Topics

- Check the Health Status of a Neptune Instance
- Monitoring Neptune Using Amazon CloudWatch

- Using Audit Logs with Amazon Neptune Clusters
- Publishing Neptune Logs to Amazon CloudWatch Logs
- Using Amazon Neptune slow-query logging
- Logging Amazon Neptune API Calls with AWS CloudTrail
- Amazon Neptune events
- Tagging Amazon Neptune resources

Check the Health Status of a Neptune Instance

Amazon Neptune provides a mechanism to check the status of the graph database on the host. It's also a good way to confirm that you are able to connect to an instance.

To check the health of an instance and get DB cluster status using `curl`:

```
curl -G https://your-neptune-endpoint:port/status
```

Or, starting with engine release 1.2.1.0.R6, you can use the following CLI command instead:

```
aws neptunedata get-engine-status
```

If the instance is healthy, the `status` command returns a JSON object with the following fields:

- **`status`** – Set to `"healthy"` if the instance is not experiencing problems.

  If the instance is recovering from a crash or from being rebooted and there are active transactions running from the latest server shutdown, `status` is set to `"recovery"`.
- **`startTime`** – Set to the UTC time at which the current server process started.
- **`dbEngineVersion`** – Set to the Neptune engine version running on your DB cluster.

  If this engine version has been manually patched since it was released, the version number is prefixed by `"Patch-"`.
- **`role`** – Set to `"reader"` if the instance is a read-replica, or to `"writer"` if the instance is the primary instance.
- **`dfeQueryEngine`** – Set to `"enabled"` if the DFE engine is fully enabled, or to `viaQueryHint` if the DFE engine is only used with queries that have the `useDFE` query hint set to `true` (`viaQueryHint` is the default).

- **`gremlin`** – Contains information about the Gremlin query language available on your cluster. Specifically, it contains a `version` field that specifies the current TinkerPop version being used by the engine.
- **`sparql`** – Contains information about the SPARQL query language available on your cluster. Specifically, it contains a `version` field that specifies the current SPARQL version being used by the engine.
- **`opencypher`** – Contains information about the openCypher query language available on your cluster. Specifically, it contains a `version` field that specifies the current operCypher version being used by the engine.
- **`labMode`** – Contains Lab Mode settings being used by the engine. This is a selective list of Lab Mode settings and not the complete set. Consult the cluster parameter group for the full set in use.
- **`rollingBackTrxCount`** – If there are transactions being rolled back, this field is set to the number of such transactions. If there are none, the field doesn't appear at all.
- **`rollingBackTrxEarliestStartTime`** – Set to the start time of the earliest transaction being rolled back. If no transactions are being rolled back, the field doesn't appear at all.
- **`features`** – Contains status information about the features enabled on your DB cluster:
  - **`lookupCache`** – The current status of the Lookup cache. This field only appears on R5d instance types, since those are the only instances where a lookup cache can exist. The field is a JSON object in the form:

    ```
    "lookupCache": {
      "status": "current lookup cache status"
    }
    ```

    On an R5d instance:

    - If the lookup cache is enabled, the status is listed as `"Available"`.
    - If the lookup cache has been disabled, the status is listed as `"Disabled"`.
    - If the disk limit has been reached on the instance, the status is listed as `"Read Only Mode - Storage Limit Reached"`.
  - **`ResultCache`** – The current status of the Caching query results. This field is a JSON object in the form:

    ```
    "ResultCache": {
      "status": "current results cache status"
    ```

```
}
```

  - If the results cache has been enabled, the status is listed as `"Available"`.
  - If the cache is disabled, the status is listed as `"Disabled"`.
- **`IAMAuthentication`** – Specifies whether or not AWS Identity and Access Management (IAM) authentication has been enabled on your DB cluster:
  - If IAM authentication been enabled, the status is listed as `"enabled"`.
  - If IAM authentication is disabled, the status is listed as `"disabled"`.
- **`Streams`** – Specifies whether or not Neptune streams have been enabled on your DB cluster:
  - If streams are enabled, the status is listed as `"enabled"`.
  - If streams are disabled, the status is listed as `"disabled"`.
- **`AuditLog`** – Equal to `enabled` if audit logs are enabled, or otherwise `disabled`.
- **`SlowQueryLogs`** – Equal to `info` or debug if slow-query logging is enabled, or otherwise `disabled`.
- **`QueryTimeout`** – The value, in milliseconds, of the query timeout.
- **`settings`** – Settings applied to the instance:
  - **`clusterQueryTimeoutInMs`** – The value, in milliseconds, of the query timeout, set for the whole cluster.
  - **`SlowQueryLogsThreshold`** – The value, in milliseconds, of the query timeout, set for the whole cluster.
- **`serverlessConfiguration`** – Serverless settings for a cluster if it is running as serverless:
  - **`minCapacity`** – The smallest size to which a serverless instance in your DB cluster can shrink, in Neptune Capacity Units (NCUs).
  - **`maxCapacity`** – The largest size to which a serverless instance in your DB cluster can grow, in Neptune Capacity Units (NCUs).

Example output from the instance status command

The following is an example of the output from the instance status command, (in this case, run on an R5d instance):

```
{
  'status': 'healthy',
  'startTime': 'Thu Aug 24 21:47:12 UTC 2023',
```

```
  'dbEngineVersion': '1.2.1.0.R4',
  'role': 'writer',
  'dfeQueryEngine': 'viaQueryHint',
  'gremlin': {'version': 'tinkerpop-3.6.2'},
  'sparql': {'version': 'sparql-1.1'},
  'opencypher': {'version': 'Neptune-9.0.20190305-1.0'},
  'labMode': {
    'ObjectIndex': 'disabled',
    'ReadWriteConflictDetection': 'enabled'
  },
  'features': {
    'SlowQueryLogs': 'disabled',
    'ResultCache': {'status': 'disabled'},
    'IAMAuthentication': 'disabled',
    'Streams': 'disabled',
    'AuditLog': 'disabled'
  },
  'settings': {
    'clusterQueryTimeoutInMs': '120000',
    'SlowQueryLogsThreshold': '5000'
  },
  'serverlessConfiguration': {
    'minCapacity': '1.0',
    'maxCapacity': '128.0'
  }
}
```

If there is a problem with the instance, the status command returns the `HTTP 500` error code. If the host is unreachable, the request times out. Ensure that you are accessing the instance from within the virtual private cloud (VPC), and that your security groups allow you access to it.

Monitoring Neptune Using Amazon CloudWatch

Amazon Neptune and Amazon CloudWatch are integrated so that you can gather and analyze performance metrics. You can monitor these metrics using the CloudWatch console, the AWS Command Line Interface (AWS CLI), or the CloudWatch API.

CloudWatch also lets you set alarms so that you can be notified if a metric value breaches a threshold that you specify. You can even set up CloudWatch Events to take corrective action if a breach occurs. For more information about using CloudWatch and alarms, see the CloudWatch Documentation.

Topics

- Viewing CloudWatch Data (Console)
- Viewing CloudWatch Data (AWS CLI)
- Viewing CloudWatch Data (API)
- Using CloudWatch to monitor DB instance performance in Neptune
- Neptune CloudWatch Metrics
- Neptune CloudWatch Dimensions

Viewing CloudWatch Data (Console)

To view CloudWatch data for a Neptune cluster (console)

1. Sign in to the AWS Management Console and open the CloudWatch console at https://eusc-de-east-1.console.amazonaws-eusc.eu/cloudwatch/.
2. In the navigation pane, choose **Metrics**.
3. In the **All Metrics** pane, choose **Neptune**, and then choose **DBClusterIdentifier**.
4. In the metrics pane, review the full list of metrics for your cluster. The available Neptune metric options appear in the **Viewing** list.

To select or deselect an individual metric, in the results pane, select the check box next to the resource name and metric. Graphs showing the metrics for the selected items appear in the graph pane. To learn more about CloudWatch graphs, see Graph Metrics in the *Amazon CloudWatch User Guide*.

Viewing CloudWatch Data (AWS CLI)

To view CloudWatch data for a Neptune cluster (AWS CLI)

1. Install the AWS CLI. For instructions, see the AWS Command Line Interface User Guide.
2. Use the AWS CLI to fetch information. The relevant CloudWatch parameters for Neptune are listed in Neptune CloudWatch Metrics.

   The following example retrieves CloudWatch metrics for the number of Gremlin requests per second for the `gremlin-cluster` cluster.

```
<![CDATA[
aws cloudwatch get-metric-statistics \
    --namespace AWS/Neptune  --metric-name GremlinRequestsPerSec \
    --dimensions Name=DBClusterIdentifier,Value=gremlin-cluster \
    --start-time 2018-03-03T00:00:00Z --end-time 2018-03-04T00:00:00Z \
    --period 60 --statistics=Average
]]>
```

Viewing CloudWatch Data (API)

CloudWatch also supports a `Query` action so that you can request information programmatically. For more information, see the CloudWatch Query API documentation and Amazon CloudWatch API Reference.

When a CloudWatch action requires a parameter that is specific to Neptune monitoring, such as `MetricName`, use the values listed in Neptune CloudWatch Metrics.

The following example shows a low-level CloudWatch request, using the following parameters:

- `Statistics.member.1` = `Average`
- `Dimensions.member.1` = `DBClusterIdentifier=gremlin-cluster`
- `Namespace` = `AWS/Neptune`
- `StartTime` = `2013-11-14T00:00:00Z`
- `EndTime` = `2013-11-16T00:00:00Z`
- `Period` = `60`
- `MetricName` = `GremlinRequestsPerSec`

Here is what the CloudWatch request looks like. However, this is just to show the form of the request; you must construct your own request based on your metrics and timeframe.

```
https://monitoring.amazonaws.eu/
      ?SignatureVersion=2
      &Action=GremlinRequestsPerSec
      &Version=2010-08-01
      &StartTime=2018-03-03T00:00:00
```

```
&EndTime=2018-03-04T00:00:00
&Period=60
&Statistics.member.1=Average
&Dimensions.member.1=DBClusterIdentifier=gremlin-cluster
&Namespace=AWS/Neptune
&MetricName=GremlinRequests
&Timestamp=2018-03-04T17%3A48%3A21.746Z
&AWSAccessKeyId=AWS Access Key ID;
&Signature=signature
```

Using CloudWatch to monitor DB instance performance in Neptune

You can use CloudWatch metrics in Neptune to monitor what is happening on your DB instances and keep track of the query queue length as observed by the database. The following metrics are particularly useful:

- **`CPUUtilization`** – Shows the percentage of CPU utilization.
- **`VolumeWriteIOPs`** – Shows the average number of disk I/O writes to the cluster volume, reported at 5-minute intervals.
- **`MainRequestQueuePendingRequests`** – Shows the number of requests waiting in the input queue pending execution.

You can also find out how many requests are pending on the server by using the Gremlin query status endpoint with the `includeWaiting` parameter. This will give you the status of all waiting queries.

The following indicators can help you adjust your Neptune provisioning and query strategies to improve efficiency and performance:

- Consistent latency, high `CPUUtilization`, high `VolumeWriteIOPs` and low `MainRequestQueuePendingRequests` together show that the server is actively engaged processing concurrent write requests at a sustainable rate, with little I/O wait.
- Consistent latency, low `CPUUtilization`, low `VolumeWriteIOPs` and no `MainRequestQueuePendingRequests` together show that you have excess capacity on the primary DB instance for processing write requests.
- High `CPUUtilization` and high `VolumeWriteIOPs` but variable latency and `MainRequestQueuePendingRequests` together show that you are sending more work than the server can process in a given interval. Consider creating or resizing batch requests so as to do

the same amount of work with less transactional overhead and/or scaling the primary instance up to increase the number of query threads capable of processing write requests concurrently.

- Low `CPUUtilization` with high `VolumeWriteIOPs` mean that query threads are waiting for I/O operations to the storage layer to complete. If you see variable latencies and some increase in `MainRequestQueuePendingRequests`, consider creating or resizing batch requests so as to do the same amount of work with less transactional overhead.

Neptune CloudWatch Metrics

> **Note**
>
> Amazon Neptune sends metrics to CloudWatch only when they have a non-zero value. For all Neptune metrics, the aggregation granularity is 5 minutes.

Topics

- Neptune CloudWatch metrics
- CloudWatch Metrics That Are Now Deprecated in Neptune

Neptune CloudWatch metrics

The following table lists the CloudWatch metrics that Neptune supports.

> **Note**
>
> All cumulative metrics are reset to zero whenever the server restarts, whether for maintenance, a reboot, or recovering from a crash.

Neptune CloudWatch metrics

Metric	Description	Time Interval	Instance Statistic
`BackupRetentionPeriodStorageUsed`	The total amount of backup storage, in bytes, used to support from the		

Metric	Description	Time Interval	Instance Statistic
	Neptune DB cluster's backup retention window. Included in the total reported by the `TotalBackupStorageBilled` metric.		
`BufferCacheHitRatio`	The percentage of requests that are served by the buffer cache. This metric can be useful in diagnosing query latency, because cache misses induce significant latency. If the cache hit ratio is below 99.9, consider upgrading the instance type to cache more data in memory.	1 minute	average
`ClusterReplicaLag`	For a read replica, the amount of lag when replicating updates from the primary instance, in milliseconds.	1 minute	average

Metric	Description	Time Interval	Instance Statistic
`ClusterReplicaLagMaximum`	The maximum amount of lag between the primary instance and each Neptune DB instance in the DB cluster, in milliseconds.	1 minute	max/min
`ClusterReplicaLagMinimum`	The minimum amount of lag between the primary instance and each Neptune DB instance in the DB cluster, in milliseconds.	1 minute	max/min
`CPUCreditBalance`	The number of CPU credits that an instance has accumulated, reported at 5-minute intervals. You can use this metric to determine how long a DB instance can burst beyond its baseline performance level at a given rate.	5 min	average

Metric	Description	Time Interval	Instance Statistic
`CPUCreditUsage`	The number of CPU credits consumed during the specified period, reported at 5-minute intervals. This metric measures the amount of time during which physical CPUs have been used for processing instructions by virtual CPUs allocated to the DB instance.	5 min	average

Metric	Description	Time Interval	Instance Statistic
`CPUSurplusCreditBalance`	The number of surplus credits that have been spent by an unlimited instance when its `CPUCreditBalance` value is zero. The `CPUSurplusCreditBalance` value is paid down by earned CPU credits. If the number of surplus credits exceeds the maximum number of credits that the instance can earn in a 24-hour period, the spent surplus credits above the maximum incur an additional charge. CPU credit metrics are available at a 5-minute frequency only.	5 min	average
`CPUSurplusCreditsCharged`	The number of spent surplus credits that are not paid down by earned CPU credits, and incur an additional charge.	5 min	average
`CPUUtilization`	The percentage of CPU utilization.	1 minute	average/P99

Metric	Description	Time Interval	Instance Statistic
`EngineUptime`	The amount of time that the instance has been running, in seconds.	1 min	average
`FreeableMemory`	The amount of available random access memory, in bytes.	1 min	average
`GremlinClientErrorsPerSec`	Number of client-side errors per second in Gremlin traversals.	1 min	average
`GremlinServerErrorsPerSec`	Number of server-side errors per second in Gremlin traversals.	1 min	average
`GremlinRequestsPerSec`	Number of requests per second to the Gremlin engine.	1 min	average
`GremlinWebSocketOpenConnections`	The number of open WebSocket connections to Neptune.	1 min	sum
`LoaderClientErrorsPerSec`	Number of client-side errors per second from loader requests.	1 min	average
`LoaderRequestsPerSec`	Number of loader requests per second.	1 min	average
`LoaderServerErrorsPerSec`	Number of loader server-side errors per second.	1 min	average

Metric	Description	Time Interval	Instance Statistic
`MainRequestQueuePendingRequests`	The number of requests waiting in the input queue pending execution. Neptune starts throttling requests when they exceed the maximum queue capacity.	1 min	sum

Metric	Description	Time Interval	Instance Statistic
`NCUUtilization`	Only applicable to a Neptune Serverless DB instance or DB cluster. At an instance level, reports a percentage calculated as the number of Neptune capacity units (NCUs) currently being used by the instance in question, divided by the maximum NCU capacity setting for the cluster. An NCU, or Neptune capacity unit, consists of 2 GiB (gibibyte) of memory (RAM), along with associated virtual processor capacity (vCPU) and networking. At a cluster level, `NCUUtilization` reports the percentage of maximum capacity being used by the cluster as a whole.		

Metric	Description	Time Interval	Instance Statistic
`NetworkReceiveThroughput`	The amount of incoming network throughput received from clients by each instance in the Neptune DB cluster, in bytes per second. This throughput does **not** include network traffic between instances in the DB cluster and the cluster volume.	1 min	average
`NetworkThroughput`	The amount of network throughput both received from and transmitted to clients by each instance in the Neptune DB cluster, in bytes per second. This throughput does **not** include network traffic between instances in the DB cluster and the cluster volume.	1 min	average

Metric	Description	Time Interval	Instance Statistic
`NetworkTransmitThroughput`	The amount of outgoing network throughput transmitted to clients by each instance in the Neptune DB cluster, in bytes per second. This throughput does **not** include network traffic between instances in the DB cluster and the cluster volume.	1 min	average
`NumIndexDeletesPerSec`	Number of deletes from individual indexes. Deletes from each index are counted individually. This includes the deletes that may get rolled back if a query encounters an error.	1 min	average
`NumIndexInsertsPerSec`	Number of inserts to individual indexes. Inserts to each index are counted separately. This includes the inserts that may get rolled back if a query encounters an error.	1 min	average

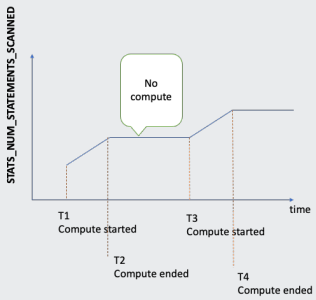
Metric	Description	Time Interval	Instance Statistic
`NumIndexReadsPerSec`	Number of statements scanned from any index. Any access pattern starts with a search on an index and reads of all matching statements. An increase in this metric can cause an increase in query latencies or CPU utilization.	1 min	average
`NumQueuedRequestsPerSec`	The number of requests queued per second.		
`NumResultCacheHit`	Number of Gremlin result cache hits.	1 min	sum
`NumResultCacheMiss`	Number of Gremlin result cache misses.	1 min	sum
`NumTxCommitted`	The number of transactions successfully committed per second.	1 min	sum
`NumTxOpened`	The number of transactions opened on the server per second.	1 min	sum

Metric	Description	Time Interval	Instance Statistic
`NumTxRolledBack`	For write queries, the number of transactions per second rolled back on the server because of errors. For read-only queries, this metric is equal to the number of completed read-only transactions per second.	1 min	sum
`NumUndoPagesPurged`	This metric indicates the number of batches purged. This metric is indicator of progress in purging. The value is `0` for reader instances, and the metric only applies to the writer instance.	1 min	sum
`OpenCypherRequestsPerSec`	Number of requests per second (both HTTPS and Bolt) to the openCypher engine.	1 min	average
`OpenCypherBoltOpenConnections`	The number of open Bolt connections to Neptune.	1 min	sum

Metric	Description	Time Interval	Instance Statistic
`ResultCacheSizeInBytes`	Total estimated size (in bytes) of all cached items in the Gremlin result cache.	1 min	sum
`ResultCacheItemCount`	Number of items in the Gremlin result cache.	1 min	sum
`ResultCacheOldestItemTimestamp`	The timestamp of the oldest item cached in the Gremlin result cache.	1 min	sum
`ResultCacheNewestItemTimestamp`	The timestamp of the newest item cached in the Gremlin result cache.		

Metric	Description	Time Interval	Instance Statistic
`ServerlessDatabaseCapacity`	As an instance-level metric, `ServerlessDatabaseCapacity` reports the current instance capacity of a given Neptune serverless instance, in NCUs. An NCU, or Neptune capacity unit, consists of 2 GiB (gibibyte) of memory (RAM), along with associated virtual processor capacity (vCPU) and networking. At a cluster-level, `ServerlessDatabaseCapacity` reports the average of all the `ServerlessDatabaseCapacity` values of the DB instances in the cluster.		

Metric	Description	Time Interval	Instance Statistic
`SnapshotStorageUsed`	The total amount of backup storage consumed by all snapshots for a Neptune DB cluster outside its backup retention window, in bytes. Included in the total reported by the `TotalBackupStorageBilled` metric.	1 min	sum
`SparqlClientErrorsPerSec`	The number of client-side errors per second in SPARQL queries.	1 min	average
`SparqlRequestsPerSec`	The number of requests per second to the SPARQL engine.	1 min	average
`SparqlServerErrorsPerSec`	The number of SPARQL server errors per second.	1 min	average

Metric	Description	Time Interval	Instance Statistic
`StatsNumStatementsScanned`	The total number of statements scanned for DFE statistics since the server started. Every time statistics computation is triggered, this number increases, but when no computation is happening, it remains static. As a result, if you graph it over time, you can tell when computation happened and when it didn't:  By looking at the slope of the graph in periods where the metric is increasing, you can also tell how quickly the computation was going.	1 min	sum

Metric	Description	Time Interval	Instance Statistic
	If there is no such metric, it means that the statistics feature is disabled on your DB cluster, or that the engine version you're running doesn't have the statistics feature. If the metric value is zero, it means that no statistics computati on has occurred.		
`StorageNetworkReceiveThroughput`	The amount of network throughpu t received from the storage subsystem by each instance in the Neptune DB cluster.	1 min	average
`StorageNetworkThroughput`	The amount of network throughpu t received from and sent to the storage subsystem by each instance in the Neptune DB cluster.	1 min	average
`StorageNetworkTransmitThroughput`	The amount of network throughpu t sent to the storage subsystem by each instance in the Neptune DB cluster.	1 min	average

Metric	Description	Time Interval	Instance Statistic
`SwapUsage`	The amount of swap space used.	1 min	sum
`TempStorageIOPS`	The number of IOPS for both read and writes on local storage attached to the Neptune DB instance. This metric represents a count and is measured once per second.		
`TempStorageThroughput`	The amount of data transferred to and from local storage associated with the Neptune DB instance. This metric represents bytes and is measured once per second.		

Metric	Description	Time Interval	Instance Statistic
`TotalBackupStorageBilled`	The total amount of backup storage for which you are billed for a given Neptune DB cluster, in bytes. Includes the backup storage measured by the `BackupRetentionPeriodStorageUsed` and `SnapshotStorageUsed` metrics.	1 day	sum
`TotalRequestsPerSec`	The total number of requests per second to the server from all sources.	1 min	average
`TotalClientErrorsPerSec`	The total number per second of requests that errored out because of client-side issues.	1 min	average
`TotalServerErrorsPerSec`	The total number per second of requests that errored out on the server because of internal failures.	1 min	average

Metric	Description	Time Interval	Instance Statistic
`UndoLogListSize`	The count of undo logs in the undo log list. Undo logs contain records of committed transactions that expire when all active transactions are more recent than the commit time. The expired records are periodically purged. Records for delete operations can take longer to purge than records for other types of transaction. Purging is done exclusively by the DB cluster's writer instance, so the rate of purging is dependent on the writer instance type. If the `UndoLogListSize` is high and growing in your DB cluster, upgrade the writer instance to increase the purge rate.	1 min	sum

Metric	Description	Time Interval	Instance Statistic
	Also, if you are upgrading to engine version `1.2.0.0` or higher from a version earlier than `1.2.0.0`, first make sure that the `UndoLogListSize` value is under a certain threshold. Otherwise, the patch will roll back and fail. The thresholds are based on instance type: the default limit is 40k for 4xlarge or larger instances, and 10k for instances smaller than 4xlarge. If you attempt to upgrade a cluster with the `UndoLogListSize` metric above the limit, the patch process will roll back, the upgrade will be canceled, and an event with the reason will be visible on the cluster event page. These limits can change for operational reasons without prior warning. Because		

Metric	Description	Time Interval	Instance Statistic
	engine versions `1.2.0.0` and higher use a different format for undo logs, the upgrade can only begin after your previous undo logs have been fully purged below the applicable threshold . See Upgrading to 1.2.0.0 or above for more information.		

Metric	Description	Time Interval	Instance Statistic
`VolumeBytesLeftTotal`	The remaining available space for the cluster volume, in bytes. As the cluster volume grows, this value decreases. If it reaches zero, the cluster reports an out-of-space error. If you want to detect whether your Neptune DB cluster is approaching it's size limit, this value is simpler and more reliable to monitor than `VolumeBytesUsed`. `VolumeBytesLeftTotal` takes into account storage used for internal housekeeping and other allocations that don't affect your storage billing. This metric is reported only by the writer instance. To avoid disruption during failovers, use the cluster-	1 min	average

Metric	Description	Time Interval	Instance Statistic
	level dimension (`DBCluster Identifier`).		
`VolumeBytesUsed`	The total amount of storage allocated to your Neptune DB cluster, in bytes. This is the amount of storage for which you are billed. It is the maximum amount of storage allocated to your DB cluster at any point in its existence, not the amount you are currently using (see Neptune storage billing).	5 min	sum
`VolumeReadIOPs`	The total number of billed read I/O operations from a cluster volume, reported a 5-minute intervals. Billed read operations are calculated at the cluster volume level, aggregated from all instances in the Neptune DB cluster, and then reported at 5-minute intervals.	5 min	sum

Metric	Description	Time Interval	Instance Statistic
`VolumeWriteIOPs`	The total number of write disk I/O operations to the cluster volume, reported at 5-minute intervals.	5 min	sum

CloudWatch Metrics That Are Now Deprecated in Neptune

Use of these Neptune metrics has now been deprecated. They are still supported, but may be eliminated in the future as new and better metrics become available.

Metric	Description
`GremlinHttp1xx`	Number of HTTP 1xx responses for the Gremlin endpoint per second. We recommend that you use the new `Http1xx` combined metric instead.
`GremlinHttp2xx`	Number of HTTP 2xx responses for the Gremlin endpoint per second.
`GremlinHttp4xx`	Number of HTTP 4xx errors for the Gremlin endpoint per second. We recommend that you use the new `Http4xx` combined metric instead.
`GremlinHttp5xx`	Number of HTTP 5xx errors for the Gremlin endpoint per second. We recommend that you use the new `Http5xx` combined metric instead.
`GremlinErrors`	Number of errors in Gremlin traversals.

Metric	Description
`GremlinRequests`	Number of requests to Gremlin engine.
`GremlinWebSocketSuccess`	Number of successful WebSocket connections to the Gremlin endpoint per second.
`GremlinWebSocketClientErrors`	Number of WebSocket client errors on the Gremlin endpoint per second.
`GremlinWebSocketServerErrors`	Number of WebSocket server errors on the Gremlin endpoint per second.
`GremlinWebSocketAvailableConnections`	Number of potential WebSocket connections currently available.
`Http100`	Number of HTTP 100 responses for the endpoint per second. We recommend that you use the new `Http1xx` combined metric instead.
`Http101`	Number of HTTP 101 responses for the endpoint per second. We recommend that you use the new `Http1xx` combined metric instead.
`Http1xx`	Number of HTTP 1xx responses for the endpoint per second.
`Http200`	Number of HTTP 200 responses for the endpoint per second.
`Http2xx`	Number of HTTP 2xx responses for the endpoint per second.

Metric	Description
`Http400`	Number of HTTP 400 errors for the endpoint per second. We recommend that you use the new `Http4xx` combined metric instead.
`Http403`	Number of HTTP 403 errors for the endpoint per second. We recommend that you use the new `Http4xx` combined metric instead.
`Http405`	Number of HTTP 405 errors for the endpoint per second. We recommend that you use the new `Http4xx` combined metric instead.
`Http413`	Number of HTTP 413 errors for the endpoint per second. We recommend that you use the new `Http4xx` combined metric instead.
`Http429`	Number of HTTP 429 errors for the endpoint per second. We recommend that you use the new `Http4xx` combined metric instead.
`Http4xx`	Number of HTTP 4xx errors for the endpoint per second.
`Http500`	Number of HTTP 500 errors for the endpoint per second. We recommend that you use the new `Http5xx` combined metric instead.

Metric	Description
`Http501`	Number of HTTP 501 errors for the endpoint per second. We recommend that you use the new `Http5xx` combined metric instead.
`Http5xx`	Number of HTTP 5xx errors for the endpoint per second.
`LoaderErrors`	Number of errors from Loader requests.
`LoaderRequests`	Number of Loader Requests.
`SparqlHttp1xx`	Number of HTTP 1xx responses for the SPARQL endpoint per second. We recommend that you use the new `Http1xx` combined metric instead.
`SparqlHttp2xx`	Number of HTTP 2xx responses for the SPARQL endpoint per second.
`SparqlHttp4xx`	Number of HTTP 4xx errors for the SPARQL endpoint per second. We recommend that you use the new `Http4xx` combined metric instead.
`SparqlHttp5xx`	Number of HTTP 5xx errors for the SPARQL endpoint per second. We recommend that you use the new `Http5xx` combined metric instead.
`SparqlErrors`	Number of errors in the SPARQL queries.
`SparqlRequests`	Number of requests to the SPARQL engine.

Metric	Description
`StatusErrors`	Number of errors from the status endpoint.
`StatusRequests`	Number of requests to the status endpoint.

Neptune CloudWatch Dimensions

The metrics for Amazon Neptune are qualified by the values for the account, graph name, or operation. You can use the Amazon CloudWatch console to retrieve Neptune data along with any of the dimensions in the following table.

Dimension	Description
`DBInstanceIdentifier`	Filters the data you request for a specific database instance within a cluster.
`DBClusterIdentifier`	Filters the data you request for a specific Neptune DB cluster.
`DBClusterIdentifier, EngineName`	Filters the data by the cluster. The engine name for all Neptune instances is `neptune`.
`DBClusterIdentifier, Role`	Filters the data you request for a specific Neptune DB cluster, aggregating the metric by instance role (WRITER/READER). For example, you can aggregate metrics for all READER instances that belong to a cluster.
`DBClusterIdentifier, SourceRegion`	Filters the data by the primary cluster in a global database primary region.
`DatabaseClass`	Filters the data you request for all instances in a database class. For example, you can aggregate metrics for all instances that belong to the database class db.r4.large

Dimension	Description
`EngineName`	The engine name for all Neptune instances is `neptune`.
`GlobalDbDBClusterIdentifier` , `SecondaryRegion`	Filters the data by the secondary cluster of a specified global database in a secondary region.

Using Audit Logs with Amazon Neptune Clusters

To audit Amazon Neptune DB cluster activity, enable the collection of audit logs by setting a DB cluster parameter. When audit logs are enabled, you can use it to log any combination of supported events. You can view or download the audit logs to review them.

Enabling Neptune Audit Logs

Use the `neptune_enable_audit_log` parameter to enable (`1`) or disable (`0`) audit logs.

Set this parameter in the parameter group that is used by your DB cluster. You can use the procedure shown in Editing a DB Cluster Parameter Group or DB Parameter Group to modify the parameter using the AWS Management Console, or use the modify-db-cluster-parameter-group AWS CLI command or the ModifyDBClusterParameterGroup API command to modify the parameter programatically.

You must reboot your DB instances after modifying this parameter in order to apply the change.

Viewing Neptune Audit Logs Using the Console

You can view and download the audit logs by using the AWS Management Console. On the **Instances** page, choose the DB instance to show its details, and then scroll to the **Logs** section.

To download a log file, select that file in the **Logs** section, and then choose **Download**.

Neptune Audit Log Details

Log files are in UTF-8 format. Logs are written in multiple files, the number of which varies based on the instance size. To see the latest events, you might have to review all the audit log files.

Log entries are not in sequential order. You can use the `timestamp` value for ordering them.

Log files are rotated when they reach 100 MB in aggregate. This limit is not configurable.

The audit log files include the following comma-delimited information in rows, in the following order:

Field	Description
Timestamp	The Unix timestamp for the logged event with microsecond precision.
ClientHost	The hostname or IP that the user connected from.
ServerHost	The hostname or IP of the instance that the event is logged for.
ConnectionType	The connection type. Can be `Websocket` , `HTTP_POST` , `HTTP_GET`, or `Bolt`.

Caller's IAM ARN	The ARN of the IAM user or IAM role used to sign the request. Empty if IAM authentication is disabled. Its format is:
	
	`arn:partition :service:region:account:resource`
	
	For example:
	
	`arn:aws:iam::123456789012:user/Anna`
	
	`arn:aws:sts::123456789012:assumed-role/AWSNeptuneNotebookRole/SageMaker`

Auth Context	Contains a serialized JSON object that has authentication information. The field `authenticationSucceeded` is `True` if the user was authenticated. Empty if IAM authentication is disabled.
HttpHeader	The HTTP header information. Can contain a query. Empty for WebSocket and Bolt connections.
Payload	The Gremlin, SPARQL, or openCypher query.

Publishing Neptune Logs to Amazon CloudWatch Logs

You can configure a Neptune DB cluster to publish audit log data and/or slow-query log data to a log group in Amazon CloudWatch Logs. With CloudWatch Logs, you can perform real-time analysis of the log data, and use CloudWatch to create alarms and view metrics. You can use CloudWatch Logs to store your log records in highly durable storage.

To publish audit logs to CloudWatch Logs, audit logs must be explicitly enabled (see Enable Audit Logs). Similarly, to publish slow-query logs to CloudWatch Logs, slow-query logs must be explicitly enabled (see Using Amazon Neptune slow-query logging).

> **Note**
>
> Be aware of the following:
>
> - Additional charges apply when you publish logs to CloudWatch. See the CloudWatch pricing page for details.
> - You can't publish logs to CloudWatch Logs for the China (Beijing) or China (Ningxia) region.
> - If exporting log data is disabled, Neptune doesn't delete existing log groups or log streams. If exporting log data is disabled, existing log data remains available in CloudWatch Logs, depending on log retention, and you still incur charges for stored audit log data. You can delete log streams and log groups using the CloudWatch Logs console, the AWS CLI, or the CloudWatch Logs API.

Using the Console to Publish Neptune Logs to CloudWatch Logs

To publish Neptune logs to CloudWatch Logs from the console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Databases**.
3. Choose the Neptune DB cluster that you want to publish the log data for.
4. For **Actions**, choose **Modify**.
5. In the **Log exports** section, choose the logs that you want to start publishing to CloudWatch Logs.

6. Choose **Continue**, and then choose **Modify DB Cluster** on the summary page.

Using the CLI to publish Neptune audit logs to CloudWatch Logs

You can create a new DB cluster that publishes audit logs to CloudWatch Logs using the AWS CLI `create-db-cluster` command with the following parameters:

```
aws neptune create-db-cluster \
    --region us-east-1  \
    --db-cluster-identifier my_db_cluster_id \
    --engine neptune \
    --enable-cloudwatch-logs-exports '["audit"]'
```

You can configure an existing DB cluster to publish audit logs to CloudWatch Logs using the AWS CLI `modify-db-cluster` command with the following parameters:

```
aws neptune modify-db-cluster \
    --region us-east-1  \
    --db-cluster-identifier my_db_cluster_id \
    --cloudwatch-logs-export-configuration '{"EnableLogTypes":["audit"]}'
```

Using the CLI to publish Neptune slow-query logs to CloudWatch Logs

You can also create a new DB cluster that publishes slow-query logs to CloudWatch Logs using the AWS CLI `create-db-cluster` command with the following parameters:

```
aws neptune create-db-cluster \
    --region us-east-1  \
    --db-cluster-identifier my_db_cluster_id \
    --engine neptune \
    --enable-cloudwatch-logs-exports '["slowquery"]'
```

Similarly, you can configure an existing DB cluster to publish slow-query logs to CloudWatch Logs using the AWS CLI `modify-db-cluster` command with the following parameters:

```
aws neptune modify-db-cluster --region us-east-1  \
    --db-cluster-identifier my_db_cluster_id \
    --cloudwatch-logs-export-configuration '{"EnableLogTypes":["slowquery"]}'
```

Monitoring Neptune Log Events in Amazon CloudWatch

After enabling Neptune logs, you can monitor log events in Amazon CloudWatch Logs. A new log group is automatically created for the Neptune DB cluster under the following prefix, in which *`cluster-name`* represents the DB cluster name, and *`log_type`* represents the log type:

```
/aws/neptune/cluster-name/log_type
```

For example, if you configure the export function to include the audit log for a DB cluster named `mydbcluster`, log data is stored in the `/aws/neptune/mydbcluster/audit` log group.

All of the events from all of the DB instances in a DB cluster are pushed to a log group using different log streams.

If a log group with the specified name exists, Neptune uses that log group to export log data for the Neptune DB cluster. You can use automated configuration, such as AWS CloudFormation, to create log groups with predefined log retention periods, metric filters, and customer access. Otherwise, a new log group is automatically created using the default log retention period, **Never Expire**, in CloudWatch Logs.

You can use the CloudWatch Logs console, the AWS CLI, or the CloudWatch Logs API to change the log retention period. For more information about changing log retention periods in CloudWatch Logs, see Change Log Data Retention in CloudWatch Logs.

You can use the CloudWatch Logs console, the AWS CLI, or the CloudWatch Logs API to search for information within the log events for a DB cluster. For more information about searching and filtering log data, see Searching and Filtering Log Data.

Using Amazon Neptune slow-query logging

Identifying, debugging and optimizing a slow-running query can be difficult. When Neptune's slow-query logging is enabled, attributes of all long-running queries are automatically logged to make this process easier.

> **Note**
>
> Slow-query logging was introduced in Neptune engine release 1.2.1.0.

You enable slow-query logging using the neptune_enable_slow_query_log DB cluster parameter. By default, this parameter is set to `disabled`. Setting it to `info` or `debug` enables slow-query logging. The `info` setting logs a few useful attributes of each slow-running query, whereas the `debug` setting logs all available attributes.

To set the threshold for what is considered a slow-running query, use the neptune_slow_query_log_threshold DB cluster parameter to specify the number of milliseconds after which a running query is considered slow and is logged when slow-query logging is enabled. The default value is 5000 milliseconds (5 seconds).

You can set these DB cluster parameters in the AWS Management Console, or using the modify-db-cluster-parameter-group AWS CLI command, or the ModifyDBClusterParameterGroup management function.

> **Note**
>
> The slow-query logging parameters are dynamic, meaning that changing their values does not require or cause a restart of your DB cluster.

To view slow-query logs in the AWS Management Console

You can view and download slow-query logs in the AWS Management Console, as follows:

On the **Instances** page, choose a DB instance and then scroll to the **Logs** section. You can then select a log file there and then choose **Download** to download it.

The files generated by Neptune slow-query logging

The log files generated by slow-query logging in Neptune have the following characteristics:

- The files are encoded as UTF-8.
- Queries and their attributes are logged in JSON form.
- Null and empty attributes are not logged, except for `queryTime` data.
- Logs span multiple files, the number of which varies with instance size.
- Log entries are not in sequential order. You can use their `timestamp` values to order them.
- To see the latest events, you may have to review all the slow-query log files.
- Log files are rotated when they reach 100 MiB in aggregate. This limit is not configurable.

Query attributes logged in `info` mode

The following attributes are logged for slow queries when the `neptune_enable_slow_query_log` DB cluster parameter has been set to `info`:

Group	Attribute	Description
requestResponseMetadata	`requestId`	Request id of query.
	`requestType`	Request type, like HTTP or WebSocket.
	`responseStatusCode`	Query response status code, like 200.
	`exceptionClass`	Exception class of the error returned after query execution.
queryStats	`query`	Query string.
	`queryFingerprint`	Fingerprint of the query.
	`queryLanguage`	Query language, like Gremlin, SPARQL or openCypher.
memoryStats	`allocatedPermits`	Permits allocated to the query.
	`approximateUsedMemoryBytes`	Approximate memory used by the query during execution.
queryTimeStats	`startTime`	Query start time (UTC).
	`overallRunTimeMs`	Query total run time, in milliseconds.
	`parsingTimeMs`	Query parsing time, in milliseconds.

Group	Attribute	Description
	`waitingTimeMs`	Query Gremlin/SPARQL/openCypher queue waiting time, in milliseconds
	`executionTimeMs`	Query execution time, in milliseconds.
	`serializationTimeMs`	Query serialization time, in milliseconds.
	`authTimeMs`	Query authentication and authorization time, in milliseconds.
statementCounters	`read`	Number of statements read.
	`written`	Number of statements written.
	`deleted`	Number of statements deleted.
transactionCounters	`committed`	Number of transactions commited.
	`rolledBack`	Number of transactions rolled back.
vertexCounters	`added`	Number of vertices added.
	`removed`	Number of vertices removed.
	`propertiesAdded`	Number of vertex properties added.
	`propertiesRemoved`	Number of vertex properties removed.

Group	Attribute	Description
edgeCounters	`added`	Number of edges added.
	`removed`	Number of edges removed.
	`propertiesAdded`	Number of edge properties added.
	`propertiesRemoved`	Number of edge properties removed.
resultCache	`hitCount`	Result cache hit count.
	`missCount`	Result cache miss count.
	`putCount`	Result cache put count.
concurrentExecutionStats	`acceptedQueryCountAtStart`	Parallel queries accepted with the current query execution at start.
	`runningQueryCountAtStart`	Parallel queries running with the current query execution at start.
	`acceptedQueryCountAtEnd`	Parallel queries accepted with the current query execution at end.
	`runningQueryCountAtEnd`	Parallel queries running with the current query execution at end.
queryBatch	`queryProcessingBatchSize`	Batch size during query processing.
	`querySerialisationBatchSize`	Batch size during query serialization.

Query attributes logged in debug mode

When the `neptune_enable_slow_query_log` DB cluster parameter has been set to `debug`, the following storage-counter attributes are logged in addition to the attributes that are logged as in `info` mode:

Attribute	Description
`statementsScannedInAllIndexes`	Statements scanned in all indexes.
`statementsScannedSPOGIndex`	Statements scanned in SPOG index.
`statementsScannedPOGSIndex`	Statements scanned in POGS index.
`statementsScannedGPSOIndex`	Statements scanned in GPSO index.
`statementsScannedOSGPIndex`	Statements scanned in OSGP index.
`statementsScannedInChunk`	Statements scanned together in chunk.
`postFilteredStatementScans`	Statements left after post filtering after scans.
`distinctStatementScans`	Distinct statements scanned.
`statementsReadInAllIndexes`	Statements read after scan post filtering in all indexes.
`statementsReadSPOGIndex`	Statements read after scan post filtering in SPOG index.
`statementsReadPOGSIndex`	Statements read after scan post filtering in POGS index.
`statementsReadGPSOIndex`	Statements read after scan post filtering in GPSO index.
`statementsReadOSGPIndex`	Statements read after scan post filtering in OSGP index.
`accessPathSearches`	Number of access path searches.

Attribute	Description
`fullyBoundedAccessPathSearches`	Number of fully bounded key access path searches.
`accessPathSearchedByPrefix`	Number of access path searched by prefix.
`searchesWhereRecordsWereFound`	Number of searches which had 1 or more records as output.
`searchesWhereRecordsWereNotFound`	Number of searches which had no records as output.
`totalRecordsFoundInSearches`	Total records found from all searches.
`statementsInsertedInAllIndexes`	Number of statements inserted in all indexes.
`statementsUpdatedInAllIndexes`	Number of statements updated in all indexes.
`statementsDeletedInAllIndexes`	Number of statements deleted in all indexes.
`predicateCount`	Number of predicates.
`dictionaryReadsFromValueToIdTable`	Number of dictionary reads from value to ID table.
`dictionaryReadsFromIdToValueTable`	Number of dictionary reads from ID of value table.
`dictionaryWritesToValueToIdTable`	Number of dictionary writes to value to ID table.
`dictionaryWritesToIdToValueTable`	Number of dictionary writes to ID to value table.
`rangeCountsInAllIndexes`	Number of range counts in all indexes.
`deadlockCount`	Number of deadlocks in the query.
`singleCardinalityInserts`	Number of single cardinanity inserts performed.

Attribute	Description
`singleCardinalityInsertDeletions`	Number of statements deleted during single cardinality insert.
`sharedLocksWaitTimeMillis`	Number of milliseconds spent waiting for shared locks.
`exclusiveLocksWaitTimeMillis`	Number of milliseconds spent waiting for exclusive locks.

Example of debug logging for a slow query

The following Gremlin query could take longer to run than the threshold set for slow queries:

```
gremlin=g.V().has('code','AUS').repeat(out().simplePath()).until(has('code','AGR')).path().by('
```

Then, if slow-query logging was enabled in debug mode, the following attributes would be logged for the query, in a form like this:

```
{
  "requestResponseMetadata": {
    "requestId": "5311e493-0e98-457e-9131-d250a2ce1e12",
    "requestType": "HTTP_GET",
    "responseStatusCode": 200
  },
  "queryStats": {
    "query":
 "gremlin=g.V().has('code','AUS').repeat(out().simplePath()).until(has('code','AGR')).path().by
    "queryFingerprint":
 "g.V().has(string0,string1).repeat(__.out().simplePath()).until(__.has(string0,string2)).path(
    "queryLanguage": "Gremlin"
  },
  "memoryStats": {
    "allocatedPermits": 20,
    "approximateUsedMemoryBytes": 14838
  },
  "queryTimeStats": {
    "startTime": "23/02/2023 11:42:52.657",
    "overallRunTimeMs": 2249,
```

```
    "executionTimeMs": 2229,
    "serializationTimeMs": 13
  },
  "statementCounters": {
    "read": 69979
  },
  "transactionCounters": {
    "committed": 1
  },
  "concurrentExecutionStats": {
    "acceptedQueryCountAtStart": 1
  },
  "queryBatchStats": {
    "queryProcessingBatchSize": 1000,
    "querySerialisationBatchSize": 1000
  },
  "storageCounters": {
    "statementsScannedInAllIndexes": 69979,
    "statementsScannedSPOGIndex": 44936,
    "statementsScannedPOGSIndex": 4,
    "statementsScannedGPSOIndex": 25039,
    "statementsReadInAllIndexes": 68566,
    "statementsReadSPOGIndex": 43544,
    "statementsReadPOGSIndex": 2,
    "statementsReadGPSOIndex": 25020,
    "accessPathSearches": 27,
    "fullyBoundedAccessPathSearches": 27,
    "dictionaryReadsFromValueToIdTable": 10,
    "dictionaryReadsFromIdToValueTable": 17,
    "rangeCountsInAllIndexes": 4,
    "sharedLocksWaitTimeMillis": 0,
    "exclusiveLocksWaitTimeMillis": 0
  }
}
```


Logging Amazon Neptune API Calls with AWS CloudTrail

Amazon Neptune is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Neptune. CloudTrail captures API calls for Neptune as events, including calls from the Neptune console and from code calls to the Neptune APIs.

CloudTrail only logs events for Neptune Management API calls, such as creating an instance or cluster. If you want to audit changes to your graph, you can use audit logs. For more information, see Using Audit Logs with Amazon Neptune Clusters.

> **Important**
>
> Amazon Neptune console, AWS CLI, and API calls are logged as calls made to the Amazon Relational Database Service (Amazon RDS) API.

If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Neptune. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Neptune, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

Neptune Information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Amazon Neptune, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for Neptune, create a trail. A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see:

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations
- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts

If an action is taken on behalf of your AWS account using the Neptune console, the Neptune command line interface, or the Neptune SDK APIs, AWS CloudTrail logs the action as calls made to the Amazon RDS API. For example, if you use the Neptune console to modify a DB instance or call the AWS CLI modify-db-instance command, the AWS CloudTrail log shows a call to the Amazon RDS API ModifyDBInstance action. For a list of the Neptune API actions that are logged by AWS CloudTrail, see the Neptune API Reference.

> **Note**
>
> AWS CloudTrail only logs events for Neptune Management API calls, such as creating an instance or cluster. If you want to audit changes to your graph, you can use audit logs. For more information, see Using Audit Logs with Amazon Neptune Clusters.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or IAM user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity Element.

Understanding Neptune log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files are not an ordered stack trace of the public API calls, so they do not appear in any specific order.

The following example shows a CloudTrail log for a user that created a snapshot of a DB instance and then deleted that instance using the Neptune console. The console is identified by the `userAgent` element. The requested API calls made by the console (`CreateDBSnapshot` and `DeleteDBInstance`) are found in the `eventName` element for each record. Information about the user (`Alice`) can be found in the `userIdentity` element.

```
{
```

```
  Records:[
  {
    "awsRegion":"us-west-2",
    "eventName":"CreateDBSnapshot",
    "eventSource":"domainSource",
    "eventTime":"2014-01-14T16:23:49Z",
    "eventVersion":"1.0",
    "sourceIPAddress":"192.0.2.01",
    "userAgent":"AWS Console, aws-sdk-java\/unknown-version Linux\/2.6.18-
kaos_fleet-1108-prod.2 Java_HotSpot(TM)_64-Bit_Server_VM\/24.45-b08",
    "userIdentity":
    {
      "accessKeyId":"0123456789012",
      "accountId":"123456789012",
      "arn":"arn:aws-eusc:iam::123456789012:user/Alice",
      "principalId":"AIDAI2JXM4FBZZEXAMPLE",
      "sessionContext":
      {
        "attributes":
        {
          "creationDate":"2014-01-14T15:55:59Z",
          "mfaAuthenticated":false
        }
      },
      "type":"IAMUser",
      "userName":"Alice"
    }
  },
  {
    "awsRegion":"us-west-2",
    "eventName":"DeleteDBInstance",
    "eventSource":"domainSource",
    "eventTime":"2014-01-14T16:28:27Z",
    "eventVersion":"1.0",
    "sourceIPAddress":"192.0.2.01",
    "userAgent":"AWS Console, aws-sdk-java\/unknown-version Linux\/2.6.18-
kaos_fleet-1108-prod.2 Java_HotSpot(TM)_64-Bit_Server_VM\/24.45-b08",
    "userIdentity":
    {
      "accessKeyId":"0123456789012",
      "accountId":"123456789012",
      "arn":"arn:aws-eusc:iam::123456789012:user/Alice",
      "principalId":"AIDAI2JXM4FBZZEXAMPLE",
      "sessionContext":
```

```
      {
        "attributes":
        {
          "creationDate":"2014-01-14T15:55:59Z",
          "mfaAuthenticated":false
        }
      },
      "type":"IAMUser",
      "userName":"Alice"
    }
  }
  ]
}
```

Amazon Neptune events

Topics

- Amazon Neptune event categories and event messages
- Viewing Neptune events
- Subscribing to Neptune event notification
- Managing Neptune event notification subscriptions

Amazon Neptune uses Amazon Simple Notification Service (Amazon SNS) to provide notifications when a Neptune event occurs. These notifications can be in any form that is supported by Amazon SNS for an AWS Region, such as an email, a text message, or a call to an HTTP endpoint.

Neptune groups these events into categories that you can subscribe to so that you can be notified when an event in that category occurs. You can subscribe to an event category for a DB instance, DB cluster, DB snapshot, DB cluster snapshot, or for a DB parameter group. For example, if you subscribe to the Backup category for a given DB instance, you are notified whenever a backup-related event occurs that affects the DB instance. You also receive notification when an event notification subscription changes.

Events occur at both the DB cluster and the DB instance level, so you can receive events if you subscribe to a DB cluster or a DB instance.

Event notifications are sent to the addresses you provide when you create the subscription. You might want to create several different subscriptions, such as a subscription that receives all event

notifications and another subscription that includes only critical events for your production DB instances. You can easily turn off notification without deleting a subscription. To do so, set the **Enabled** radio button to **No** in the Neptune console.

> **Important**
>
> Amazon Neptune doesn't guarantee the order of events sent in an event stream. The event order is subject to change.

Neptune uses the Amazon Resource Name (ARN) of an Amazon SNS topic to identify each subscription. The Neptune console creates the ARN for you when you create the subscription.

Billing for Neptune event notification is through Amazon SNS. Amazon SNS fees apply when using event notification. For more information, see Amazon Simple Notification Service Pricing.

Amazon Neptune event categories and event messages

Neptune generates a significant number of events in categories that you can subscribe to using the Neptune console. Each category applies to a source type, which can be a DB instance, DB snapshot, or DB parameter group.

> **Note**
>
> Neptune uses existing Amazon RDS event definitions and IDs.

Neptune events originating from DB instances

The following table shows a list of events by event category when a DB instance is the source type.

Category	Amazon RDS event ID	Description
availability	RDS-EVENT-0006	The DB instance restarted.
	RDS-EVENT-0004	DB instance shutdown.

Category	Amazon RDS event ID	Description
	RDS-EVENT-0022	An error occurred while restarting the Neptune engine.
backup	RDS-EVENT-0001	Backing up DB instance.
	RDS-EVENT-0002	Finished DB Instance backup.
configuration change	RDS-EVENT-0009	The DB instance has been added to a security group.
	RDS-EVENT-0024	The DB instance is being converted to a Multi-AZ DB instance.
	RDS-EVENT-0030	The DB instance is being converted to a Single-AZ DB instance.
	RDS-EVENT-0012	Applying modificat ion to database instance class.
	RDS-EVENT-0018	The current storage settings for this DB instance are being changed.
	RDS-EVENT-0011	A parameter group for this DB instance has changed.

Category	Amazon RDS event ID	Description
	RDS-EVENT-0092	A parameter group for this DB instance has finished updating.
	RDS-EVENT-0028	Automatic backups for this DB instance have been disabled.
	RDS-EVENT-0032	Automatic backups for this DB instance have been enabled.
	RDS-EVENT-0025	The DB instance has been converted to a Multi-AZ DB instance.
	RDS-EVENT-0029	The DB instance has been converted to a Single-AZ DB instance.
	RDS-EVENT-0014	The DB instance class for this DB instance has changed.
	RDS-EVENT-0017	The storage settings for this DB instance have changed.
	RDS-EVENT-0010	The DB instance has been removed from a security group.
creation	RDS-EVENT-0005	DB instance created.

Category	Amazon RDS event ID	Description
deletion	RDS-EVENT-0003	The DB instance has been deleted.
failover	RDS-EVENT-0034	Neptune is not attempting a requested failover because a failover recently occurred on the DB instance.
	RDS-EVENT-0013	A Multi-AZ failover that resulted in the promotion of a standby instance has started.
	RDS-EVENT-0015	A Multi-AZ failover that resulted in the promotion of a standby instance is complete. It may take several minutes for the DNS to transfer to the new primary DB instance.
	RDS-EVENT-0065	The instance has recovered from a partial failover.
	RDS-EVENT-0049	A Multi-AZ failover has completed.

Category	Amazon RDS event ID	Description
	RDS-EVENT-0050	A Multi-AZ activatio n has started after a successful instance recovery.
	RDS-EVENT-0051	A Multi-AZ activatio n is complete. Your database should be accessible now.
	RDS-EVENT-0031	The DB instance has failed due to an incompatible configuration or an underlying storage issue. Begin a point-in-time-restore for the DB instance.
	RDS-EVENT-0036	The DB instance is in an incompatible network. Some of the specified subnet IDs are invalid or do not exist.

Category	Amazon RDS event ID	Description
	RDS-EVENT-0035	The DB instance has invalid parameters. For example, if the DB instance could not start because a memory-related parameter is set too high for this instance class, the customer action would be to modify the memory parameter and reboot the DB instance.
	RDS-EVENT-0082	Neptune was unable to copy backup data from an Amazon S3 bucket. It is likely that the permissio ns for Neptune to access the Amazon S3 bucket are configured incorrectly.
low storage	RDS-EVENT-0089	The DB instance has consumed more than 90% of its allocated storage. You can monitor the storage space for a DB instance using the **Free Storage Space** metric.

Category	Amazon RDS event ID	Description
	RDS-EVENT-0007	The allocated storage for the DB instance has been exhausted. To resolve this issue, you should allocate additional storage for the DB instance.
maintenance	RDS-EVENT-0026	Offline maintenance of the DB instance is taking place. The DB instance is currently unavailable.
	RDS-EVENT-0027	Offline maintenance of the DB instance is complete. The DB instance is now available.
	RDS-EVENT-0047	Patching of the DB instance has completed.
notification	RDS-EVENT-0044	Operator-issued notification. For more information, see the event message.
	RDS-EVENT-0048	Patching of the DB instance has been delayed.
	RDS-EVENT-0087	The DB instance has been stopped.

Category	Amazon RDS event ID	Description
	RDS-EVENT-0088	The DB instance has been started.
	RDS-EVENT-0154	The DB instance is being started due to it exceeding the maximum allowed time being stopped.
	RDS-EVENT-0158	DB instance is in a state that can't be upgraded.
	RDS-EVENT-0173	DB instance has been patched.
read replica	RDS-EVENT-0045	An error has occurred in the read replicati on process. For more information, see the event message.

Category	Amazon RDS event ID	Description
	RDS-EVENT-0046	The read replica has resumed replication. This message appears when you first create a read replica, or as a monitoring message confirmin g that replication is functioning properly. If this message follows an RDS-EVENT-0045 notificat ion, then replicati on has resumed following an error or after replication was stopped.
	RDS-EVENT-0057	Replication on the read replica was terminated.
	RDS-EVENT-0062	Replication on the read replica was manually stopped.
	RDS-EVENT-0063	Replication on the read replica was reset.

Category	Amazon RDS event ID	Description
recovery	RDS-EVENT-0020	Recovery of the DB instance has started. Recovery time will vary with the amount of data to be recovered.
	RDS-EVENT-0021	Recovery of the DB instance is complete.
	RDS-EVENT-0023	A manual backup has been requested but Neptune is currently in the process of creating a DB snapshot. Submit the request again after Neptune has completed the DB snapshot.
	RDS-EVENT-0052	Recovery of the Multi-AZ instance has started. Recovery time will vary with the amount of data to be recovered.
	RDS-EVENT-0053	Recovery of the Multi-AZ instance is complete.
restoration	RDS-EVENT-0008	The DB instance has been restored from a DB snapshot.

Category	Amazon RDS event ID	Description
	RDS-EVENT-0019	The DB instance has been restored from a point-in-time backup.

Neptune events originating from a DB cluster

The following table shows a list of events by event category when a DB cluster is the source type.

Category	RDS event ID	Description
failover	RDS-EVENT-0069	A failover for the DB cluster has failed.
	RDS-EVENT-0070	A failover for the DB cluster has restarted.
	RDS-EVENT-0071	A failover for the DB cluster has finished.
	RDS-EVENT-0072	A failover for the DB cluster has begun within the same Availability Zone.
	RDS-EVENT-0073	A failover for the DB cluster has begun across Availability Zones.
	RDS-EVENT-0083	Neptune was unable to copy backup data from an Amazon S3 bucket. It is likely that the permissio ns for Neptune to access the Amazon S3

Category	RDS event ID	Description
		bucket are configured incorrectly.
maintenance	RDS-EVENT-0156	The DB cluster has a DB engine minor version upgrade available.
notification	RDS-EVENT-0076	Migration to an Neptune DB cluster failed.
	RDS-EVENT-0077	An attempt to convert a table from the source database to database form failed during the migration to an Neptune DB cluster.
	RDS-EVENT-0150	The DB cluster stopped.
	RDS-EVENT-0151	The DB cluster started.
	RDS-EVENT-0152	The DB cluster stop failed.
	RDS-EVENT-0153	The DB cluster is being started due to it exceeding the maximum allowed time being stopped.

Neptune events originating from DB cluster snapshot

The following table shows the event category and a list of events when a Neptune DB cluster snapshot is the source type.

Category	RDS event ID	Description
backup	RDS-EVENT-0074	Creation of a manual DB cluster snapshot has started.
backup	RDS-EVENT-0075	A manual DB cluster snapshot has been created.
notification	RDS-EVENT-0162	DB cluster snapshot export task failed.
notification	RDS-EVENT-0163	DB cluster snapshot export task canceled.
notification	RDS-EVENT-0164	DB cluster snapshot export task completed.
backup	RDS-EVENT-0168	Creating automated cluster snapshot.
backup	RDS-EVENT-0169	Automated cluster snapshot created.
creation	RDS-EVENT-0170	DB cluster created.
deletion	RDS-EVENT-0171	DB cluster deleted.
notification	RDS-EVENT-0172	Renamed DB cluster from [old DB cluster name] to [new DB cluster name].

Neptune events originating from DB cluster parameter group

The following table shows the event category and a list of events when a DB cluster parameter group is the source type.

Category	RDS event ID	Description
configuration change	RDS-EVENT-0037	The parameter group was modified.

Neptune events originating from a security group

The following table shows the event category and a list of events when a security group is the source type.

Category	RDS event ID	Description
configuration change	RDS-EVENT-0038	The security group has been modified.
failure	RDS-EVENT-0039	The security group owned by [user] does not exist; authoriza tion for the security group has been revoked.

Viewing Neptune events

You can view the following information for each Neptune event:

- Resource name
- Resource type
- Event category
- Time of the event

- A summary message describing the event

You can access events in the following parts of the AWS Management Console:

- The **Events** page in the Neptune console navigation pane, which shows events from the past 24 hours.
- The **Recent events** table in the **Logs & events** tab on a DB cluster or DB instance detail page, which can show events for up to the past 14 days.

You can also retrieve events by using the describe-events AWS CLI command or the DescribeEvents API operation. With the AWS CLI or API, you can retrieve events for up to the past 14 days.

Viewing events (console)

To view Neptune events in the console

1. Sign in to the AWS Management Console and open the Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Events**.

   The page displays events from the past 24 hours. You can filter events by source type and event category.

Viewing events (AWS CLI)

To view all Neptune events generated in the past hour, run `describe-events` with no parameters:

```
aws neptune describe-events
```

To view events for the past 7 days (10,080 minutes), use the `--duration` parameter:

```
aws neptune describe-events --duration 10080
```

To view events for a specific DB cluster within a time range, use the `--source-identifier`, `--source-type`, `--start-time`, and `--end-time` parameters:

```
aws neptune describe-events \
```

```
    --source-identifier my-cluster \
    --source-type db-cluster \
    --start-time 2026-04-01T00:00Z \
    --end-time 2026-04-01T23:59Z
```

For more information, see describe-events in the AWS CLI Command Reference. To retrieve events programmatically, see DescribeEvents in the Neptune API Reference.

Subscribing to Neptune event notification

You can use the Neptune console to subscribe to event notifications, as follows:

To subscribe to Neptune event notification

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Event subscriptions**.
3. In the **Event subscriptions** pane, choose **Create event subscription**.
4. In the **Create event subscription** dialog box, do the following:
   a. For **Name**, enter a name for the event notification subscription.
   b. For **Send notifications to**, choose an existing Amazon SNS ARN for an Amazon SNS topic, or choose **create topic** to enter the name of a topic and a list of recipients.
   c. For **Source type**, choose a source type.
   d. Choose **Yes** to enable the subscription. If you want to create the subscription but to not have notifications sent yet, choose **No**.
   e. Depending on the source type you selected, choose the event categories and the sources that you want to receive event notifications from.
   f. Choose **Create**.

Managing Neptune event notification subscriptions

If you choose **Event subscriptions** in the navigation pane of the Neptune console, you can view subscription categories and a list of your current subscriptions.

You can also modify or delete a specific subscription.

Modifying Neptune event notification subscriptions

To modify your current Neptune event notification subscriptions

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Event subscriptions**. The **Event subscriptions** pane shows all your event notification subscriptions.
3. In the **Event subscriptions** pane, choose the subscription that you want to modify and choose **Edit**.
4. Make your changes to the subscription in either the **Target** or **Source** section. You can add or remove source identifiers by selecting or deselecting them in the **Source** section.
5. Choose **Edit**. The Neptune console indicates that the subscription is being modified.

Deleting a Neptune event notification subscription

You can delete a subscription when you no longer need it. All subscribers to the topic will no longer receive event notifications specified by the subscription.

To delete an Neptune event notification subscription

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Event subscriptions**.
3. In the **Event subscriptions** pane, choose the subscription that you want to delete.
4. Choose **Delete**.
5. The Neptune console indicates that the subscription is being deleted.

Tagging Amazon Neptune resources

You can use Neptune tags to add metadata to your Neptune resources. In addition, you can use tags with AWS Identity and Access Management (IAM) policies to manage access to Neptune resources and control what actions can be applied to those resources. Finally, you can use tags to track costs by grouping expenses for similarly tagged resources.

All Neptune administrative resources can be tagged, including the following:

- DB instances
- DB clusters
- Read Replicas
- DB snapshots
- DB cluster snapshots
- Event subscriptions
- DB parameter groups
- DB cluster parameter groups
- DB subnet groups

Overview of Neptune resource tags

An Amazon Neptune tag is a name-value pair that you define and associate with a Neptune resource. The name is referred to as the *key*. Supplying a value for the key is optional. You can use tags to assign arbitrary information to a Neptune resource. You can use a tag key, for example, to define a category, and the tag value might be an item in that category. For example, you might define a tag key of "project" and a tag value of "Salix," indicating that the Neptune resource is assigned to the Salix project. You can also use tags to designate Neptune resources as being used for test or production by using a key such as `environment=test` or `environment=production`. We recommend that you use a consistent set of tag keys to make it easier to track metadata that is associated with Neptune resources.

Use tags to organize your AWS bill to reflect your own cost structure. To do this, sign up to get your AWS account bill with tag key values included. Then, to see the cost of combined resources, organize your billing information according to resources with the same tag key values. For example, you can tag several resources with a specific application name, and then organize your billing information to see the total cost of that application across several services. For more information, see Using Cost Allocation Tags in the *AWS Billing User Guide*.

Each Neptune resource has a tag set, which contains all the tags that are assigned to that Neptune resource. A tag set can contain as many as 10 tags, or it can be empty. If you add a tag to a Neptune resource that has the same key as an existing tag on resource, the new value overwrites the old value.

AWS does not apply any semantic meaning to your tags; tags are interpreted strictly as character strings. Neptune can set tags on a DB instance or other Neptune resources, depending on the

settings that you use when you create the resource. For example, Neptune might add a tag indicating that a DB instance is for production or for testing.

- The tag key is the required name of the tag. The string value can be from 1 to 128 Unicode characters in length and cannot be prefixed with "`aws:`" or "`rds:`". The string can contain only the set of Unicode letters, digits, white space, '_', '.', '/', '=', '+', '-' (Java regex: "`^([\\p{L}\\p{Z}\\p{N}_.:/=+\\-]*)$`").
- The tag value is an optional string value of the tag. The string value can be from 1 to 256 Unicode characters in length and cannot be prefixed with "`aws:`". The string can contain only the set of Unicode letters, digits, white space, '_', '.', '/', '=', '+', '-' (Java regex: "`^([\\p{L}\\p{Z}\\p{N}_.:/=+\\-]*)$`").

  Values do not have to be unique in a tag set and can be null. For example, you can have a key-value pair in a tag set of `project/Trinity` and `cost-center/Trinity`.

> **Note**
>
> You can add a tag to a snapshot. However, your bill won't reflect this grouping.

You can use the AWS Management Console, the AWS CLI, or the Neptune API to add, list, and delete tags on Neptune resources. When using the AWS CLI or the Neptune API, you must provide the Amazon Resource Name (ARN) for the Neptune resource that you want to work with. For more information about constructing an ARN, see Constructing an ARN for Neptune.

Tags are cached for authorization purposes. Because of this, additions and updates to tags on Neptune resources can take several minutes before they are available.

Copying tags in Neptune

When you create or restore a DB instance, you can specify that the tags from the DB instance are copied to snapshots of the DB instance. Copying tags ensures that the metadata for the DB snapshots matches that of the source DB instance, and that any access policies for the DB snapshot also match those of the source DB instance. Tags are not copied by default.

You can specify that tags are copied to DB snapshots for the following actions:

- Creating a DB instance.

- Restoring a DB instance.
- Creating a Read Replica.
- Copying a DB snapshot.

> **Note**
>
> If you include a value for the `--tag-key` parameter of the create-db-cluster-snapshot AWS CLI command (or supply at least one tag to the CreateDBClusterSnapshot API action), Neptune doesn't copy tags from the source DB instance to the new DB snapshot. This is true even if the source DB instance has the `--copy-tags-to-snapshot` (`CopyTagsToSnapshot`) option enabled.
> This means that you can create a copy of a DB instance from a DB snapshot and avoid adding tags that don't apply to the new DB instance. After you create your DB snapshot using the AWS CLI `create-db-cluster-snapshot` command (or the `CreateDBClusterSnapshot` Neptune API action), you can then add tags as described later in this topic.

Tagging in Neptune using the AWS Management Console

The process to tag an Amazon Neptune resource is similar for all resources. The following procedure shows how to tag a Neptune DB instance.

To add a tag to a DB instance

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Instances**.

   > **Note**
   >
   > To filter the list of DB instances in the **Instances** pane, type a text string in the **Filter instances** box. Only DB instances that contain the string appear.

3. Choose the DB instance that you want to tag.
4. Choose **Instance actions**, and then choose **See details**.

5. In the details section, find the **Tags** section.
6. Choose **Add**. The **Add tags** window appears.
7. Type a value for **Tag key** and **Value**.
8. To add another tag, you can choose **Add another Tag** and type a value for its **Tag key** and **Value**.

   Repeat this step as many times as necessary.
9. Choose **Add**.

To delete a tag from a DB instance

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Instances**.

   > **Note**
   >
   > To filter the list of DB instances in the **Instances** pane, type a text string in the **Filter instances** box. Only DB instances that contain the string appear.

3. Choose the DB instance that you want to tag.
4. Choose **Instance actions**, and then choose **See details**.
5. In the details section, find the **Tags** section.
6. Choose the tag that you want to delete.
7. Choose **Remove**, and then choose **Remove** in the **Remove tags** window.

Tagging in Neptune using the AWS CLI

You can add, list, or remove tags for a DB instance in Neptune using the AWS CLI.

- To add one or more tags to a Neptune resource, use the AWS CLI command `add-tags-to-resource`.
- To list the tags on a Neptune resource, use the AWS CLI command `list-tags-for-resource`.
- To remove one or more tags from a Neptune resource, use the AWS CLI command `remove-tags-from-resource`.

To learn more about how to construct the required Amazon Resource Name (ARN), see Constructing an ARN for Neptune.

Tagging in Neptune using the API

You can add, list, or remove tags for a DB instance using the Neptune API.

- To add a tag to a Neptune resource, use the `AddTagsToResource` operation.
- To list tags that are assigned to a Neptune resource, use the `ListTagsForResource`.
- To remove tags from a Neptune resource, use the `RemoveTagsFromResource` operation.

To learn more about how to construct the required ARN, see Constructing an ARN for Neptune.

When working with XML using the Neptune API, tags use the following schema:

```
<Tagging>
    <TagSet>
        <Tag>
            <Key>Project</Key>
            <Value>Trinity</Value>
        </Tag>
        <Tag>
            <Key>User</Key>
            <Value>Jones</Value>
        </Tag>
    </TagSet>
</Tagging>
```

The following table provides a list of the allowed XML tags and their characteristics. Values for `Key` and `Value` are case-dependent. For example, `project=Trinity` and `PROJECT=Trinity` are two distinct tags.

Tagging Element	Description
`TagSet`	A tag set is a container for all tags that are assigned to a Neptune resource. There can be only one tag set per resource. You work with a `TagSet` only through the Neptune API.

Tagging Element	Description
Tag	A tag is a user-defined key-value pair. There can be from 1 to 50 tags in a tag set.
Key	A key is the required name of the tag. The string value can be from 1 to 128 Unicode characters in length and cannot be prefixed with "rds:" or "aws:". The string can contain only the set of Unicode letters, digits, white space, '_', '.', '/', '=', '+', '-' (Java regex: "^([\\p{L}\\p{Z}\\p{N}_.:/=+\\-]*)$"). Keys must be unique to a tag set. For example, you can't have a key-pair in a tag set with the key the same but with different values, such as project/Trinity and project/Xanadu .
Value	A value is the optional value of the tag. The string value can be from 1 to 256 Unicode characters in length and cannot be prefixed with "rds:" or "aws:". The string can contain only the set of Unicode letters, digits, white space, '_', '.', '/', '=', '+', '-' (Java regex: "^([\\p{L}\\p{Z}\\p{N}_.:/=+\\-]*)$"). Values don't have to be unique in a tag set and can be null. For example, you can have a key-value pair in a tag set of project/Trinity and cost-center/Trinity .

Working with administrative ARNs in Amazon Neptune

Resources that are created in Amazon Web Services are each uniquely identified with an Amazon Resource Name (ARN). For certain Amazon Neptune operations, you must uniquely identify a Neptune resource by specifying its ARN.

> **Important**
>
> Amazon Neptune shares the format of Amazon RDS ARNs for administrative actions that use the Management API reference. Neptune administrative ARNs contain rds and not neptune-db. For data-plane ARNs that identify Neptune data resources, see Specifying data resources.

Topics

- Constructing an ARN for Neptune
- Getting an existing ARN in Amazon Neptune

Constructing an ARN for Neptune

You can construct an ARN for an Amazon Neptune resource using the following syntax. Note that Neptune shares the format of Amazon RDS ARNs.

```
arn:aws-eusc:rds:<region>:<account number>:<resourcetype>:<name>
```

The following table shows the format that you should use when constructing an ARN for a particular Neptune administrative resource type.

Resource Type	ARN Format
DB instance	arn:aws-eusc:rds:`<region>`:`<account>` `:db:`<name>`  For example:  `arn:aws-eusc:rds: us-east-2 :123456789012 :db:my-instance-1`
DB cluster	arn:aws-eusc:rds:`<region>`:`<account>` `:cluster:` `<name>` For example: `arn:aws-eusc:rds: us-east-2 :123456789012 :cluster: my-cluster-1`
Event subscription	arn:aws-eusc:rds:`<region>`:`<account>` `:es:`<name>`  For example:  `arn:aws-eusc:rds: us-east-2 :123456789012 :es:my-subscription`
DB parameter group	arn:aws-eusc:rds:`<region>`:`<account>` `:pg:`<name>`

Resource Type	ARN Format
	For example: `arn:aws-eusc:rds: us-east-2 :123456789012 :pg:my-param-enable-logs`
DB cluster parameter group	arn:aws-eusc:rds:`<region>`:`<account>` `:cluster-pg:` `<name>` For example: `arn:aws-eusc:rds: us-east-2 :123456789012 :cluster-pg: my-cluster-param-timezone`
DB cluster snapshot	arn:aws-eusc:rds`:`<region>`:`<account>` `:cluster-snapshot:` `<name>` For example: `arn:aws-eusc:rds: us-east-2 :123456789012 :cluster-snapshot: my-snap-20160809`
DB subnet group	arn:aws-eusc:rds`:`<region>`:`<account>` `:subgrp:`<name> For example: `arn:aws-eusc:rds: us-east-2 :123456789012 :subgrp:my-subnet-10`

Getting an existing ARN in Amazon Neptune

You can get the ARN of a Neptune resource by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or Neptune API.

Getting an existing ARN using the AWS Management Console

To get an ARN using the console, navigate to the resource that you want an ARN for, and view the details for that resource. For example, to get the ARN for a DB instance, choose **Instances** in the

navigation panel, and choose the instance that you want from the list. The ARN is in the **Instance Details** section.

Getting an existing ARN using the AWS CLI

To use the AWS CLI to get an ARN for a particular Neptune resource, use the `describe` command for that resource. The following table shows each AWS CLI command and the ARN property that is used with the command to get an ARN.

AWS CLI Command	ARN Property
describe-event-subscriptions	EventSubscriptionArn
describe-certificates	CertificateArn
describe-db-parameter-groups	DBParameterGroupArn
describe-db-cluster-parameter-groups	DBClusterParameterGroupArn
describe-db-instances	DBInstanceArn
describe-events	SourceArn
describe-db-subnet-groups	DBSubnetGroupArn
describe-db-clusters	DBClusterArn
describe-db-cluster-snapshots	DBClusterSnapshotArn

For example, the following AWS CLI command gets the ARN for a DB instance.

Example

For Linux, OS X, or Unix:

```
aws neptune describe-db-instances \
--db-instance-identifier DBInstanceIdentifier \
--region us-west-2
```

For Windows:

```
aws neptune describe-db-instances ^
--db-instance-identifier DBInstanceIdentifier ^
--region us-west-2
```

Getting an existing ARN using the API

To get an ARN for a particular Neptune resource, call the following API actions and use the ARN properties shown.

Neptune API Action	ARN Property
DescribeEventSubscriptions	EventSubscriptionArn
DescribeCertificates	CertificateArn
DescribeDBParameterGroups	DBParameterGroupArn
DescribeDBClusterParameterGroups	DBClusterParameterGroupArn
DescribeDBInstances	DBInstanceArn
DescribeEvents	SourceArn
DescribeDBSubnetGroups	DBSubnetGroupArn
DescribeDBClusters	DBClusterArn
DescribeDBClusterSnapshots	DBClusterSnapshotArn

Backing up and restoring an Amazon Neptune DB cluster

This section shows how you can back up and restore Amazon Neptune DB clusters.

Topics

- Neptune Backups
- Fault tolerance for a Neptune DB cluster
- CloudWatch metrics that are useful for managing Neptune backup storage
- Restoring data from a Neptune backup
- Backup window in Neptune
- Creating a DB Cluster Snapshot in Neptune
- Restoring from a DB Cluster Snapshot
- Copying a DB Cluster Snapshot
- Sharing a DB Cluster Snapshot
- Deleting a Neptune Snapshot

Neptune Backups

Neptune backs up your cluster volume automatically and retains restore data for the length of the *backup retention period*. Neptune backups are continuous and incremental so you can quickly restore to any point within the backup retention period. No performance impact or interruption of database service occurs as backup data is being written. You can specify a backup retention period, from 1 to 35 days, when you create or modify a DB cluster.

To control your backup storage usage, you can reduce the backup retention interval, remove old manual snapshots when they are no longer needed, or both. To help manage your costs, you can monitor the amount of storage consumed by continuous backups and manual snapshots that persist beyond the retention period. You can reduce the backup retention interval and remove manual snapshots when they are no longer needed.

If you want to retain a backup beyond the backup retention period, you can also take a snapshot of the data in your cluster volume. Storing snapshots incurs the standard storage charges for Neptune. For more information about Neptune storage pricing, see Amazon Neptune Pricing.

Neptune retains incremental restore data for the entire backup retention period. So you only need to create a snapshot for data that you want to retain beyond the backup retention period. You can create a new DB cluster from the snapshot.

> **Important**
>
> If you delete a DB cluster, all its automated backups are deleted at the same time and cannot be recovered. This means that unless you choose to create a final DB snapshot manually, you can't restore the DB instance to its final state at a later time. Manual snapshots are not deleted when the cluster is deleted.

> **Note**
>
> - For Amazon Neptune DB clusters, the default backup retention period is one day regardless of how the DB cluster is created.
> - You cannot disable automated backups on Neptune. The backup retention period for Neptune is managed by the DB cluster.

Fault tolerance for a Neptune DB cluster

A Neptune DB cluster is fault tolerant by design. The cluster volume spans multiple Availability Zones in a single AWS Region, and each Availability Zone contains a copy of the cluster volume data. This functionality means that your DB cluster can tolerate a failure of an Availability Zone without any loss of data and only a brief interruption of service.

If the primary instance in a DB cluster fails, Neptune automatically fails over to a new primary instance in one of two ways:

- By promoting an existing Neptune replica to the new primary instance
- By creating a new primary instance

If the DB cluster has one or more Neptune replicas, then a Neptune replica is promoted to the primary instance during a failure event. A failure event results in a brief interruption, during which read and write operations fail with an exception. However, service is typically restored in less

than 120 seconds, and often less than 60 seconds. To increase the availability of your DB cluster, we recommend that you create at least one or more Neptune replicas in two or more different Availability Zones.

You can customize the order in which your Neptune replicas are promoted to the primary instance after a failure by assigning each replica a priority. Priorities range from 0 for the highest priority to 15 for the lowest priority. If the primary instance fails, Neptune promotes the Neptune replica with the highest priority to the new primary instance. You can modify the priority of a Neptune replica at any time. Modifying the priority doesn't trigger a failover.

You can use the AWS CLI to set the failover priority of a DB instance, as follows:

```
aws neptune modify-db-instance --db-instance-identifier (the instance ID) --promotion-
tier (the failover priority value)
```

More than one Neptune replica can share the same priority, resulting in promotion tiers. If two or more Neptune replicas share the same priority, then Neptune promotes the replica that is largest in size. If two or more Neptune replicas share the same priority and size, then Neptune promotes an arbitrary replica in the same promotion tier.

If the DB cluster doesn't contain any Neptune replicas, then the primary instance is recreated during a failure event. A failure event results in an interruption during which read and write operations fail with an exception. Service is restored when the new primary instance is created, which typically takes less than 10 minutes. Promoting a Neptune replica to the primary instance is much faster than creating a new primary instance.

CloudWatch metrics that are useful for managing Neptune backup storage

You can use the Amazon CloudWatch metrics `TotalBackupStorageBilled`, `SnapshotStorageUsed`, and `BackupRetentionPeriodStorageUsed` to review and monitor the amount of storage used by your Neptune backups, as follows:

- `BackupRetentionPeriodStorageUsed` represents the amount of backup storage used, in bytes, for storing continuous backups at the current time. This value depends on the size of the cluster volume and the amount of changes you make during the retention period. However, for billing purposes it doesn't exceed the cumulative cluster volume size during the retention period. For example, if your cluster's `VolumeBytesUsed` size is

107,374,182,400 bytes (100 GiB), and your retention period is two days, the maximum value for `BackupRetentionPeriodStorageUsed` is 214,748,364,800 bytes (100 GiB + 100 GiB).

- `SnapshotStorageUsed` represents the amount of backup storage used, in bytes, for storing manual snapshots beyond the backup retention period. Manual snapshots don't count against your snapshot backup storage while their creation timestamp is within the retention period. All automatic snapshots also don't count against your snapshot backup storage. The size of each snapshot is the size of the cluster volume at the time you take the snapshot. The `SnapshotStorageUsed` value depends on the number of snapshots you keep and the size of each snapshot. For example, suppose you have one manual snapshot outside the retention period, and the cluster's `VolumeBytesUsed` size was 100 GiB when that snapshot was taken. The amount of SnapshotStorageUsed is 107,374,182,400 bytes (100 GiB).
- `TotalBackupStorageBilled` represents the sum, in bytes, of `BackupRetentionPeriodStorageUsed` and `SnapshotStorageUsed`, minus an amount of free backup storage, which equals the size of the cluster volume for one day. The free backup storage is equal to the latest volume size. For example if your cluster's `VolumeBytesUsed` size is 100 GiB, your retention period is two days, and you have one manual snapshot outside the retention period, the `TotalBackupStorageBilled` is 214,748,364,800 bytes (200 GiB + 100 GiB - 100 GiB).

You can monitor a Neptune cluster and build reports using CloudWatch metrics through the CloudWatch console. For more information about how to use CloudWatch metrics, see Monitoring Neptune and the table of metrics in Neptune CloudWatch metrics.

Restoring data from a Neptune backup

You can recover your data by creating a new Neptune DB cluster from the backup data that Neptune retains, or from a DB cluster snapshot that you have saved. You can quickly restore a new copy of a DB cluster created from backup data to any point in time during your backup retention period. The continuous and incremental nature of Neptune backups during the backup retention period means you don't need to take frequent snapshots of your data to improve restore times.

To determine the latest or earliest restorable time for a DB instance, look for the `Latest Restorable Time` or `Earliest Restorable Time` values on the Neptune console. The latest restorable time for a DB cluster is the most recent point at which you can restore your DB cluster, typically within 5 minutes of the current time. The earliest restorable time specifies how far back within the backup retention period that you can restore your cluster volume.

You can determine when the restore of a DB cluster is complete by checking the `Latest Restorable Time` and `Earliest Restorable Time` values. The `Latest Restorable Time` and `Earliest Restorable Time` values return NULL until the restore operation is complete. You can't request a backup or restore operation if `Latest Restorable Time` or `Earliest Restorable Time` returns NULL.

To restore a DB instance to a specified time using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Instances**. Choose the primary instance for the DB cluster that you want to restore.
3. Choose **Instance actions**, and then choose **Restore to point in time**.

   In the **Launch DB Instance** window, choose **Custom** under **Restore time**.
4. Specify the date and time that you want to restore to under **Custom**.
5. Type a name for the new, restored DB instance for **DB instance identifier** under **Settings**.
6. Choose **Launch DB Instance** to launch the restored DB instance.

   A new DB instance is created with the name you specified, and a new DB cluster is created. The DB cluster name is the new DB instance name followed by `-cluster`. For example, if the new DB instance name is `myrestoreddb`, the new DB cluster name is `myrestoreddb-cluster`.

Backup window in Neptune

Automated backups occur daily during the preferred backup window. If the backup requires more time than allotted to the backup window, the backup continues after the window ends, until it finishes. The backup window can't overlap with the weekly maintenance window for the DB instance.

During the automatic backup window, storage I/O might be suspended briefly while the backup process initializes (typically under a few seconds). You might experience elevated latencies for a few minutes during backups for Multi-AZ deployments.

The backup window is normally selected at random from an eight-hour block of time per Region by the Amazon RDS control plane underlying Neptune. The time blocks for each Region from which

the default backups windows are assigned is documented in the Backup Window section of the Amazon RDS User Guide.

Creating a DB Cluster Snapshot in Neptune

Neptune creates a storage volume snapshot of your DB cluster, backing up the entire DB cluster and not just individual databases. When you create a DB cluster snapshot, you need to identify which DB cluster you are going to back up. Then give your DB cluster snapshot a name so that you can restore from it later. The amount of time it takes to create a DB cluster snapshot varies with the size of your databases. The snapshot includes the entire storage volume. So the size of files (such as temporary files) also affects the amount of time it takes to create the snapshot.

You can create a DB cluster snapshot using the AWS Management Console, the AWS CLI, or the Neptune API.

Using the Console to Create a DB Cluster Snapshot

To create a DB cluster snapshot

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Databases**.
3. In the list of DB instances, choose the primary instance for the DB cluster.
4. Choose **Instance actions**, and then choose **Take snapshot**.

   The **Take DB Snapshot** window appears.
5. Enter the name of the DB cluster snapshot in the **Snapshot name** box.
6. Choose **Take Snapshot**.

Restoring from a DB Cluster Snapshot

When you create an Amazon Neptune snapshot of a DB cluster, Neptune creates a storage volume snapshot of the cluster, backing up all its data and not just individual instances. You can later create a new DB cluster by restoring from this DB cluster snapshot. When you restore the DB cluster, you provide the name of the DB cluster snapshot to restore from, and then provide a name for the new DB cluster that is created by the restore.

Contents

- Things to keep in mind about restoring a Neptune DB cluster from a snapshot
  - You cannot restore to an existing DB cluster
  - No instances are restored
  - No custom parameter group is restored
  - No custom security groups are restored
  - You can't restore from a shared encrypted snapshot
  - A restored DB cluster uses as much storage as before
- How to restore from a snapshot
  - Using the Console to Restore from a Snapshot

Things to keep in mind about restoring a Neptune DB cluster from a snapshot

You cannot restore to an existing DB cluster

The restore process always creates a new DB cluster, so you can't restore to a DB cluster that already exists.

No instances are restored

A new DB cluster created by a restore has no instances associated with it.

As soon as the restore is complete and your new DB cluster is available, explicitly create the instances that you will need. You can do this on the Neptune console, or using the CreateDBInstance API.

No custom parameter group is restored

A new DB cluster created by a restore automatically has the default DB parameter group associated with it.

As soon as the restore is complete and your new DB cluster is available, associate any custom DB parameter group that the instance you restored from was using. To do that, use the **Modify** command on the Neptune console, or the ModifyDBInstance API.

> **Important**
>
> We recommend that you save a custom parameter group being used in a DB cluster that you are creating a snapshot of. Then, when you restore from that snapshot, you can easily associate the correct parameter group with the restored DB cluster.

No custom security groups are restored

A new DB cluster created by a restore automatically has the default security group associated with it.

As soon as the restore is complete and your new DB cluster is available, associate any custom security groups that the instance you restored from was using. To do that, use the **Modify** command on the Neptune console, or the ModifyDBInstance API.

You can't restore from a shared encrypted snapshot

You cannot restore a DB cluster from a DB cluster snapshot that is both shared and encrypted.

Instead, make an unshared copy of the snapshot and restore from the copy.

For more information about handling encryption when copying a snapshot, see Handling Encryption When Copying Snapshots. For more information about sharing encrypted snapshots, see Sharing an Encrypted DB Cluster Snapshot.

A restored DB cluster uses as much storage as before

When you restore a DB cluster from a DB cluster snapshot, the amount of storage allocated to the new cluster is the same as was allocated to the DB cluster from which the snapshot was made, regardless of how much of that allocated storage is actually being used.

In other words, the "high water mark" for which you are billed does not change. Resetting the high water mark requires exporting the data from your graph and then reloading it onto a new DB cluster (see Neptune storage billing).

How to restore from a snapshot

You can restore a DB cluster from a DB cluster snapshot using the AWS Management Console, the AWS CLI, or the Neptune API.

Using the Console to Restore from a Snapshot

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Snapshots**.
3. Choose the DB cluster snapshot that you want to restore from.
4. Choose **Actions**, **Restore Snapshot**.
5. On the **Restore DB Instance** page, in the **DB Instance Identifier** box, enter the name for your restored DB cluster.
6. Choose **Restore DB Instance**.
7. If you want to restore the functionality of the DB cluster to that of the DB cluster that the snapshot was created from, you must modify the DB cluster to use the security group. The next steps assume that your DB cluster is in a virtual private cloud (VPC). If your DB cluster is not in a VPC, use the Amazon EC2 console to locate the security group that you need for the DB cluster.

    a. Open the Amazon VPC console at https://eusc-de-east-1.console.amazonaws-eusc.eu/vpc/.
    b. In the navigation pane, choose **Security Groups**.
    c. Choose the security group that you want to use for your DB clusters. If necessary, add rules to link the security group to a security group for an EC2 instance.

Copying a DB Cluster Snapshot

With Neptune, you can copy automated or manual DB cluster snapshots. After you copy a snapshot, the copy is a manual snapshot.

Copying an automated snapshot to another AWS account is a two-step process: First, you create a manual snapshot from the automated snapshot, and then you copy the manual snapshot to the other account.

As an alternative to copying, you can also share manual snapshots with other AWS accounts. For more information, see Sharing a DB Cluster Snapshot.

Topics

- Limitations on Copying a Snapshot
- Retention of DB Cluster Snapshot Copies
- Handling Encryption When Copying Snapshots
- Copying Snapshots Across AWS Regions
- Copying a DB Cluster Snapshot Using the Console
- Copying a DB Cluster Snapshot Using the AWS CLI

Limitations on Copying a Snapshot

The following are some limitations when you copy snapshots:

- If you delete a source snapshot before the target snapshot becomes available, the snapshot copy might fail. Verify that the target snapshot has a status of `AVAILABLE` before you delete a source snapshot.

Retention of DB Cluster Snapshot Copies

Neptune deletes automated snapshots as follows:

- At the end of their retention period.
- When you disable automated snapshots for a DB cluster.
- When you delete a DB cluster.

If you want to keep an automated snapshot for a longer period, copy it to create a manual snapshot, which is then retained until you delete it. Neptune storage costs might apply to manual snapshots if they exceed your default storage space.

Handling Encryption When Copying Snapshots

You can copy a snapshot that has been encrypted using an AWS KMS encryption key. If you copy an encrypted snapshot, the copy of the snapshot must also be encrypted. You can encrypt the copy with the same AWS KMS encryption key as the original snapshot, or you can specify a different AWS KMS encryption key.

You cannot encrypt an unencrypted DB cluster snapshot when you copy it.

For Amazon Neptune DB cluster snapshots, you can also leave the DB cluster snapshot unencrypted and instead specify a AWS KMS encryption key when restoring. The restored DB cluster is encrypted using the specified key.

Copying Snapshots Across AWS Regions

Copying a DB Cluster Snapshot Using the Console

If your source database engine is Neptune, then your snapshot is a DB cluster snapshot. For each AWS account, you can copy up to five DB cluster snapshots at a time per AWS Region. Copying both encrypted and unencrypted DB cluster snapshots is supported.

To cancel a copy operation after it is in progress, delete the target DB cluster snapshot while that DB cluster snapshot is in **copying** status.

The following procedure works for copying encrypted or unencrypted DB cluster snapshots:

To copy a DB cluster snapshot

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Snapshots**.
3. Select the check box for the DB cluster snapshot you want to copy.
4. Choose **Actions**, and then choose **Copy Snapshot**. The **Make Copy of DB Snapshot** page appears.

5. Enter the name of the DB cluster snapshot copy in **New DB Snapshot Identifier**.
6. To copy tags and values from the snapshot to the copy of the snapshot, choose **Copy Tags**.
7. For **Enable Encryption**, choose one of the following options:
   - Choose **Disable encryption** if the DB cluster snapshot isn't encrypted and you don't want to encrypt the copy.
   - Choose **Enable encryption** if the DB cluster snapshot isn't encrypted but you want to encrypt the copy. In this case, for **Master Key**, specify the AWS KMS key identifier to use to encrypt the DB cluster snapshot copy.
   - Choose **Enable encryption** if the DB cluster snapshot is encrypted. In this case, you must encrypt the copy, so **Yes** is already selected. For **Master Key**, specify the AWS KMS key identifier to use to encrypt the DB cluster snapshot copy.
8. Choose **Copy Snapshot**.

Copying a DB Cluster Snapshot Using the AWS CLI

You can copy a DB snapshot using the copy-db-cluster-snapshot AWS CLI command.

If you are copying the snapshot to a new AWS Region, run the command in the new Region.

Use the following parameter descriptions and examples to determine which parameters to use in copying a snapshot with the AWS CLI.

- `--source-db-cluster-snapshot-identifier` – The identifier for the source DB snapshot.
  -
  - If you are copying from a shared manual DB snapshot, this parameter must be the Amazon Resource Name (ARN) of the shared DB snapshot.
  - If you are copying an encrypted snapshot, this parameter must be in the ARN format for the source AWS Region, and must match the `SourceDBSnapshotIdentifier` in the `PreSignedUrl` parameter.
- `--target-db-cluster-snapshot-identifier` – The identifier for the new copy of the encrypted DB snapshot.
- `--kms-key-id` – The AWS KMS key ID for an encrypted DB snapshot. The AWS KMS key ID is the Amazon Resource Name (ARN), AWS KMS key identifier, or the AWS KMS key alias for the AWS KMS encryption key.

- If you copy an encrypted DB snapshot from your AWS account, you can specify a value for this parameter to encrypt the copy with a new AWS KMS encryption key. If you don't specify a value for this parameter, then the copy of the DB snapshot is encrypted with the same AWS KMS key as the source DB snapshot.
- You cannot use this parameter to create an encrypted copy of an unencrypted snapshot. Trying to do so will generate an error.
-
-

Example From Unencrypted, To Same Region

The following code creates a copy of a snapshot, with the new name `mydbsnapshotcopy`, from the `us-east-1` AWS region to the `us-west-2` region.

For Linux, OS X, or Unix:

```
aws neptune copy-db-cluster-snapshot \
  --source-db-cluster-snapshot-identifier instance1-snapshot-20130805 \
  --target-db-cluster-snapshot-identifier mydbsnapshotcopy
```

For Windows:

```
aws neptune copy-db-cluster-snapshot ^
  --source-db-cluster-snapshot-identifier instance1-snapshot-20130805 ^
  --target-db-cluster-snapshot-identifier mydbsnapshotcopy
```

Sharing a DB Cluster Snapshot

Using Neptune, you can share a manual DB cluster snapshot in the following ways:

- Sharing a manual DB cluster snapshot, whether encrypted or unencrypted, enables authorized AWS accounts to copy the snapshot.
- Sharing an unencrypted manual DB cluster snapshot enables authorized AWS accounts to directly restore a DB cluster from the snapshot instead of taking a copy of it and restoring from that. Encrypted snapshots cannot be restored directly; they must be copied first and then restored from the copy.

> **Note**
>
> To share an automated DB cluster snapshot, create a manual DB cluster snapshot by copying the automated snapshot, and then share that copy.

For more information about restoring a DB cluster from a DB cluster snapshot, see How to restore from a snapshot.

You can share a manual snapshot with up to 20 other AWS accounts. You can also share an unencrypted manual snapshot as public, which makes the snapshot available to all AWS accounts. Take care when sharing a snapshot as public so that none of your private information is included in any of your public snapshots.

> **Note**
>
> When you restore a DB cluster from a shared snapshot using the AWS Command Line Interface (AWS CLI) or Neptune API, you must specify the Amazon Resource Name (ARN) of the shared snapshot as the snapshot identifier.

Topics

- Sharing an Encrypted DB Cluster Snapshot
- Sharing a DB Cluster Snapshot

Sharing an Encrypted DB Cluster Snapshot

You can share DB cluster snapshots that have been encrypted "at rest" using the AES-256 encryption algorithm. For more information, see Encrypting Neptune resources at rest. To do this, you must take the following steps:

1. Share the AWS Key Management Service (AWS KMS) encryption key that was used to encrypt the snapshot with any accounts that you want to be able to access the snapshot.

   You can share AWS KMS encryption keys with another AWS account by adding the other account to the KMS key policy. For details on updating a key policy, see Key Policies in the *AWS KMS Developer Guide*. For an example of creating a key policy, see Creating an IAM Policy to Enable Copying of the Encrypted Snapshot later in this topic.
2. Use the AWS Management Console, AWS CLI, or Neptune API to share the encrypted snapshot with the other accounts.

These restrictions apply to sharing encrypted snapshots:

- You cannot share encrypted snapshots as public.
- You cannot share a snapshot that has been encrypted using the default AWS KMS encryption key of the AWS account that shared the snapshot.

Allowing Access to an AWS KMS Encryption Key

For another AWS account to copy an encrypted DB cluster snapshot shared from your account, the account that you share your snapshot with must have access to the KMS key that encrypted the snapshot. To allow another AWS account access to an AWS KMS key, update the key policy for the KMS key with the ARN of the AWS account that you are sharing to as a `Principal` in the KMS key policy. Then allow the `kms:CreateGrant` action. See Allowing users in other accounts to use a KMS key in the *AWS Key Management Service Developer Guide* for general instructions.

After you have given an AWS account access to your KMS encryption key, to copy your encrypted snapshot, that AWS account must create an IAM user if it doesn’t already have one. KMS security restrictions don't permit use of a root AWS account identity for this. The AWS account must also attach an IAM policy to that IAM user that allows the IAM user to copy an encrypted DB cluster snapshot using your KMS key.

In the following key policy example, user `111122223333` is the owner of the KMS encryption key, and user `444455556666` is the account that the key is being shared with. This updated key policy gives the AWS account access to the KMS key by including the ARN for the root AWS account identity for user `444455556666` as a `Principal` for the policy, and by allowing the `kms:CreateGrant` action.

JSON

```
{
    "Id": "key-policy-1",
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowUseOfTheKey",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws-eusc:iam::111122223333:user/KeyUser",
                    "arn:aws-eusc:iam::444455556666:root"
                ]
            },
            "Action": [
                "kms:CreateGrant",
                "kms:Encrypt",
                "kms:Decrypt",
                "kms:ReEncrypt*",
                "kms:GenerateDataKey*",
                "kms:DescribeKey"
            ],
            "Resource": "*"
        },
        {
            "Sid": "AllowAttachmentOfPersistentResources",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws-eusc:iam::111122223333:user/KeyUser",
                    "arn:aws-eusc:iam::444455556666:root"
                ]
            },
            "Action": [
                "kms:CreateGrant",
```

```
                "kms:ListGrants",
                "kms:RevokeGrant"
            ],
            "Resource": "*",
            "Condition": {
                "Bool": {
                    "kms:GrantIsForAWSResource": true
                }
            }
        }
    ]
}
```

Creating an IAM Policy to Enable Copying of the Encrypted Snapshot

After the external AWS account has access to your KMS key, the owner of that account can create a policy that allows an IAM user created for the account to copy an encrypted snapshot encrypted with that KMS key.

The following example shows a policy that can be attached to an IAM user for AWS account `444455556666`. It enables the IAM user to copy a shared snapshot from AWS account `111122223333` that has been encrypted with the KMS key `c989c1dd-a3f2-4a5d-8d96-e793d082ab26` in the `us-west-2` Region.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowUseOfTheKey",
            "Effect": "Allow",
            "Action": [
                "kms:Encrypt",
                "kms:Decrypt",
                "kms:ReEncrypt*",
                "kms:GenerateDataKey*",
                "kms:DescribeKey",
                "kms:CreateGrant",
                "kms:RetireGrant"
            ],
```

```
            "Resource": ["arn:aws-eusc:kms:us-west-2:111122223333:key/c989c1dd-
a3f2-4a5d-8d96-e793d082ab26"]
        },
        {
            "Sid": "AllowAttachmentOfPersistentResources",
            "Effect": "Allow",
            "Action": [
                "kms:CreateGrant",
                "kms:ListGrants",
                "kms:RevokeGrant"
            ],
            "Resource": ["arn:aws-eusc:kms:us-west-2:111122223333:key/c989c1dd-
a3f2-4a5d-8d96-e793d082ab26"],
            "Condition": {
                "Bool": {
                    "kms:GrantIsForAWSResource": true
                }
            }
        }
    ]
}
```

For details on updating a key policy, see Key Policies in the *AWS Key Management Service Developer Guide*.

Sharing a DB Cluster Snapshot

You can share a DB cluster snapshot using the AWS Management Console, the AWS CLI, or the Neptune API.

Using the Console to Share a DB Cluster Snapshot

Using the Neptune console, you can share a manual DB cluster snapshot with up to 20 AWS accounts. You can also stop sharing a manual snapshot with one or more accounts.

To share a manual DB cluster snapshot

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Snapshots**.
3. Choose the manual snapshot that you want to share.

4. Choose **Actions**, **Share Snapshot**.
5. Choose one of the following options for **DB snapshot visibility**.

   - If the source is unencrypted, choose **Public** to permit all AWS accounts to restore a DB cluster from your manual DB cluster snapshot. Or choose **Private** to permit only AWS accounts that you specify to restore a DB cluster from your manual DB cluster snapshot.

     > **Warning**
     >
     > If you set **DB snapshot visibility** to **Public**, all AWS accounts can restore a DB cluster from your manual DB cluster snapshot and have access to your data. Do not share any manual DB cluster snapshots that contain private information as **Public**.

   - If the source is encrypted, **DB snapshot visibility** is set as **Private** because encrypted snapshots can't be shared as public.
6. For **AWS Account ID**, enter the AWS account identifier for an account that you want to permit to restore a DB cluster from your manual snapshot. Then choose **Add**. Repeat to include additional AWS account identifiers, up to 20 AWS accounts.

   If you make an error when adding an AWS account identifier to the list of permitted accounts, you can delete it from the list by choosing **Delete** at the right of the incorrect AWS account identifier.
7. After you add identifiers for all of the AWS accounts that you want to permit to restore the manual snapshot, choose **Save**.

To stop sharing a manual DB cluster snapshot with an AWS account

1. Open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Snapshots**.
3. Choose the manual snapshot that you want to stop sharing.
4. Choose **Actions**, and then choose **Share Snapshot**.
5. To remove permission for an AWS account, choose **Delete** for the AWS account identifier for that account from the list of authorized accounts.
6. Choose **Save**.

Deleting a Neptune Snapshot

You can delete a DB snapshot using the AWS Management Console, the AWS CLI, or the Neptune management API:

Deleting Using the Console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/home.
2. In the navigation pane, choose **Snapshots**.
3. Choose the DB snapshot that you want to delete.
4. For **Actions**, choose **Delete Snapshot**.
5. Choose **Delete** on the confirmation page.

Deleting Using the AWS CLI

You can also delete a DB snapshot using the AWS CLI delete_db_cluster_snapshot command, using the `--db-snapshot-identifier` parameter to identify the snapshot you want to delete:

For Linux, OS X, or Unix:

```
aws neptune delete-db-cluster-snapshot \
    --db-snapshot-identifier <name-of-the-snapshot-to-delete>
```

For Windows:

```
aws neptune delete-db-cluster-snapshot ^
    --db-snapshot-identifier <name-of-the-snapshot-to-delete>
```

Deleting Using the Neptune Management API

You can use one of the SDKs to delete a DB snapshot by calling the DeleteDBClusterSnapshot API and use the `DBSnapshotIdentifier` parameters to identify the DB snapshot to be deleted.

Best practices: getting the most out of Neptune

The following are some general recommendations for working with Amazon Neptune. Use this information as a reference to quickly find recommendations for using Amazon Neptune and maximizing performance.

Contents

- Amazon Neptune basic operational guidelines
  - Amazon Neptune security best practices
  - Avoid different instance classes in a cluster
  - Avoid repeated restarts during bulk loading
  - Enable the OSGP Index if you have a large number of predicates
  - Avoid long-running transactions where possible
  - Best practices for using Neptune metrics
  - Best practices for tuning Neptune queries
  - Load balancing across read replicas
  - Loading faster using a temporary larger instance
  - Resize your writer instance by failing over to a read-replica
  - Retry upload after data prefetch task interrupted error
- General Best Practices for Using Gremlin with Neptune
  - Heartbeat Configuration for Neptune Serverless
  - Structure upsert queries to take advantage of the DFE engine
  - Test Gremlin code in the context where you will deploy it
  - Creating Efficient Multithreaded Gremlin Writes
  - Pruning Records with the Creation Time Property
  - Using the datetime( ) Method for Groovy Time Data
  - Using Native Date and Time for GLV Time Data
- Best practices using the Gremlin Java client with Neptune
  - Re-use the client object across multiple threads
  - Create separate Gremlin Java client objects for read and write endpoints
  - Add multiple read replica endpoints to a Gremlin Java connection pool

- Close the client to avoid the connections limit
- Create a new connection after failover
- Set maxInProcessPerConnection and maxSimultaneousUsagePerConnection to the same value
- Send queries to the server as bytecode rather than as strings
- Always completely consume the ResultSet or Iterator returned by a query
- Bulk add vertices and edges in batches
- Disable DNS caching in the Java Virtual Machine
- Optionally, set timeouts at a per-query level
- Troubleshooting java.util.concurrent.TimeoutException
- Neptune Best Practices Using openCypher and Bolt
  - Create a new connection after failover
  - Connection handling for long-lived applications
  - Connection handling for AWS Lambda
  - Prefer directed to bi-directional edges in queries
  - Neptune does not support multiple concurrent queries in a transaction
  - Close driver objects when you're done
  - Use explicit transaction modes for reading and writing
    - Read-only transactions
    - Mutation transactions
  - Retry logic for exceptions
  - Set multiple properties at once using a single SET clause
    - Use the SET clause to remove multiple properties at once
  - Use parameterized queries
  - Use flattened maps instead of nested maps in UNWIND clause
  - Place more restrictive nodes on the left side in Variable-Length Path (VLP) expressions
  - Avoid redundant node label checks by using granular relationship names
  - Specify edge labels where possible
  - Avoid using the WITH clause when possible
  - Place restrictive filters as early in the query as possible

  - Explicitly check whether properties exist

- Do not use named path (unless it is required)
- Avoid COLLECT(DISTINCT())
- Prefer the properties function over individual property lookup when retrieving all property values
- Perform static computations outside of the query
- Batch inputs using UNWIND instead of individual statements
- Prefer using custom IDs for node/relationship
- Avoid doing ~id computations in the query
- Updating/Merging multiple nodes
- Neptune Best Practices Using SPARQL
  - Querying All Named Graphs by Default
  - Specifying a Named Graph for Load
  - Choosing Between FILTER, FILTER...IN, and VALUES in Your Queries
- Best practices for using Neptune with Amazon ECS and Amazon EKS
  - Manage connection pools and lifecycle in containerized environments
  - Handle host replacement and connection stalling
  - Configure networking, security groups, and IAM authentication
- Best practices for using the AWS CLI and SDKs with Neptune

Amazon Neptune basic operational guidelines

The following are basic operational guidelines that you should follow when working with Neptune.

- Understand Neptune DB instances so that you can size them appropriately for your performance and use-case requirements. See Amazon Neptune DB Clusters and Instances.
- Monitor your CPU and memory usage. This helps you know when to migrate to a DB instance class with greater CPU or memory capacity to achieve the query performance that you require. You can set up Amazon CloudWatch to notify you when usage patterns change or when you approach the capacity of your deployment. Doing so can help you maintain system performance and availability. See Monitoring instances and Monitoring Neptune for details.

Because Neptune has its own memory manager, it is normal to see relatively low memory usage even when CPU usage is high. Encountering out-of-memory exceptions when executing queries is the best indicator that you need to increase freeable memory.

- Enable automatic backups and set the backup window to occur at a convenient time.
- Test failover for your DB instance to understand how long the process takes for your use case. It also helps ensure that the application that accesses your DB instance can automatically connect to the new DB instance after failover.
- When you create a read-replica instance, it should be at least as large as the primary writer instance. This helps keep replication lag in check, and avoids replica restarts. See Avoid different instance classes in a cluster.
- Before upgrading to a new major engine version, be sure to test your application on it before you upgrade. You can do this by cloning your DB cluster so that the clone cluster runs the new engine version, and then test your application on the clone.
- To facilitate failovers, all instances should ideally be the same size.

Topics

- Amazon Neptune security best practices
- Avoid different instance classes in a cluster
- Avoid repeated restarts during bulk loading
- Enable the OSGP Index if you have a large number of predicates
- Avoid long-running transactions where possible
- Best practices for using Neptune metrics
- Best practices for tuning Neptune queries
- Load balancing across read replicas
- Loading faster using a temporary larger instance
- Resize your writer instance by failing over to a read-replica
- Retry upload after data prefetch task interrupted error

Amazon Neptune security best practices

Use AWS Identity and Access Management (IAM) accounts to control access to Neptune API actions. Control actions that create, modify, or delete Neptune resources (such as DB instances, security

groups, option groups, or parameter groups), and actions that perform common administrative actions (such as backing up and restoring DB instances).

- Use temporary rather than persistent credentials whenever possible.
- Assign an individual IAM account to each person who manages Amazon Relational Database Service (Amazon RDS) resources. Never use AWS account root users to manage Neptune resources. Create an IAM user for everyone, including yourself.
- Grant each user the minimum set of permissions required to perform their duties.
- Use IAM groups to effectively manage permissions for multiple users.
- Rotate your IAM credentials regularly.

For more information about using IAM to access Neptune resources, see Securing your Amazon Neptune database. For general information about working with IAM, see AWS Identity and Access Management and IAM Best Practices in the *IAM User Guide*.

Avoid different instance classes in a cluster

When your DB cluster contains instances of different classes, problems can occur over time. The most common problem is that a small reader instance can get into a cycle of repeated restarts because of replication lag. If a reader node has a weaker DB instance class configuration than that of a writer DB instance, the volume of changes can be too big for the reader to catch up.

> **Important**
>
> To avoid repeated restarts caused by replication lag, configure your DB cluster so that all instances have the same instance class (size).

You can see the lag between the writer instance (the primary) and the readers in your DB cluster using the `ClusterReplicaLag` metric in Amazon CloudWatch. The `VolumeWriteIOPs` metric also lets you detect bursts of write activity in your cluster that can create replication lag.

Avoid repeated restarts during bulk loading

If you experience a cycle of repeated read-replica restarts because of replication lag during a bulk load, your replicas are likely unable to keep up with the writer in your DB cluster.

Either scale the readers to be larger than the writer, or temporarily remove them during the bulk load and then recreate them after it completes.

Enable the OSGP Index if you have a large number of predicates

If your data model contains a large number of distinct predicates (more than thousand in most cases), you may experience reduced performance and higher operational costs.

If this is the case, you can improve performance by enabling the OSGP index. See The OSGP index.

Avoid long-running transactions where possible

Long-running transactions, either read-only or read-write, can cause unexpected problems of the following kinds:

A long-running transaction on a reader instance or on a writer instance with concurrent writes can result in a large accumulation of different versions of data. This can introduce higher latencies for read queries that filter out a large portion of their results.

In some cases, the accumulated versions over hours can cause new writes to be throttled.

A long-running read-write transaction with many writes can also cause issues if the instance restarts. If an instance restarts from a maintenance event or a crash, all uncommitted writes are rolled back. Such undo operations typically run in the background and do not block the instance from coming back up, but any new writes that conflict with the operations being rolled back then fail.

For example, if the same query is retried after the connection was severed in the previous run might fail when the instance is restarted.

The time needed for undo operations is proportional to the size of the changes involved.

Best practices for using Neptune metrics

To identify performance issues caused by insufficient resources and other common bottlenecks, you can monitor the metrics available for your Neptune DB cluster.

Monitor performance metrics on a regular basis to gather data about the average, maximum, and minimum values for a variety of time ranges. This helps identify when performance is degraded.

Using this data, you can set Amazon CloudWatch alarms for particular metric thresholds so you are alerted if they are reached.

When you set up a new DB cluster and get it running with a typical workload, try to capture the average, maximum, and minimum values of all of the performance metrics at a number of different intervals (for example, one hour, 24 hours, one week, two weeks). This gives you an idea of what is normal. It helps to get comparisons for both peak and off-peak hours of operation. You can then use this information to identify when performance is dropping below standard levels, and can set alarms accordingly.

See Monitoring Neptune Using Amazon CloudWatch for information about how to view Neptune metrics.

The following are the most important metrics to start with:

- **BufferCacheHitRatio** — The percentage of requests that are served by the buffer cache. Cache misses add significant latency to query execution. If the cache hit ratio is below 99.9% and latency is an issue for your application, consider upgrading the instance type to cache more data in memory.
- **CPU utilization** — Percentage of computer processing capacity used. High values for CPU consumption might be appropriate, depending on your query-performance goals.
- **Freeable memory** — How much RAM is available on the DB instance, in megabytes. Neptune has its own memory manager, so this metric may be lower than you expect. A good sign that you should consider upgrading your instance class to one with more RAM is if queries often throw out-of-memory exceptions.

The red line in the **Monitoring** tab metrics is marked at 75% for CPU and Memory Metrics. If instance memory consumption frequently crosses that line, check your workload and consider upgrading your instance to improve query performance.

Best practices for tuning Neptune queries

One of the best ways to improve Neptune performance is to tune your most commonly used and most resource-intensive queries to make them less expensive to run.

For information about how to tune Gremlin queries, see Gremlin query hints and Tuning Gremlin queries . For information about how to tune SPARQL queries, see SPARQL query hints.

Load balancing across read replicas

The reader endpoint round-robin routing works by changing the host that the DNS entry points to. The client must create a new connection and resolve the DNS record to get a connection to a new read replica, because WebSocket connections are often kept alive for long periods.

To get different read replicas for successive requests, ensure that your client resolves the DNS entry each time it connects. This may require closing the connection and reconnecting to the reader endpoint.

You can also load balance requests across read replicas by connecting to instance endpoints explicitly.

Loading faster using a temporary larger instance

Your load performance increases with larger instance sizes. If you're not using a large instance type, but you want increased load speeds, you can use a larger instance to load and then delete it.

> **Note**
>
> The following procedure is for a new cluster. If you have an existing cluster, you can add a new larger instance and then promote it to a primary DB instance.

To load data using a larger instance size

1. Create a cluster with a single `r5.12xlarge` instance. This instance is the primary DB instance.
2. Create one or more read replicas of the same size (`r5.12xlarge`).

   You can create the read replicas in a smaller size, but if they are not large enough to keep up with writes made by the primary instance, they may have to restart frequently. The resulting downtime reduces performance dramatically.
3. In the bulk loader command, include `"parallelism" : "OVERSUBSCRIBE"` to tell Neptune to use all available CPU resources for loading (see Neptune Loader Request Parameters). The load operation will then proceed as fast as I/O permits, which generally requires 60-70% of CPU resources.
4. Load your data using the Neptune loader. The load job runs on the primary DB instance.

5. After the data is finished loading, be sure to scale all the instances in the cluster down to the same instance type to avoid additional charges and repeated restart problems (see Avoid different instance sizes).

Resize your writer instance by failing over to a read-replica

The best way to resize an instance in your DB cluster, including the writer instance, is to create or modify a read-replica instance so that it has the size your want, and then deliberately fail over to that read-replica. The downtime seen by your application is only the time required to change the writer's IP address, which should be around 3 to 5 seconds.

The Neptune management API that you use to fail over the current writer instance to a read-replica instance deliberately is FailoverDBCluster. If you are using the Gremlin Java client, you may need to create a new Client object after the failover to pick up the new IP address, as mentioned here.

Make sure to change all your instances to the same size so that you avoid a cycle of repeated restarts, as mentioned below.

Retry upload after data prefetch task interrupted error

When you are loading data into Neptune using the bulk loader, a `LOAD_FAILED` status may occasionally result, with a `PARSING_ERROR` and `Data prefetch task interrupted` message reported in response to a request for detailed information, like this:

```
"errorLogs" : [
  {
    "errorCode" : "PARSING_ERROR",
    "errorMessage" : "Data prefetch task interrupted: Data prefetch task for 11467
 failed",
    "fileName" : "s3://amzn-s3-demo-bucket/some-source-file",
    "recordNum" : 0
  }
]
```

If you encounter this error, just retry the bulk upload request again.

The error occurs when there was a temporary interruption that was typically not caused by your request or your data, and it can usually be resolved by running the bulk upload request again.

If you are using default settings, namely `"mode":"AUTO"`, and `"failOnError":"TRUE"`, the loader skips the files that it already successfully loaded and resumes loading files it had not yet loaded when the interruption occurred.

General Best Practices for Using Gremlin with Neptune

Follow these recommendations when using the Gremlin graph traversal language with Neptune. For information about using Gremlin with Neptune, see the section called "Gremlin".

Important

A change was made in TinkerPop version 3.4.11 that improves correctness of how queries are processed, but for the moment can sometimes seriously impact query performance. For example, a query of this sort may run significantly slower:

```
g.V().hasLabel('airport').
  order().
    by(out().count(),desc).
  limit(10).
  out()
```

The vertices after the limit step are now fetched in a non-optimal way because of the TinkerPop 3.4.11 change. To avoid this, you can modify the query by adding the barrier() step at any point after the `order().by()`. For example:

```
g.V().hasLabel('airport').
  order().
    by(out().count(),desc).
  limit(10).
  barrier().
  out()
```

TinkerPop 3.4.11 was enabled in Neptune engine version 1.0.5.0.

Topics

- Heartbeat Configuration for Neptune Serverless
- Structure upsert queries to take advantage of the DFE engine

- Test Gremlin code in the context where you will deploy it
- Creating Efficient Multithreaded Gremlin Writes
- Pruning Records with the Creation Time Property
- Using the datetime( ) Method for Groovy Time Data
- Using Native Date and Time for GLV Time Data

Heartbeat Configuration for Neptune Serverless

When using Gremlin WebSocket clients with Neptune Serverless, you need to configure the client's ping interval appropriately to maintain stable connections during scaling events. The Gremlin client uses WebSocket connections and sends periodic pings to verify the connection is active. The client expects a response from the server within the ping interval timeframe. If the server doesn't respond, the client automatically closes the connection.

For Neptune **provisioned** instances, we recommend setting the ping interval to **5 seconds**. For Neptune **Serverless clusters**, we recommend setting the ping interval to at least **20 seconds** to accommodate potential delays during scaling operations. This parameter controls how long the client waits between writes to the server before sending a ping to verify the connection is still active.

The configuration of this parameter varies depending on the client implementation:

Java Client Configuration

For the Java TinkerPop Gremlin client, configure the `keepAliveInterval` parameter:

```
Cluster.Builder builder = Cluster.build()
    .addContactPoint(endpoint)
    .keepAliveInterval(20000); // Configure ping interval in milliseconds
```

For more details about the Java driver configuration, refer to the Java TinkerPop documentation.

Go Client Configuration

For the Gremlin Go client, configure the `KeepAliveInterval` parameter:

```
rc, err := driver.NewDriverRemoteConnection(endpoint,
    func(settings *driver.DriverRemoteConnectionSettings) {
        settings.TraversalSource = "g"
```

```
        settings.AuthInfo = auth
        settings.KeepAliveInterval = 20 * time.Second // Configure ping interval
        ...
    })
```

For more details about the Go driver configuration, refer to the Go TinkerPop documentation.

JavaScript/Node.js Client Configuration

For the JavaScript/Node.js Gremlin client, configure the `pingInterval` parameter:

```
const gremlin = require('gremlin');
const DriverRemoteConnection = gremlin.driver.DriverRemoteConnection;

const connection = new DriverRemoteConnection(endpoint, {
    traversalSource: 'g',
    pingInterval: 20000  // Configure ping interval in milliseconds
});
```

For more details about the JavaScript driver configuration, refer to the JavaScript TinkerPop documentation.

Python Client Configuration

For the Python Gremlin client, the ping interval is typically managed at the transport layer. Consult the specific transport implementation documentation for configuration options:

```
from gremlin_python.driver.driver_remote_connection import DriverRemoteConnection

g = traversal().with_remote(
    DriverRemoteConnection('wss://your-neptune-endpoint:your-neptune-port/gremlin','g',
        transport_factory=lambda: AiohttpTransport(read_timeout=60,
                                                   write_timeout=20,
                                                   heartbeat=20, // Configure
 heartbeat
                                                   call_from_event_loop=True,
                                                   max_content_length=100*1024*1024,

 ssl_options=ssl.create_default_context(Purpose.CLIENT_AUTH))))
```

For more details about the Python driver configuration, refer to the Python TinkerPop documentation.

This configuration ensures your client maintains connection stability during Neptune Serverless scaling events, preventing unnecessary connection closures and improving application reliability.

Structure upsert queries to take advantage of the DFE engine

Making efficient upserts with Gremlin `mergeV()` and `mergeE()` steps explains how to structure upsert queries to use the DFE engine as effectively as possible.

Test Gremlin code in the context where you will deploy it

Gremlin provides multiple ways for clients to submit queries to the server. These submission modes differ in how queries are evaluated and how transactions behave. These differences can lead to unexpected results if you develop in one mode and deploy in another.

Script mode (string-based submission)

In script mode, the client sends the entire query as a text string to the server. The server evaluates the complete string, including any *terminal steps*, and streams the results back to the client. The following tools and methods use script mode:

- Gremlin Console – when using the `:remote` command
- Neptune graph notebook
- HTTP with AWS SDK or AWS CLI
- Submitting query strings through TinkerPop drivers (for example, using `client.submit("g.V().count()")` in Java)

In script mode, the client sends a query like the following to the server as a complete string:

```
// Script mode - the entire string, including .next(), is sent to the server
// V('non-existing-id') yields nothing because no vertex with that ID exists
Cluster cluster = Cluster.build().addContactPoint("your-neptune-endpoint")
                         .port(8182).enableSsl(true).create();
Client client = cluster.connect();
client.submit("g.V('existing-id').addV('person').V('non-existing-id').next()");
```

The server evaluates the full query including `.next()`. If `.next()` finds no results, the server raises a `NoSuchElementException` and the transaction *rolls back*.

Bytecode mode (GLV traversal objects)

In bytecode mode, the client builds a traversal object locally using a Gremlin Language Variant (GLV). The driver serializes the traversal steps as bytecode and sends them to the server. Terminal steps such as `.toList()` or `.next()` execute on the *client side* to iterate over the results returned by the server. You can use bytecode mode with Java, Python, Go, .NET, JavaScript, and other third-party TinkerPop-compliant drivers suitable to the Neptune engine version used.

In bytecode mode, the same query looks like this:

```
// Bytecode mode - the driver sends the traversal steps as bytecode
// .next() executes on the client to iterate results
Cluster cluster = Cluster.build().addContactPoint("your-neptune-endpoint")
                         .port(8182).enableSsl(true).create();
GraphTraversalSource g = traversal().withRemote(
    DriverRemoteConnection.using(cluster));
g.V("existing-id").addV("person").V("non-existing-id").next();
```

The driver sends only the traversal steps (`g.V().addV().V()`) as bytecode. The server successfully evaluates the traversal, commits the transaction, and returns the result set. The client then calls `.next()` locally to read from the result set. If the result set is empty, the client raises a `NoSuchElementException`, but the transaction has *already committed* on the server.

Transaction behavior differences

The critical difference between these modes is how terminal steps affect transactions:

- **Script mode** – The server evaluates terminal steps. If a terminal step like `.next()` fails because the result set is empty, the server treats the query as failed and *rolls back* the transaction. The server does not persist any mutations in the same traversal (such as `addV()`).
- **Bytecode mode** – The client evaluates terminal steps. The server evaluates only the traversal steps, successfully commits the transaction, and returns results. If the client then calls `.next()` on an empty result set, the resulting `NoSuchElementException` is a client-side error only. The transaction has already committed, so the server *persists* any mutations.

Terminal steps

In Gremlin, terminal steps are the steps that cause a traversal to be submitted to Neptune for evaluation. In bytecode mode, the terminal step triggers the driver to serialize and send the traversal. In script mode, the terminal step is part of the query string evaluated on the server. The terminal steps are:

- `hasNext()` – Returns `true` if results are available, `false` otherwise.
- `next()` – Returns the next result. Throws `NoSuchElementException` if no results exist.
- `next(n)` – Returns the next *n* results as a list.
- `toList()` – Returns all results as a list. Returns an empty list if no results exist.
- `toSet()` – Returns all results as a set. Returns an empty set if no results exist.
- `iterate()` – Iterates all results without returning them. Use this for mutations where you do not need the return value.

> **Note**
>
> Individual language GLVs may provide additional terminal steps specific to their implementation. See the language-specific pages for details.

If you develop and test your code in one context, you can run into problems. For example, the Gremlin console submits queries as scripts. Your code might behave differently in production if you deploy it in a different context, such as through the Java driver using bytecode.

> **Important**
>
> Be sure to test Gremlin code using the same submission mode in which it will be deployed, to avoid unexpected transaction behavior.

Creating Efficient Multithreaded Gremlin Writes

There are a few guidelines for multithreaded loading of data into Neptune using Gremlin.

If possible, give each thread a set of vertices or edges to insert or modify that do not collide. For example, thread 1 addresses ID range 1–50,000, thread 2 addresses ID range 50,001–100,000, and so on. This reduces the chance of hitting a `ConcurrentModificationException`. To be safe, put a `try/catch` block around all writes. If any fail, you can retry them after a short delay. For guidance on developing a retry strategy with exponential backoff and jitter, see the section called "Exceptions and Retries".

Batching writes in a batch size between 50 and 100 (vertices or edges) generally works well. If you have a lot of properties being added for each vertex, a number closer to 50 than 100 might be a

better choice. Some experimentation is worthwhile. So for batched writes, you can use something like this:

```
g.addV(‘test’).property(id,’1’).as(‘a’).
  addV(‘test’).property(id,’2’).
  addE(‘friend’).to(‘a’).
```

This is then repeated in each batch operation.

Using batches is significantly more efficient than adding one vertex or edge per Gremlin round trip to the server.

If you are using a Gremlin Language Variant (GLV) client, you can create a batch programmatically by first creating a traversal. Then add to it, and finally, iterate over it; for example:

```
  t.addV(‘test’).property(id,’1’).as(‘a’)
  t.addV(‘test’).property(id,’2’)
  t.addE(‘friend’).to(‘a’)
  t.iterate()
```

It's best to use the Gremlin Language Variant client if possible. But you can do something similar with a client that submits queries as text strings by concatenating strings to build up a batch.

If you are using one of the Gremlin Client libraries rather than basic HTTP for queries, the threads should all share the same client, cluster, or connection pool. You might need to tune settings to get the best possible throughput—settings such as the size of the connection pool and the number of worker threads that the Gremlin client uses.

Pruning Records with the Creation Time Property

You can prune stale records by storing the creation time as a property on vertices and dropping them periodically.

If you need to store data for a specific lifetime and then remove it from the graph (vertex time to live), you can store a timestamp property at the creation of the vertex. You can then periodically issue a `drop()` query for all vertices that were created before a certain time; for example:

```
g.V().has(“timestamp”, lt(datetime('2018-10-11')))
```

Using the `datetime( )` Method for Groovy Time Data

Neptune provides the `datetime` method for specifying dates and times for queries sent in the Gremlin **Groovy** variant. This includes the Gremlin Console, text strings using the HTTP REST API, and any other serialization that uses Groovy.

> **Important**
>
> This *only* applies to methods where you send the Gremlin query as a *text string*. If you are using a Gremlin Language Variant, you must use the native date classes and functions for the language. For more information, see the next section, the section called "Native Date and Time".
>
> Starting with TinkerPop `3.5.2` (introduced in Neptune engine release 1.1.1.0), `datetime` is an integral part of TinkerPop.

You can use the `datetime` method to store and compare dates:

```
g.V('3').property('date',datetime('2001-02-08'))
```

```
g.V().has('date',gt(datetime('2000-01-01')))
```

Using Native Date and Time for GLV Time Data

If you are using a Gremlin Language Variant (GLV), you must use the native date and time classes and functions provided by the programming language for Gremlin time data.

The official TinkerPop libraries are all Gremlin Language Variant libraries.

- Go
- Java
- Javascript
- .NET
- Python

> **Important**
>
> This page only applies to Gremlin Language Variant (GLV) libraries. If you are using a method where you send the Gremlin query as text string, you must use Gremlin's datetime() function. This includes the Gremlin Console, text strings using the HTTP REST API or directly submitting Gremlin strings via the drivers.

Go

The following is a partial example in Go that creates a single property named 'date' for the vertex with an ID of '3'. It sets the value to be a date generated using the Go time.Now() function.

```
import ( "time" )

g.V('3').property('date', time.Now()).next();
```

For a complete example for connecting to Neptune using Go, see Using a Go client to connect to a Neptune DB instance.

Java

The following is a partial example in Java that creates a single property named `'date'` for the vertex with an ID of '3'. It sets the value to a date returned by `java.time.OffsetDateTime.now()`.

```
import java.time.OffsetDateTime;

g.V('3').property('date', OffsetDateTime.now()).next();
```

> **Neptune 1.5.0.0: Java GLV datetime change**
>
> Starting with TinkerPop 3.8.0, the Java GLV uses `java.time.OffsetDateTime` as the standard datetime type instead of `java.util.Date`. If you have existing code that passes a `java.util.Date` through the Java GLV, migrate it to `OffsetDateTime` when you upgrade.

For a complete example for connecting to Neptune using Java, see Using a Java client to connect to a Neptune DB instance.

Node.js (JavaScript)

The following is a partial example in JavaScript that creates a single property named '`date`' for the vertex with an ID of '3'. It sets the value to be a date generated using the Node.js `Date()` constructor.

```
g.V('3').property('date', new Date()).next()
```

For a complete example for connecting to Neptune using Node.js, see Using Node.js to connect to a Neptune DB instance.

.NET (C#)

The following is a partial example in C# that creates a single property named '`date`' for the vertex with an ID of '3'. It sets the value to be a date generated using the .NET *`DateTime.UtcNow`* property.

```
Using System;

g.V('3').property('date', DateTime.UtcNow).next()
```

For a complete example for connecting to Neptune using C#, see Using .NET to connect to a Neptune DB instance.

Python

The following is a partial example in Python that creates a single property named '`date`' for the vertex with an ID of '3'. It sets the value to be a date generated using the Python `datetime.now()` method.

```
import datetime

g.V('3').property('date',datetime.datetime.now()).next()
```

For a complete example for connecting to Neptune using Python, see Using Python to connect to a Neptune DB instance.

Best practices using the Gremlin Java client with Neptune

Follow these recommendations when using the Gremlin Java client with Neptune. These best practices help you optimize performance, manage connections effectively, and avoid common pitfalls when working with the Java driver.

For information about configuring heartbeat intervals for Neptune Serverless, see Heartbeat Configuration for Neptune Serverless.

Topics

- Re-use the client object across multiple threads
- Create separate Gremlin Java client objects for read and write endpoints
- Add multiple read replica endpoints to a Gremlin Java connection pool
- Close the client to avoid the connections limit
- Create a new connection after failover
- Set maxInProcessPerConnection and maxSimultaneousUsagePerConnection to the same value
- Send queries to the server as bytecode rather than as strings
- Always completely consume the ResultSet or Iterator returned by a query
- Bulk add vertices and edges in batches
- Disable DNS caching in the Java Virtual Machine
- Optionally, set timeouts at a per-query level
- Troubleshooting java.util.concurrent.TimeoutException

Re-use the client object across multiple threads

Re-use the same client (or `GraphTraversalSource`) object across multiple threads. That is, create a shared instance of a `org.apache.tinkerpop.gremlin.driver.Client` class in your application rather than doing so in every thread. The `Client` object is thread safe, and the overhead of initializing it is considerable.

This also applies to `GraphTraversalSource`, which creates a `Client` object internally. For example, the following code causes a new `Client` object to be instantiated:

```
import static
 org.apache.tinkerpop.gremlin.process.traversal.AnonymousTraversalSource.traversal;
```

```
  /////

GraphTraversalSource traversal = traversal()
                                    .withRemote(DriverRemoteConnection.using(cluster));
```

Create separate Gremlin Java client objects for read and write endpoints

You can increase performance by only performing writes on the writer endpoint and reading from one or more read-only endpoints.

```
Client readerClient = Cluster.build("https://reader-endpoint")
          ...
          .connect()

Client writerClient = Cluster.build("https://writer-endpoint")
          ...
          .connect()
```

Add multiple read replica endpoints to a Gremlin Java connection pool

When creating a Gremlin Java `Cluster` object, you can use the `.addContactPoint()` method to add multiple read replica instances to the connection pool's contact points.

```
Cluster.Builder readerBuilder = Cluster.build()
          .port(8182)
          .minConnectionPoolSize(…)
          .maxConnectionPoolSize(…)
          .........
          .addContactPoint("reader-endpoint-1")
          .addContactPoint("reader-endpoint-2")
```

Close the client to avoid the connections limit

It is important to close the client when you are finished with it to ensure that the WebSocket connections are closed by the server and all resources associated with the connections are released. This happens automatically if you close the cluster using `Cluster.close( )`, because `client.close( )` is then called internally.

If the client is not closed properly, Neptune terminates all idle WebSocket connections after 20 to 25 minutes. However, if you don't explicitly close WebSocket connections when you're done with them and the number of live connections reaches the WebSocket concurrent connection limit, additional connections are then refused with an HTTP 429 error code. At that point, you must restart the Neptune instance to close the connections.

The advice to call `cluster.close()` does not apply to Java AWS Lambda functions. See Managing Gremlin WebSocket connections in AWS Lambda functions for details.

Create a new connection after failover

In case of failover, the Gremlin Driver might continue connecting to the old writer because the cluster DNS name is resolved to an IP address. If this occurs, you can create a new `Client` object after failover.

Set `maxInProcessPerConnection` and `maxSimultaneousUsagePerConnection` to the same value

Both the `maxInProcessPerConnection` and the `maxSimultaneousUsagePerConnection` parameters are related to the maximum number of simultaneous queries you can submit on a single WebSocket connection. Internally, these parameters are co-related and modification of one without the other could lead to a client receiving a timeout while trying to fetch a connection from the client connection pool.

We recommend keeping the default minimum in-process and simultaneous usage values, and setting `maxInProcessPerConnection` and `maxSimultaneousUsagePerConnection` to the same value.

The value to set these parameters at is a function of query complexity and the data model. A use case where the query returns a lot of data would require more connection bandwidth per query and hence, should have lower values for the parameters, and a higher value for `maxConnectionPoolSize`.

By contrast, in a case where the query returns a smaller amount of data, `maxInProcessPerConnection` and `maxSimultaneousUsagePerConnection` should be set to a higher value than `maxConnectionPoolSize`.

Send queries to the server as bytecode rather than as strings

There are advantages to using bytecode rather than a string when submitting queries:

- **Catch invalid query syntax early:** Using the bytecode variant lets you detect invalid query syntax at the compilation stage. If you use the string-based variation, you won't discover the invalid syntax until the query is submitted to the server and an error is returned.
- **Avoid string-based performance penalty:** Any string-based query submission, whether you're using WebSockets or HTTP, results in a detached vertex, which implies that the Vertex object consists of the ID, Label and all the properties associated with the Vertex (see Properties of Elements).

  This can lead to unnecessary computation on the server in cases where the properties are not required. For example, if the customer is interested in getting the vertex with the ID "hakuna#1" using the query, `g.V("hakuna#1")`. If the query is sent as a string based submission, the server would spend time in retrieving the ID, label and all properties for this vertex. If the query is sent as a bytecode submission, the server only spends time retrieving the ID and the label of the vertex.

In other words, rather than submit a query like this:

```
final Cluster cluster = Cluster.build("localhost")
                               .port(8182)
                               .maxInProcessPerConnection(32)
                               .maxSimultaneousUsagePerConnection(32)
                               .serializer(Serializers.GRAPHBINARY_V1D0)
                               .create();

try {
    final Client client = cluster.connect();
    List<Result> results =
client.submit("g.V().has('name','pumba').out('friendOf').id()").all().get();
    System.out.println(verticesWithNamePumba);
} finally {
    cluster.close();
}
```

Instead, submit the query using bytecode, like this:

```
final Cluster cluster = Cluster.build("localhost")
                               .port(8182)
                               .maxInProcessPerConnection(32)
                               .maxSimultaneousUsagePerConnection(32)
                               .serializer(Serializers.GRAPHBINARY_V1D0)
```

```
                                .create();

 try {
     final GraphTraversalSource g =
traversal().withRemote(DriverRemoteConnection.using(cluster));
     List<Object> verticesWithNamePumba = g.V().has("name",
"pumba").out("friendOf").id().toList();
     System.out.println(verticesWithNamePumba);
 } finally {
     cluster.close();
 }
```

Always completely consume the ResultSet or Iterator returned by a query

The client object should always completely consume the `ResultSet` (in the case of string-based submission), or the iterator returned by `GraphTraversal`. If the query results are not completely consumed, the server holds onto them, waiting for the client to finish consuming them.

If your application only needs a partial set of results, you can use a `limit(X)` step with your query to restrict the number of results that the server generates.

Bulk add vertices and edges in batches

Every query to the Neptune DB runs in the scope of a single transaction, unless you use a session. This means that if you need to insert a lot of data using gremlin queries, batching them together in a batch size of 50-100 improves performance by reducing the number of transactions created for the load.

As an example, adding 5 vertices to the database would look like this:

```
// Create a GraphTraversalSource for the remote connection
final GraphTraversalSource g =
 traversal().withRemote(DriverRemoteConnection.using(cluster));
// Add 5 vertices in a single query
g.addV("Person").property(T.id, "P1")
 .addV("Person").property(T.id, "P2")
 .addV("Person").property(T.id, "P3")
 .addV("Person").property(T.id, "P4")
 .addV("Person").property(T.id, "P5").iterate();
```

Similarly, you can batch-add edges using `addE`. Use `V()` to reference existing vertices as the source and target of each edge:

```
// Add edges in a single batched query
g.V("P1").addE("knows").to(V("P2"))
 .V("P2").addE("knows").to(V("P3"))
 .V("P3").addE("knows").to(V("P4"))
 .V("P4").addE("knows").to(V("P5")).iterate();
```

You can also combine vertex and edge creation in a single batch. Use `as()` to label newly created vertices so you can reference them when adding edges in the same traversal:

```
// Add vertices and edges together in a single query
g.addV("Person").property(T.id, "P1").as("p1")
 .addV("Person").property(T.id, "P2").as("p2")
 .addV("Person").property(T.id, "P3").as("p3")
 .addE("knows").from("p1").to("p2")
 .addE("knows").from("p2").to("p3").iterate();
```

Disable DNS caching in the Java Virtual Machine

In an environment where you want to load-balance requests across multiple read replicas, you need to disable DNS caching in the Java Virtual Machine (JVM) and provide Neptune's reader endpoint while creating the Cluster object. Disabling the JVM DNS cache ensures that DNS is resolved again for every new connection so that the requests are distributed across all of the read replicas. You can do this in your application's initialization code with the following line:

```
java.security.Security.setProperty("networkaddress.cache.ttl", "0");
```

However, a more complete and robust solution for load-balancing is provided by the Amazon Gremlin Java client code on GitHub. The Amazon Java Gremlin client is aware of your cluster topology and fairly distributes connections and requests across a set of instances in your Neptune cluster. See this blog post for a sample Java Lambda function that uses that client.

Optionally, set timeouts at a per-query level

Neptune provides you with the ability to set a timeout for your queries using the parameter group option `neptune_query_timeout` (see Parameters). You can also override the global timeout, with code like this:

> **Note**
>
> Query timeout settings apply only to query evaluation. Bytecode-based transaction control operations such as `tx().commit()` and `tx().rollback()` are not subject to query timeouts. For more information, see Timeout behavior for bytecode commit and rollback.

```
final Cluster cluster = Cluster.build("localhost")
                               .port(8182)
                               .maxInProcessPerConnection(32)
                               .maxSimultaneousUsagePerConnection(32)
                               .serializer(Serializers.GRAPHBINARY_V1D0)
                               .create();

try {
    final GraphTraversalSource g =
traversal().withRemote(DriverRemoteConnection.using(cluster));
    List<Object> verticesWithNamePumba = g.with(ARGS_EVAL_TIMEOUT,
500L).V().has("name", "pumba").out("friendOf").id().toList();
    System.out.println(verticesWithNamePumba);
} finally {
    cluster.close();
}
```

Or, for string-based query submission, the code would look like this:

```
RequestOptions options = RequestOptions.build().timeout(500).create();
List<Result> result = client.submit("g.V()", options).all().get();
```

If a query exceeds the per-query timeout, Neptune terminates it. Whether to retry a timed-out query depends on the nature of the failure and your workload. For guidance, see the section called "Exceptions and Retries".

> **Note**
>
> It is possible to incur unexpected costs if you set the query timeout value too high, particularly on a serverless instance. Without a reasonable timeout setting, your query may keeps running much longer than you expected, incurring costs you never anticipated. This is particularly true on a serverless instance that could scale up to a large, expensive instance type while running the query.

You can avoid unexpected expenses of this kind by using a query timeout value that accomodates the run-time your expect and only causes an unusually long run to time out.
Starting from Neptune engine version 1.3.2.0, Neptune supports a new neptune_lab_mode parameter as `StrictTimeoutValidation`. When this parameter has a value of `Enabled`, a per-query timeout value specified as a request option or a query hint cannot exceed the value set globally in the parameter group. In such a case, Neptune will throw `InvalidParameterException`.
This setting can be confirmed in a response on the '/status' endpoint when the value is `Disabled`. In engine version `1.3.2.0`, the default value of this parameter is `Disabled`.
Starting in engine version `1.4.0.0`, the `StrictTimeoutValidation` parameter is `Enabled` by default.
For more information about how timeout precedence is determined when multiple timeout settings are configured, see the neptune_query_timeout parameter documentation.

Troubleshooting `java.util.concurrent.TimeoutException`

The Gremlin Java client throws a `java.util.concurrent.TimeoutException` when a Gremlin request times out at the client itself while waiting for a slot in one of the WebSocket connections to become available. This timeout duration is controlled by the `maxWaitForConnection` client-side configurable parameter.

> **Note**
>
> Because requests that time out at the client are never sent to the server, they aren't reflected in any of the metrics captured at the server, such as `GremlinRequestsPerSec`.

This kind of timeout is generally caused in one of two ways:

- **The server actually reached maximum capacity.** If this is the case, the queue on the server fills up, which you can detect by monitoring the MainRequestQueuePendingRequests CloudWatch metric. The number of parallel queries that the server can handle depends on its instance size.

  If the `MainRequestQueuePendingRequests` metric doesn't show a build-up of pending requests on the server, then the server can handle more requests and the timeout is being caused by client-side throttling.

- **Client throttling of requests.** This can generally be fixed by changing client configuration settings.

  The maximum number of parallel requests that the client can send can be roughly estimated as follows:

  ```
  maxParallelQueries = maxConnectionPoolSize * Max( maxSimultaneousUsagePerConnection,
   maxInProcessPerConnection )
  ```

  Sending more than `maxParallelQueries` to the client causes `java.util.concurrent.TimeoutException` exceptions. You can generally fix this in several ways:

  - *Increase the connection timeout duration.* If latency is not crucial for your application, increase the client's `maxWaitForConnection` setting. The client then waits longer before it times out, which in turn can increase latency.
  - *Increase the maximum requests per connection.* This allows more requests to be sent using the same WebSocket connection. Do this by increasing the client's `maxSimultaneousUsagePerConnection` and `maxInProcessPerConnection` settings. These settings should generally have the same value.
  - *Increase the number of connections in the connection pool.* Do this by increasing the client's `maxConnectionPoolSize` setting. The cost is increased resource consumption, because each connection uses memory and an operating-system file descriptor, and requires an SSL and WebSocket handshake during initialization.

Neptune Best Practices Using openCypher and Bolt

Follow these best practices when using the openCypher query language and Bolt protocol with Neptune. For information about using openCypher in Neptune, see Accessing the Neptune Graph with openCypher.

Topics

- Create a new connection after failover
- Connection handling for long-lived applications
- Connection handling for AWS Lambda
- Prefer directed to bi-directional edges in queries

- Neptune does not support multiple concurrent queries in a transaction
- Close driver objects when you're done
- Use explicit transaction modes for reading and writing
- Retry logic for exceptions
- Set multiple properties at once using a single SET clause
- Use parameterized queries
- Use flattened maps instead of nested maps in UNWIND clause
- Place more restrictive nodes on the left side in Variable-Length Path (VLP) expressions
- Avoid redundant node label checks by using granular relationship names
- Specify edge labels where possible
- Avoid using the WITH clause when possible
- Place restrictive filters as early in the query as possible
- Explicitly check whether properties exist
- Do not use named path (unless it is required)
- Avoid COLLECT(DISTINCT())
- Prefer the properties function over individual property lookup when retrieving all property values
- Perform static computations outside of the query
- Batch inputs using UNWIND instead of individual statements
- Prefer using custom IDs for node/relationship
- Avoid doing ~id computations in the query
- Updating/Merging multiple nodes

Create a new connection after failover

In case of a failover, the Bolt driver can continue connecting to the old writer instance rather than the new active one, because the DNS name resolved to a specific IP address.

To prevent this, close and then reconnect the `Driver` object after any failover.

Connection handling for long-lived applications

When building long-lived applications, such as those running within containers or on Amazon EC2 instances, instantiate a `Driver` object once and then reuse that object for the lifetime of the application. The `Driver` object is thread safe, and the overhead of initializing it is considerable.

Connection handling for AWS Lambda

Bolt drivers are not recommended for use within AWS Lambda functions, because of their connection overhead and management requirements. Use the HTTPS endpoint instead.

Prefer directed to bi-directional edges in queries

When Neptune performs query optimizations, bi-directional edges make it difficult to create optimal query plans. Sub-optimal plans require the engine to do unnecessary work and result in poorer performance.

Therefore, use directed edges rather than bi-directional ones whenever possible. For example, use:

```
MATCH p=(:airport {code: 'ANC'})-[:route]->(d) RETURN p)
```

instead of:

```
MATCH p=(:airport {code: 'ANC'})-[:route]-(d) RETURN p)
```

Most data models don't actually need to traverse edges in both directions, so queries can achieve significant performance improvements by making the switch to using directed edges.

If your data model does require traversing bi-directional edges, make the first node (left-hand side) in the `MATCH` pattern the node with the most restrictive filtering.

Take the example, "Find me all the `routes` to and from the ANC airport". Write this query to start at the ANC airport, like this:

```
MATCH p=(src:airport {code: 'ANC'})-[:route]-(d) RETURN p
```

The engine can perform the minimal amount of work to satisfy the query, because the most restricted node is placed as the first node (left-hand side) in the pattern. The engine can then optimize the query.

This is far preferable than filtering the ANC airport at the end of the pattern, like this:

```
MATCH p=(d)-[:route]-(src:airport {code: 'ANC'}) RETURN p
```

When the most restricted node is not placed first in the pattern, the engine must perform additional work because it can't optimize the query and has to perform additional lookups to arrive at the results.

Neptune does not support multiple concurrent queries in a transaction

Although the Bolt driver itself allows concurrent queries in a transaction, Neptune does not support multiple queries in a transaction running concurrently. Instead, Neptune requires that multiple queries in a transaction be run sequentially, and that the results of each query be completely consumed before the next query is initiated.

The example below shows how to use Bolt to run multiple queries sequentially in a transaction, so that the results of each one are completely consumed before the next one begins:

```
final String query = "MATCH (n) RETURN n";

try (Driver driver = getDriver(HOST_BOLT, getDefaultConfig())) {
  try (Session session = driver.session(readSessionConfig)) {
    try (Transaction trx = session.beginTransaction()) {
      final Result res_1 = trx.run(query);
      Assert.assertEquals(10000, res_1.list().size());
      final Result res_2 = trx.run(query);
      Assert.assertEquals(10000, res_2.list().size());
    }
  }
}
```

Close driver objects when you're done

Be sure to close the client when you are finished with it, so that the Bolt connections are closed by the server and all resources associated with the connections are released. This happens automatically if you close the driver using `driver.close()`.

If the driver is not closed properly, Neptune terminates all idle Bolt connections after 20 minutes, or after 10 days if you are using IAM authentication.

Neptune supports no more than 1000 concurrent Bolt connections. If you don't explicitly close connections when you're done with them, and the number of live connections reaches that limit of 1000, any new connection attempts fail.

Use explicit transaction modes for reading and writing

When using transactions with Neptune and the Bolt driver, it is best to explicitly set the access mode for both read and write transactions to the right settings.

Read-only transactions

For read-only transactions, if you don't pass in the appropriate access mode configuration when building the session, the default isolation level is used, which is mutation query isolation. As a result, it's important for read-only transactions to set the access mode to `read` explicitly.

Auto-commit read transaction example:

```
SessionConfig sessionConfig = SessionConfig
  .builder()
  .withFetchSize(1000)
  .withDefaultAccessMode(AccessMode.READ)
  .build();
Session session = driver.session(sessionConfig);
try {
  (Add your application code here)
} catch (final Exception e) {
  throw e;
} finally {
  driver.close()
}
```

Read transaction example:

```
Driver driver = GraphDatabase.driver(url, auth, config);
SessionConfig sessionConfig = SessionConfig
  .builder()
  .withDefaultAccessMode(AccessMode.READ)
  .build();
driver.session(sessionConfig).readTransaction(
  new TransactionWork<List<String>>() {
    @Override
```

```
    public List<String> execute(org.neo4j.driver.Transaction tx) {
      (Add your application code here)
    }
  }
);
```

In both cases, SNAPSHOT isolation is achieved using Neptune read-only transaction semantics.

Because read replicas only accept read-only queries, any query submitted to a read replica runs under SNAPSHOT isolation semantics.

There are no dirty reads or non-repeatable reads for read-only transactions.

Mutation transactions

For mutation queries, there are three different mechanisms to create a write transaction, each of which is illustrated below:

Implicit write transaction example:

```
Driver driver = GraphDatabase.driver(url, auth, config);
SessionConfig sessionConfig = SessionConfig
  .builder()
  .withDefaultAccessMode(AccessMode.WRITE)
  .build();
driver.session(sessionConfig).writeTransaction(
  new TransactionWork<List<String>>() {
    @Override
    public List<String> execute(org.neo4j.driver.Transaction tx) {
      (Add your application code here)
    }
  }
);
```

Auto-commit write transaction example:

```
SessionConfig sessionConfig = SessionConfig
  .builder()
  .withFetchSize(1000)
  .withDefaultAccessMode(AccessMode.Write)
  .build();
Session session = driver.session(sessionConfig);
```

```
try {
  (Add your application code here)
} catch (final Exception e) {
    throw e;
} finally {
    driver.close()
}
```

Explicit write transaction example:

```
Driver driver = GraphDatabase.driver(url, auth, config);
SessionConfig sessionConfig = SessionConfig
  .builder()
  .withFetchSize(1000)
  .withDefaultAccessMode(AccessMode.WRITE)
  .build();
Transaction beginWriteTransaction = driver.session(sessionConfig).beginTransaction();
  (Add your application code here)
beginWriteTransaction.commit();
driver.close();
```

Isolation levels for write transactions

- Reads made as part of mutation queries are run under `READ COMMITTED` transaction isolation.
- There are no dirty reads for reads made as part of mutation queries.
- Records and ranges of records are locked when reading in a mutation query.
- When a range of the index has been read by a mutation transaction, there is a strong guarantee that this range will not be modified by any concurrent transactions until the end of the read.

Mutation queries are not thread safe.

For conflicts, see Conflict Resolution Using Lock-Wait Timeouts.

Mutation queries are not automatically retried in case of failure.

Retry logic for exceptions

Not all exceptions should be retried. For example, syntax errors, authentication failures, or invalid queries should not trigger a retry. For exceptions that do allow a retry, use an exponential backoff strategy with jitter to progressively increase the wait time between retries and avoid synchronized

retry storms. Set a reasonable maximum number of retries to prevent infinite loops and resource exhaustion. For detailed guidance on developing a practical retry strategy, see the section called "Exceptions and Retries".

The following shows an example of an exponential backoff and retry pattern for handling `ConcurrentModificationException` errors:

```
public static void main() {
  try (Driver driver = getDriver(HOST_BOLT, getDefaultConfig())) {
    retriableOperation(driver, "CREATE (n {prop:'1'})")
        .withRetries(5)
        .withExponentialBackoff(true)
        .maxWaitTimeInMilliSec(500)
        .call();
  }
}

protected RetryableWrapper retriableOperation(final Driver driver, final String query){
  return new RetryableWrapper<Void>() {
    @Override
    public Void submit() {
      log.info("Performing graph Operation in a retry manner......");
      try (Session session = driver.session(writeSessionConfig)) {
        try (Transaction trx =  session.beginTransaction()) {
            trx.run(query).consume();
            trx.commit();
        }
      }
      return null;
    }

    @Override
    public boolean isRetryable(Exception e) {
      if (isCME(e)) {
        log.debug("Retrying on exception.... {}", e);
        return true;
      }
      return false;
    }

    private boolean isCME(Exception ex) {
      return ex.getMessage().contains("Operation failed due to conflicting concurrent
 operations");
```

```
    }
  };
}



/**
 * Wrapper which can retry on certain condition. Client can retry operation using this
 class.
 */
@Log4j2
@Getter
public abstract class RetryableWrapper<T> {

  private long retries = 5;
  private long maxWaitTimeInSec = 1;
  private boolean exponentialBackoff = true;

  /**
   * Override the method with custom implementation, which will be called in retryable
 block.
   */
  public abstract T submit() throws Exception;

  /**
   * Override with custom logic, on which exception to retry with.
   */
  public abstract boolean isRetryable(final Exception e);

  /**
   * Define the number of retries.
   *
   * @param retries -no of retries.
   */
  public RetryableWrapper<T> withRetries(final long retries) {
    this.retries = retries;
    return this;
  }

  /**
   * Max wait time before making the next call.
   *
   * @param time - max polling interval.
   */
```

```
  public RetryableWrapper<T> maxWaitTimeInMilliSec(final long time) {
    this.maxWaitTimeInSec = time;
    return this;
  }

  /**
   * ExponentialBackoff coefficient.
   */
  public RetryableWrapper<T> withExponentialBackoff(final boolean expo) {
    this.exponentialBackoff = expo;
    return this;
  }

  /**
   * Call client method which is wrapped in submit method.
   */
  public T call() throws Exception {
    int count = 0;
    Exception exceptionForMitigationPurpose = null;
    do {
      final long waitTime = exponentialBackoff ? Math.min(getWaitTimeExp(retries),
 maxWaitTimeInSec) : 0;
      try {
          return submit();
      } catch (Exception e) {
        exceptionForMitigationPurpose = e;
        if (isRetryable(e) && count < retries) {
          Thread.sleep(waitTime);
          log.debug("Retrying on exception attempt - {} on exception cause - {}",
 count, e.getMessage());
        } else if (!isRetryable(e)) {
          log.error(e.getMessage());
          throw new RuntimeException(e);
        }
      }
    } while (++count < retries);

    throw new IOException(String.format(
          "Retry was unsuccessful.... attempts %d. Hence throwing exception " + "back
 to the caller...", count),
          exceptionForMitigationPurpose);
  }

  /*
```

```
    * Returns the next wait interval, in milliseconds, using an exponential backoff
    * algorithm.
    */
  private long getWaitTimeExp(final long retryCount) {
    if (0 == retryCount) {
      return 0;
    }
    return ((long) Math.pow(2, retryCount) * 100L);
  }
}
```

Set multiple properties at once using a single SET clause

Instead of using multiple SET clauses to set individual properties, use a map to set multiple properties for an entity at once.

You can use:

```
MATCH (n:SomeLabel {`~id`: 'id1'})
SET n += {property1 : 'value1',
property2 : 'value2',
property3 : 'value3'}
```

Instead of:

```
MATCH (n:SomeLabel {`~id`: 'id1'})
SET n.property1 = 'value1'
SET n.property2 = 'value2'
SET n.property3 = 'value3'
```

The SET clause accepts either a single property or a map. If updating multiple properties on a single entity, using a single SET clause with a map allows the updates to be performed in a single operation instead of multiple operations, which can be executed more effeciently.

Use the SET clause to remove multiple properties at once

When using the openCypher language, REMOVE is used to remove properties from an entity. In Neptune, each property being removed requires a separate operation, adding query latency. You can instead use SET with a map to set all property values to `null`, which in Neptune is equivalent to removing properties. Neptune will have increased performance when multiple properties on a single entity are required to be removed.

Use:

```
WITH {prop1: null, prop2: null, prop3: null} as propertiesToRemove
MATCH (n)
SET n += propertiesToRemove
```

Instead of:

```
MATCH (n)
REMOVE n.prop1, n.prop2, n.prop3
```

Use parameterized queries

It is recommended to always use parameterized queries when querying using openCypher. The query engine can leverage repeated parameterized queries for features like query plan cache, where repeated invocation of the same parameterized structure with different parameters can leverage the cached plans. The query plan generated for parameterized queries is cached and reused only when it completes within 100ms and the parameter types are either NUMBER, BOOLEAN or STRING.

Use:

```
MATCH (n:foo) WHERE id(n) = $id RETURN n
```

With parameters:

```
parameters={"id": "first"}
parameters={"id": "second"}
parameters={"id": "third"}
```

Instead of:

```
MATCH (n:foo) WHERE id(n) = "first" RETURN n
MATCH (n:foo) WHERE id(n) = "second" RETURN n
MATCH (n:foo) WHERE id(n) = "third" RETURN n
```

Use flattened maps instead of nested maps in UNWIND clause

Deep nested structure can restrict the ability of the query engine to generate an optimal query plan. To partially alleviate this issue, the following defined patterns will create optimal plans for the following scenarios:

- Scenario 1: UNWIND with a list of cypher literals, which includes NUMBER, STRING and BOOLEAN.
- Scenario 2: UNWIND with a list of flattened maps, which includes only cypher literals (NUMBER, STRING, BOOLEAN) as values.

When writing a query containing UNWIND clause, use the above recommendation to improve performance.

Scenario 1 example:

```
UNWIND $ids as x
MATCH(t:ticket {`~id`: x})
```

With parameters:

```
parameters={
  "ids": [1, 2, 3]
}
```

An example for Scenario 2 is to generate a list of nodes to CREATE or MERGE. Instead of issuing multiple statements, use the following pattern to define the properties as a set of flattened maps:

```
UNWIND $props as p
CREATE(t:ticket {title: p.title, severity:p.severity})
```

With parameters:

```
parameters={
  "props": [
    {"title": "food poisoning", "severity": "2"},
    {"title": "Simone is in office", "severity": "3"}
  ]
}
```

Instead of nested node objects like:

```
UNWIND $nodes as n
CREATE(t:ticket n.properties)
```

With parameters:

```
parameters={
  "nodes": [
    {"id": "ticket1", "properties": {"title": "food poisoning", "severity": "2"}},
    {"id": "ticket2", "properties": {"title": "Simone is in office", "severity": "3"}}
  ]
}
```

Place more restrictive nodes on the left side in Variable-Length Path (VLP) expressions

In Variable-Length Path (VLP) queries, the query engine optimizes the evaluation by choosing to start the traversal on the left or right side of the expression. The decision is based on the cardinality of the patterns on the left and right side. Cardinality is the number of nodes matching the specified pattern.

- If the right pattern has a cardinality of one, then the right side will be the starting point.
- If the left and the right side have cardinality of one, the expansion is checked on both sides and starts on the side with the smaller expansion. Expansion is the number of outgoing or incoming edges for the node on the left and the node on the right side of the VLP expression. This part of the optimization is only used if the VLP relationship is unidirectional and the relationship type is provided.
- Otherwise, the left side will be the starting point.

For a chain of VLP expressions, this optimization can only be applied to the first expression. The other VLPs are evaluated starting with the left side. As an example, let the cardinality of (a), (b) be one, and the cardinality of (c) be greater than one.

- `(a)-[*1..]->(c)`: Evaluation starts with (a).
- `(c)-[*1..]->(a)`: Evaluation starts with (a).
- `(a)-[*1..]-(c)`: Evaluation starts with (a).

- `(c)-[*1..]-(a)`: Evaluation starts with (a).

Now let the incoming edges of (a) be two, and the outgoing edges of (a) be three, the incoming edges of (b) be four, and the outgoing edges of (b) be five.

- `(a)-[*1..]->(b)`: Evaluation starts with (a) as the outgoing edges of (a) are less than the incoming edges of (b).
- `(a)<-[*1..]-(b)`: Evaluation starts with (a) as the incoming edges of (a) are less than the outgoing edges of (b).

As a general rule, place the more restrictive pattern on the left side of a VLP expression.

Avoid redundant node label checks by using granular relationship names

When optimizing for performance, using relationship labels that are exclusive to node patterns allows the removal of label filtering on nodes. Consider a graph model where the relationship `likes` is only used to define a relationship between two `person` nodes. We could write the following query to find this pattern:

```
MATCH (n:person)-[:likes]->(m:person)
RETURN n, m
```

The `person` label check on n and m is redundant, as we defined the relationship to only appear when both are of the type `person`. To optimize on performance, we can write the query as follows:

```
MATCH (n)-[:likes]->(m)
RETURN n, m
```

This pattern can also apply when properties are exclusive to a single node label. Assume that only `person` nodes have the property `email`, therefore verifying the node label matches `person` is redundant. Writing this query as:

```
MATCH (n:person)
WHERE n.email = 'xxx@gmail.com'
RETURN n
```

Is less efficient than writing this query as:

```
MATCH (n)
WHERE n.email = 'xxx@gmail.com'
RETURN n
```

You should only adopt this pattern when performance is important and you have checks in your modeling process to ensure these edge labels are not reused for patterns involving other node labels. If you later introduce an `email` property on another node label such as `company`, then the results will differ between these two versions of the query.

Specify edge labels where possible

It is recommended to provide an edge label where possible when specifying an edge in a pattern. Consider the following example query, which is used to link all of the people living in a city with all of the people who visited that city.

```
MATCH (person)-->(city {country: "US"})-->(anotherPerson)
RETURN person, anotherPerson
```

If your graph model links people to nodes other than just cities using multiple edge labels, by not specifying the end label, Neptune will need to evaluate additional paths that will later be discarded. In the above query, as an edge label was not given, the engine does more work first and then filters out values to obtain the correct result. A better version of above query might be:

```
MATCH (person)-[:livesIn]->(city {country: "US"})-[:visitedBy]->(anotherPerson)
RETURN person, anotherPerson
```

This not only helps in evaluation, but enables the query planner to create better plans. You could even combine this best practice with redundant node label checks to remove the city label check and write the query as:

```
MATCH (person)-[:livesIn]->({country: "US"})-[:visitedBy]->(anotherPerson)
RETURN person, anotherPerson
```

Avoid using the WITH clause when possible

The WITH clause in openCypher acts as a boundary where everything before it executes, and then the resulting values are passed to the remaining portions of the query. The WITH clause is needed when you require interim aggregation or want to limit the number of results, but aside from that you should try to avoid using the WITH clause. The general guidance is to remove these simple

WITH clauses (without aggregation, order by or limit) to enable the query planner to work on the entire query to create a globally optimal plan. As an example, assume you wrote a query to return all people living in `India`:

```
MATCH (person)-[:lives_in]->(city)
WITH person, city
MATCH (city)-[:part_of]->(country {name: 'India'})
RETURN collect(person) AS result
```

In the above version, the WITH clause restricts the placement of the pattern `(city)-[:part_of]->(country {name: 'India'})` (which is more restrictive) before `(person)-[:lives_in]->(city)`. This makes the plan sub-optimal. An optimization on this query would be to remove the WITH clause and let the planner compute the best plan.

```
MATCH (person)-[:lives_in]->(city)
MATCH (city)-[:part_of]->(country {name: 'India'})
RETURN collect(person) AS result
```

Place restrictive filters as early in the query as possible

In all scenarios, early placement of filters in the query helps in reducing the intermediate solutions a query plan must consider. This means less memory and fewer compute resources are needed to execute the query.

The following example helps you understand these impacts. Suppose you write a query to return all of the people who live in `India`. One version of the query could be:

```
MATCH (n)-[:lives_in]->(city)-[:part_of]->(country)
WITH country, collect(n.firstName + " "  + n.lastName) AS result
WHERE country.name = 'India'
RETURN result
```

The above version of the query is not the most optimal way to achieve this use case. The filter `country.name = 'India'` appears later in the query pattern. It will first collect all persons and where they live, and group them by country, then filter for only the group for `country.name = India`. The optimal way to query for only people living in `India` and then perform the collect aggregation.

```
MATCH (n)-[:lives_in]->(city)-[:part_of]->(country)
```

```
WHERE country.name = 'India'
RETURN collect(n.firstName + " "  + n.lastName) AS result
```

A general rule is to place a filter as soon as possible after the variable is introduced.

Explicitly check whether properties exist

Based on openCypher semantics, when a property is accessed it is equivalent to an optional join and must retain all rows even if the property does not exist. If you know based on your graph schema that a particular property will always exist for that entity, explicitly checking that property for existence allows the query engine to create optimal plans and improve performance.

Consider a graph model where nodes of type `person` always have a property `name`. Instead of doing this:

```
MATCH (n:person)
RETURN n.name
```

Explicitly verify the property existence in the query with an IS NOT NULL check:

```
MATCH (n:person)
WHERE n.name IS NOT NULL
RETURN n.name
```

Do not use named path (unless it is required)

Named path in a query always comes at an additional cost, which can add penalties in terms of higher latency and memory usage. Consider the following query:

```
MATCH p = (n)-[:commentedOn]->(m)
WITH p, m, n, n.score + m.score as total
WHERE total > 100
MATCH (m)-[:commentedON]->(o)
WITH p, m, n, distinct(o) as o1
RETURN p, m.name, n.name, o1.name
```

In the above query, assuming we only want to know the properties of the nodes, the use of path “p” is unnecessary. By specifying the named path as a variable, the aggregation operation using DISTINCT will get expensive both in terms of time and memory usage. A more optimized version of above query could be:

```
MATCH (n)-[:commentedOn]->(m)
WITH m, n, n.score + m.score as total
WHERE total > 100
MATCH (m)-[:commentedON]->(o)
WITH m, n, distinct(o) as o1
RETURN m.name, n.name, o1.name
```

Avoid COLLECT(DISTINCT())

> **Note**
>
> As of engine version 1.4.7.0, this recommended rewrite is no longer needed.

COLLECT(DISTINCT()) is used whenever a list is to be formed containing distinct values. COLLECT is an aggregation function, and grouping is done based on additional keys being projected in the same statement. When distinct is used, the input is split in multiple chunks where each chunk denotes one group for reduction. Performance will be impacted as the number of groups increases. In Neptune, it is much more efficient to perform DISTINCT before actually collecting/forming the list. This allows grouping to be done directly on the grouping keys for the whole chunk.

Consider the following query:

```
MATCH (n:Person)-[:commented_on]->(p:Post)
WITH n, collect(distinct(p.post_id)) as post_list
RETURN n, post_list
```

A more optimal way of writing this query is:

```
MATCH (n:Person)-[:commented_on]->(p:Post)
WITH DISTINCT n, p.post_id as postId
WITH n, collect(postId) as post_list
RETURN n, post_list
```

Prefer the properties function over individual property lookup when retrieving all property values

The `properties()` function is used to return a map containing all properties for an entity, and is much more efficient than returning properties individually.

Assuming your `Person` nodes contain 5 properties, `firstName`, `lastName`, `age`, `dept`, and `company`, the following query would be preferred:

```
MATCH (n:Person)
WHERE n.dept = 'AWS'
RETURN properties(n) as personDetails
```

Rather than using:

```
MATCH (n:Person)
WHERE n.dept = 'AWS'
RETURN n.firstName, n.lastName, n.age, n.dept, n.company

=== OR ===

MATCH (n:Person)
WHERE n.dept = 'AWS'
RETURN {firstName: n.firstName, lastName: n.lastName, age: n.age,
department: n.dept, company: n.company} as personDetails
```

Perform static computations outside of the query

It is recommended to resolve static computations (simple mathematical/string operations) on the client-side. Consider this example where you want to find all people one year older or less than the author:

```
MATCH (m:Message)-[:HAS_CREATOR]->(p:person)
WHERE p.age <= ($age + 1)
RETURN m
```

Here, `$age` is injected into the query via parameters, and is then added to a fixed value. This value is then compared with `p.age`. Instead, a better approach would be doing the addition on the client-side and passing the calculated value as a parameter $ageplusone. This helps the query engine to create optimized plans, and avoids static computation for each incoming row. Following these guidelines, a more efficient verson of the query would be:

```
MATCH (m:Message)-[:HAS_CREATOR]->(p:person)
WHERE p.age <= $ageplusone
RETURN m
```

Batch inputs using UNWIND instead of individual statements

Whenever the same query needs to be executed for different inputs, instead of executing one query per input, it would be much more performant to run a query for a batch of inputs.

If you want to merge on a set of nodes, one option is to run a merge query per input:

```
MERGE (n:Person {`~id`: $id})
SET n.name = $name, n.age = $age, n.employer = $employer
```

With parameters:

```
params = {id: '1', name: 'john', age: 25, employer: 'Amazon'}
```

The above query needs to be executed for every input. While this approach works, it may require many queries to be executed for a large set of input. In this scenario, batching may help reduce the number of queries executed on the server, as well as improve the overall throughput.

Use the following pattern:

```
UNWIND $persons as person
MERGE (n:Person {`~id`: person.id})
SET n += person
```

With parameters:

```
params = {persons: [{id: '1', name: 'john', age: 25, employer: 'Amazon'},
{id: '2', name: 'jack', age: 28, employer: 'Amazon'},
{id: '3', name: 'alice', age: 24, employer: 'Amazon'}...]}
```

Experimentation with different batch sizes is recommended to determine what works best for your workload.

Prefer using custom IDs for node/relationship

Neptune allows users to explicitly assign IDs on nodes and relationships. The ID must be globally unique in the dataset and deterministic to be useful. A deterministic ID can be used as a lookup or a filtering mechanism just like properties; however, using an ID is much more optimized from query execution perspective than using properties. There are several benefits to using custom IDs -

- Properties can be null for an existing entity, but the ID must exist. This allows the query engine to use an optimized join during execution.
- When concurrent mutation queries are executed, the chances of concurrent modification exceptions (CMEs) are reduced significantly when IDs are used to access nodes because fewer locks are taking on IDs than properties due to their enforced uniqueness.
- Using IDs avoids the chance of creating duplicate data as Neptune enforces uniqueness on IDs, unlike properties.

The following query example uses a custom ID:

> **Note**
>
> The property `~id` is used to specify the ID, whereas `id` is just stored as any other property.

```
CREATE (n:Person {`~id`: '1', name: 'alice'})
```

Without using a custom ID:

```
CREATE (n:Person {id: '1', name: 'alice'})
```

If using the latter mechanism, there is no uniqueness enforcement and you could later execute the query:

```
CREATE (n:Person {id: '1', name: 'john'})
```

This creates a second node with `id=1` named `john`. In this scenario, you would now have two nodes with `id=1`, each having a different name - (alice and john).

Avoid doing ~id computations in the query

When using custom IDs in the queries, always perform static computations outside the queries and provide these values in the parameters. When static values are provided, the engine is better able to optimize lookups and avoid scanning and filtering these values.

If you want to create edges between nodes that are existing in the database, one option could be:

```
UNWIND $sections as section
MATCH (s:Section {`~id`: 'Sec-' + section.id})
MERGE (s)-[:IS_PART_OF]->(g:Group {`~id`: 'g1'})
```

With parameters:

```
parameters={sections: [{id: '1'}, {id: '2'}]}
```

In the query above, the `id` of the section is being computed in the query. Since the computation is dynamic, the engine cannot statically inline ids and ends up scanning all section nodes. The engine then performs post-filtering for required nodes. This can be costly if there are many section nodes in the database.

A better way to achieve this is to have `Sec-` prepended in the ids being passed into the database:

```
UNWIND $sections as section
MATCH (s:Section {`~id`: section.id})
MERGE (s)-[:IS_PART_OF]->(g:Group {`~id`: 'g1'})
```

With parameters:

```
parameters={sections: [{id: 'Sec-1'}, {id: 'Sec-2'}]}
```

Updating/Merging multiple nodes

When running `MERGE` or `CREATE` queries on multiple nodes it is recommended to use an `UNWIND` in combination with a single MERGE/CREATE clause versus using a MERGE/CREATE clause for each node. Queries that use one clause for one node lead to an inefficient execution plan due to each line requiring optimization. This leads to the majority of the execution time of the query being spent in the static processing instead of the actual update.

One Clause per node is not optimal as it doesn't scale with increasing number of nodes:

```
MERGE (p1:Person {name: 'NameA'})
ON CREATE SET p1 += {prop1: 'prop1V1', prop2: 'prop2V1'}
MERGE (p2:Person {name: 'NameB'})
ON CREATE SET p2 += {prop1: 'prop1V2', prop2: 'prop2V2'}
MERGE (p3:Person {name: 'NameC'})
ON CREATE SET p3 += {prop1: 'prop1V3', prop2: 'prop1V3'}
```

Using an UNWIND in conjunction with one MERGE/CREATE clause allows for the same behavior but a more optimal execution plan. With this in mind, the changed query would look like:

```
## If not using custom id for nodes/relationship
UNWIND [{name: 'NameA', prop1: 'prop1V1', prop2: 'prop2V1'}, {name: 'NameB', prop1:
 'prop1V2', prop2: 'prop2V2'}, {name: 'NameC', prop1: 'prop1V3', prop2: 'prop1V3'}] AS
 props
MERGE (p:Person {name: props.name})
ON CREATE SET p = props

## If using custom id for nodes/relationship
UNWIND [{`~id`: '1', 'name': 'NameA', 'prop1: 'prop1V1', prop2: 'prop2V1'}, {`~id`:
 '2', name: 'NameB', prop1: 'prop1V2', prop2: 'prop2V2'}, {`~id`: '3', name: 'NameC',
 prop1: 'prop1V3', prop2: 'prop1V3'}] AS props
MERGE (p:Person {`~id`: props.id})
ON CREATE SET p = removeKeyFromMap(props, '~id')
```

Neptune Best Practices Using SPARQL

Follow these best practices when using the SPARQL query language with Neptune. For information about using SPARQL in Neptune, see Accessing the Neptune graph with SPARQL.

Topics

- Querying All Named Graphs by Default
- Specifying a Named Graph for Load
- Choosing Between FILTER, FILTER...IN, and VALUES in Your Queries

Querying All Named Graphs by Default

Amazon Neptune associates every triple with a named graph. The default graph is defined as the union of all named graphs.

If you submit a SPARQL query without explicitly specifying a graph via the GRAPH keyword or constructs such as FROM NAMED, Neptune always considers all triples in your DB instance. For example, the following query returns all triples from a Neptune SPARQL endpoint:

```
SELECT * WHERE { ?s ?p ?o }
```

Triples that appear in more than one graph are returned only once.

For information about the default graph specification, see the RDF Dataset section of the SPARQL 1.1 Query Language specification.

Specifying a Named Graph for Load

Amazon Neptune associates every triple with a named graph. If you don't specify a named graph when loading, inserting, or updating triples, Neptune uses the fallback named graph defined by the URI, `http://aws.amazon.com/neptune/vocab/v01/DefaultNamedGraph`.

If you are using the Neptune bulk loader, you can specify the named graph to use for all triples (or quads with the fourth position blank) by using the `parserConfiguration: namedGraphUri` parameter. For information about the Neptune loader `Load` command syntax, see the section called "Loader Command".

Choosing Between FILTER, FILTER...IN, and VALUES in Your Queries

There are three basic ways to inject values in SPARQL queries: `FILTER`, `FILTER...IN`, and `VALUES`.

For example, suppose that you want to look up the friends of multiple people within a single query. Using `FILTER`, you might structure your query as follows:

```
PREFIX ex: <https://www.example.com/>
PREFIX foaf : <http://xmlns.com/foaf/0.1/>

SELECT ?s ?o
WHERE {?s foaf:knows ?o. FILTER (?s = ex:person1 || ?s = ex:person2)}
```

This returns all the triples in the graph that have `?s` bound to `ex:person1` or `ex:person2` and have an outgoing edge labeled `foaf:knows`.

You can also create a query using `FILTER...IN` that returns equivalent results:

```
PREFIX ex: <https://www.example.com/>
PREFIX foaf : <http://xmlns.com/foaf/0.1/>

SELECT ?s ?o
```

```
WHERE {?s foaf:knows ?o. FILTER (?s IN (ex:person1, ex:person2))}
```

You can also create a query using `VALUES` that in this case also returns equivalent results:

```
PREFIX ex: <https://www.example.com/>
PREFIX foaf : <http://xmlns.com/foaf/0.1/>

SELECT ?s ?o
WHERE {?s foaf:knows ?o. VALUES ?s {ex:person1 ex:person2}}
```

Although in many cases these queries are semantically equivalent, there are some cases where the two `FILTER` variants differ from the `VALUES` variant:

- The first case is when you inject duplicate values, such as injecting the same person twice. In that case, the `VALUES` query includes the duplicates in your result. You can explicitly eliminate such duplicates by adding a `DISTINCT` to the `SELECT` clause. But there might be situations where you actually want duplicates in the query results for redundant value injection.

  However, the `FILTER` and `FILTER...IN` versions extract the value only once when the same value appears multiple times.
- The second case is related to the fact that `VALUES` always performs an exact match, whereas `FILTER` might apply type promotion and do fuzzy matching in some cases.

  For instance, when you include a literal such as `"2.0"^^xsd:float` in your values clause, a `VALUES` query exactly matches this literal, including literal value and data type.

  By contrast, `FILTER` produces a fuzzy match for these numeric literals. The matches could include literals with the same value but different numeric data types, such as `xsd:double`.

  > **Note**
  >
  > There is no difference between the `FILTER` and `VALUES` behavior when enumerating string literals or URIs.

The differences between `FILTER` and `VALUES` can affect optimization and the resulting query evaluation strategy. Unless your use case requires fuzzy matching, we recommend using `VALUES` because it avoids looking at special cases related to type casting. As a result, `VALUES` often produces a more efficient query that runs faster and is less expensive.

Best practices for using Neptune with Amazon ECS and Amazon EKS

Running your graph application in containers on Amazon Elastic Container Service (Amazon ECS) or Amazon Elastic Kubernetes Service (Amazon EKS) introduces considerations that don't apply to traditional long-running server deployments. Containers are ephemeral — they start, stop, and replace each other during rolling updates and scaling events. This lifecycle affects how your application manages connections to Neptune, handles host replacements, and authenticates with IAM.

The following sections describe practical considerations for running Neptune workloads in containerized environments.

Topics

- Manage connection pools and lifecycle in containerized environments
- Handle host replacement and connection stalling
- Configure networking, security groups, and IAM authentication

Manage connection pools and lifecycle in containerized environments

Managing connections to Neptune effectively is essential in containerized environments where containers (pods in Kubernetes) start, stop, and scale frequently. Neptune enforces several WebSocket connection limits:

- Each instance type has a maximum number of concurrent WebSocket connections
- Idle connections are closed after 20–25 minutes of inactivity
- With IAM authentication, connections are force-closed after 10 days regardless of activity (see WebSocket connection limits)

Some drivers support periodic keep-alive heartbeats to prevent idle disconnects. For example, the Gremlin Java driver's `keepAliveInterval` sends a periodic ping to the server. Consult your specific driver's configuration options to determine what keep-alive settings are available and how they interact with Neptune's idle timeout.

Understanding these limits helps you design connection handling that works well with container orchestration patterns like rolling deployments, autoscaling, and container recycling. For general

guidance on connection management with the Java Gremlin driver, see Close the client to avoid the connections limit and Create a new connection after failover.

Close connections on container shutdown

Implement a graceful shutdown by adding a `SIGTERM` handler that closes your graph client before the container exits. For example, call Gremlin Java's `cluster.close()`, Gremlin's `driverRemoteConnection.close()` in Go or Python, or Bolt's `driver.close()` for the Neo4j driver. In Kubernetes, use a `preStop` hook to drain connections before `SIGTERM` is sent, especially during rolling updates.

> **Note**
>
> Docker and Amazon ECS only deliver `SIGTERM` to PID 1 in the container. Use the exec form in your Dockerfile (for example, `ENTRYPOINT ["java", "-jar", "app.jar"]` not `ENTRYPOINT java -jar app.jar`) or an init process like `tini` to ensure your application receives the signal.

Use a single client instance per container

Creating multiple Gremlin `DriverRemoteConnection` or Bolt `neo4j.Driver` instances in the same container multiplies connection usage. Share a single client instance across all threads in the container. The client objects are thread-safe.

Size connection pools relative to your fleet

Set `maxConnectionPoolSize` so that the total connections across all containers don't exceed the Neptune instance connection limit. For example, if you run 20 containers with 8 threads each, calculate the per-container pool size so that 20 × pool size stays within the instance limit for your instance type. For more information about configuring connection pool sizes, see Set `maxInProcessPerConnection` and `maxSimultaneousUsagePerConnection` to the same value.

Set a connection wait timeout

In the Java Gremlin driver, configure `maxWaitForConnection` to a reasonable value such as 5–10 seconds. The default is often infinite or very long, which causes threads to hang when the pool is exhausted rather than failing fast with a clear error. Not all language drivers support this setting—check your driver's documentation for the equivalent configuration option.

Handle host replacement and connection stalling

When Neptune replaces a host (for example, during maintenance or failover), existing connections to that host become invalid. In containerized environments, this can stall all threads in a container if the client doesn't handle the replacement gracefully.

Use current client versions

If you use the Gremlin query language, use a TinkerPop driver version that is compatible with your Neptune engine version (see Accessing a Neptune graph with Gremlin for the compatibility table). If you use the Java driver, consider `neptune-gremlin-client` — a wrapper around the TinkerPop Java driver that adds connection management features like endpoint health checking and failover handling. It follows the same version compatibility rules as the underlying TinkerPop driver.

Use `neptune-gremlin-client` version 3.x (or at minimum version 2.0.7), depending on what your Neptune version allows. These newer versions improve resiliency and connection handling.

For openCypher users with the Neo4j driver, close and recreate the `Driver` object when you detect a connection failure during failover. Neptune supports Bolt protocol versions 1 through 4.0. For more information, see Neptune Best Practices Using openCypher and Bolt.

Use cluster or reader endpoints

Don't connect to instance endpoints directly. Use the cluster endpoint for writes and the reader endpoint for reads. If you must use instance endpoints with `neptune-gremlin-client`, enable endpoint health-check filtering through the `/status` API.

Configure liveness probes with tolerance

Set your Kubernetes liveness probe `failureThreshold` to at least 30 with a 10-second period (300 seconds total). This prevents Kubernetes from restarting pods during the approximately 5-minute window when Neptune is completing a host replacement.

Implement retry with backoff

A single failed request during host replacement shouldn't crash the container. Implement retry logic with exponential backoff on connection failures so that transient errors during replacement resolve without intervention. For guidance on retryable exceptions, see Neptune transaction exceptions.

Configure networking, security groups, and IAM authentication

Containerized deployments on Amazon ECS or Amazon EKS require correct network configuration and IAM credential management to connect to Neptune.

Security groups checklist

- The Neptune cluster security group must allow inbound TCP on port 8182 (or the custom port configured for your Neptune cluster) from the Amazon EKS node or pod CIDR range.
- If you use the Amazon VPC CNI plugin with custom networking, verify that the pod subnet has a route to the Neptune subnet.
- Neptune must be in the same Amazon VPC as your Amazon ECS or Amazon EKS cluster, or connected through Amazon VPC peering or Transit Gateway.

Verify connectivity from inside the container

Before debugging application-level issues, confirm the network path by running one of the following commands from inside the container:

AWS CLI

```
aws neptunedata get-engine-status \
  --endpoint-url https://your-neptune-endpoint:8182
```

For more information, see get-engine-status in the AWS CLI Command Reference.

SDK

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    region_name='us-east-1',
    endpoint_url='https://your-neptune-endpoint:8182',
    config=Config(read_timeout=10)
)

response = client.get_engine_status()
print(response['status'])
```

awscurl

```
awscurl https://your-neptune-endpoint:8182/status \
  --region us-east-1 \
  --service neptune-db
```

> **Note**
>
> This example assumes that your AWS credentials are configured in your environment. Replace `us-east-1` with the Region of your Neptune cluster.

curl

```
curl -k https://your-neptune-endpoint:8182/status
```

> **Note**
>
> The `-k` flag skips certificate verification. Use this only for connectivity testing, not in production code.

A successful response confirms that security groups, routing, and DNS resolution are all working. A timeout indicates a network-level issue.

IAM credential refresh

If you use IAM authentication with Neptune, your pods must refresh credentials before they expire. We recommend Amazon EKS Pod Identity for new deployments. IAM Roles for Service Accounts (IRSA) is also supported and widely used.

With either approach, retrieve credentials *inside* the authentication generator function, not once at startup. IRSA and Pod Identity tokens expire after 1 hour. If you cache credentials at initialization, requests will fail with `AccessDeniedException: security token expired` after the token expires.

- For gremlin-go: call `cfg.Credentials.Retrieve(ctx)` inside the gen function passed to `NewDynamicAuth`.

- For Java: use `NeptuneGremlinClient` with a `HandshakeInterceptor` that refreshes SigV4 signing on each new connection.

Cross-account access

Neptune doesn't support direct cross-account IAM authentication. If your Amazon EKS cluster is in a different AWS account than your Neptune cluster, the pod must first assume a role in the Neptune account before authenticating.

Ingress configuration for Graph Explorer

If you run Graph Explorer in a containerized environment, ensure that all paths are accessible to the proxy—not only `/explorer`. On Amazon EKS, configure your ingress controller rules to allow all paths. On Amazon ECS, configure your Application Load Balancer target group and listener rules to forward all paths to the Graph Explorer container. Graph Explorer's proxy makes calls to `/summary`, `/gremlin`, `/openCypher`, and `/sparql`. Path-restricted routing rules break the proxy silently.

Best practices for using the AWS CLI and SDKs with Neptune

By default, the AWS CLI and AWS SDKs time out after 60 seconds and automatically retry failed requests. For queries that take longer than 60 seconds to complete, this default behavior causes duplicate query execution. The retry fires a new request while the original query is still running on the server.

> **Important**
>
> When you run long-running queries through the Neptune dataplane API (for example, `execute-open-cypher-query`), disable automatic retries. Also increase or remove the read timeout. Otherwise, the client retries the query while the original is still executing, which doubles the load on your Neptune cluster.

The following recommendations apply to any Neptune dataplane operation that can run longer than the default 60-second timeout:

- Set the read timeout to 0 (infinite) or a value larger than your longest expected query duration.
- Set the total maximum attempts to 1 (one attempt, zero retries). This prevents the client from retrying a request that the server is still processing.

The following examples show how to configure these settings for the AWS CLI and popular SDKs.

AWS CLI

```
export AWS_MAX_ATTEMPTS=1
aws neptunedata execute-open-cypher-query \
  --endpoint-url https://your-cluster-endpoint:port \
  --open-cypher-query "RETURN 1" \
  --cli-read-timeout 0
```

The `--cli-read-timeout 0` parameter disables the read timeout, allowing the AWS CLI to wait indefinitely for a response. The `AWS_MAX_ATTEMPTS=1` environment variable sets the total maximum attempts to 1, meaning the request is tried exactly once with no retries.

Python (boto3)

```
import boto3
from botocore.config import Config

client = boto3.client(
    'neptunedata',
    endpoint_url='https://your-cluster-endpoint:port',
    region_name='us-east-1',
    config=Config(
        retries={'total_max_attempts': 1, 'mode': 'standard'},
        read_timeout=None
    )
)
```

Setting `read_timeout=None` disables the read timeout. Setting `total_max_attempts` to 1 means the request is tried exactly once with no retries.

Java V2 SDK

```
import software.amazon.awssdk.services.neptunedata.NeptunedataClient;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.core.retry.RetryPolicy;
import software.amazon.awssdk.regions.Region;
import java.time.Duration;
import java.net.URI;

// Build the Neptune dataplane client with no retries and infinite timeout
NeptunedataClient neptunedataClient = NeptunedataClient.builder()
```

```
    .endpointOverride(URI.create("https://your-cluster-endpoint:port"))
    .region(Region.of("us-east-1"))
    .overrideConfiguration(ClientOverrideConfiguration.builder()
        .apiCallTimeout(Duration.ZERO)
        .retryPolicy(RetryPolicy.none())
        .build())
    .build();
```

Setting `apiCallTimeout` to `Duration.ZERO` configures an infinite timeout for the API call. Using `RetryPolicy.none()` disables all automatic retries.

Amazon Neptune Limits

Regions

Amazon Neptune is available in the following AWS Regions:

- US East (N. Virginia): `us-east-1`
- US East (Ohio): `us-east-2`
- US West (N. California): `us-west-1`
- US West (Oregon): `us-west-2`
- Canada (Central): `ca-central-1`
- Canada West (Calgary): `ca-west-1`
- South America (São Paulo): `sa-east-1`
- Europe (Stockholm): `eu-north-1`
- Europe (Spain): `eu-south-2`
- Europe (Ireland): `eu-west-1`
- Europe (London): `eu-west-2`
- Europe (Paris): `eu-west-3`
- Europe (Frankfurt): `eu-central-1`
- Middle East (Bahrain): `me-south-1`
- Middle East (UAE): `me-central-1`
- Israel (Tel Aviv): `il-central-1`
- Africa (Cape Town): `af-south-1`
- Asia Pacific (Hong Kong): `ap-east-1`
- Asia Pacific (Tokyo): `ap-northeast-1`
- Asia Pacific (Seoul): `ap-northeast-2`
- Asia Pacific (Osaka): `ap-northeast-3`
- Asia Pacific (Singapore): `ap-southeast-1`
- Asia Pacific (Sydney): `ap-southeast-2`
- Asia Pacific (Jakarta): `ap-southeast-3`
- Asia Pacific (Melbourne): `ap-southeast-4`

- Asia Pacific (Malaysia): `ap-southeast-5`
- Asia Pacific (Mumbai): `ap-south-1`
- Asia Pacific (Hyderabad): `ap-south-2`
- China (Beijing): `cn-north-1`
- China (Ningxia): `cn-northwest-1`
- AWS GovCloud (US-West): `us-gov-west-1`
- AWS GovCloud (US-East): `us-gov-east-1`

Differences in China regions

As is true of many AWS services, Amazon Neptune operates slightly differently in China (Beijing) and China (Ningxia) than in other AWS regions.

For example, when Neptune ML uses Amazon API Gateway to create its export service, IAM authentication is enabled by default. In China regions, the process for changing that option is slightly different than it is in other regions.

These and other differences are explained here.

Maximum size of storage cluster volumes

A Neptune cluster volume can grow to a maximum size of 128 tebibytes (TiB) in all supported regions. See Amazon Neptune storage, reliability and availability.

DB instance sizes supported

Neptune supports different DB instance classes in different AWS Regions. To find out what classes are supported in a given Region, see Amazon Neptune Pricing and choose the Region that you are interested in.

Limits for each AWS account

For certain management features, Amazon Neptune uses operational technology that is shared with Amazon Relational Database Service (Amazon RDS).

Each AWS account has limits for each Region on the number of Amazon Neptune and Amazon RDS resources that you can create. These resources include DB instances and DB clusters.

After you reach a limit for a resource, additional calls to create that resource fail with an exception.

For a list of limits shared between Amazon Neptune and Amazon RDS, see Limits in Amazon RDS in the *Amazon RDS User Guide*.

Connection to Neptune requires a VPC

Amazon Neptune is a virtual private cloud (VPC)–only service.

Additionally, instances do not allow access from outside the VPC.

Neptune requires SSL

Beginning with engine version `1.0.4.0`, Amazon Neptune only allows Secure Sockets Layer (SSL) connections through HTTPS to any instance or cluster endpoint.

Neptune requires TLS version 1.2, using the following strong cipher suites:

- `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`
- `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`
- `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384`
- `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256`
- `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA`
- `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA`

Availability zones and DB subnet groups

Amazon Neptune requires a DB subnet group for each cluster that has subnets in at least two supported Availability Zones (AZs).

We recommend using three or more subnets in different Availability Zones.

HTTP request payload maximum (150 MB)

The total size of Gremlin and SPARQL HTTP requests must be less than 150 MB. If a request exceeds this size, Neptune returns `HTTP 400: BadRequestException`.

This limit does not apply to Gremlin WebSockets connections.

Gremlin implementation differences

The Amazon Neptune Gremlin implementation has specific implementation details that might differ from other Gremlin implementations.

For more information, see Gremlin standards compliance in Amazon Neptune.

Neptune does not support null characters in string data

Neptune does not support null characters in strings. This is true in property-graph data for Gremlin and openCypher, and for RDF/SPARQL data.

SPARQL UPDATE LOAD from URI

`SPARQL UPDATE LOAD` from URI works only with resources that are within the same VPC.

This includes Amazon S3 URLs in the same Region as the cluster with an Amazon S3 VPC endpoint created.

The Amazon S3 URL must be HTTPS, and any authentication must be included in the URL. For more information, see Authenticating Requests: Using Query Parameters in the *Amazon Simple Storage Service API Reference*.

For information about creating a VPC endpoint, see Creating an Amazon S3 VPC Endpoint.

If you need to load data from a file, we recommend that you use the Amazon Neptune loader API. For more information, see Using the Amazon Neptune bulk loader to ingest data.

> **Note**
>
> The Amazon Neptune loader API is non-ACID.

IAM authentication and access control

In Neptune engine versions prior to release 1.2.0.0, IAM authentication and access control is only supported at the DB cluster level. From release `1.2.0.0` forward, however, you can control query-based access at a more granular level using condition keys in IAM policies. For more information, see Using query actions in Neptune data-access policy statements and Authenticating your Amazon Neptune database with AWS Identity and Access Management

The Amazon Neptune console requires **NeptuneReadOnlyAccess** permissions. You can restrict access to IAM users by revoking this access. For more information, see Using AWS managed policies to access Amazon Neptune databases

Amazon Neptune does not support user name/password–based access control.

WebSocket concurrent connections and maximum connection time

There is a limit to the number of concurrent WebSocket connections per Neptune DB instance. When that limit is reached, Neptune throttles any request to open a new WebSocket connection in order to prevent using up all of the allocated heap memory.

For all larger instance types supported by Neptune and all serverless instances, the maximum number concurrent of WebSocket connections is 32K (32,768).

The maximum concurrent WebSocket connections for smaller instance types are listed in the table below:

Instance Type	Maximum concurrent WebSocket connections
serverless	32768
db.r4.4xlarge	16384
db.r4.2xlarge	8192
db.r4.xlarge	4096
db.r4.large	2048
db.r5.4xlarge	16384
db.r5.2xlarge	8192
db.r5.xlarge	4096
db.r5.large	2048

Instance Type	Maximum concurrent WebSocket connections
db.r5d.8xlarge	32768
db.r5d.4xlarge	16384
db.r5d.2xlarge	8192
db.r5d.xlarge	4096
db.r5d.large	2048
db.t3.medium	512
db.t4g.medium	512
db.r6g.4xlarge	16384
db.r6g.2xlarge	8192
db.r6g.xlarge	4096
db.r6g.large	2048
db.r6gd.4xlarge	16384
db.r6gd.2xlarge	8192
db.r6gd.xlarge	4096
db.r6gd.large	2048
db.x2iezn.2xlarge	16384
db.x2iedn.xlarge	4096
db.x2gd.xlarge	4096
db.x2gd.large	2048
db.x2gd.4xlarge	16384

Instance Type	Maximum concurrent WebSocket connections
db.x2gd.2xlarge	8192
db.x2g.xlarge	4096
db.x2g.large	2048
db.x2g.4xlarge	16384
db.x2g.2xlarge	8192
db.x1e.xlarge	4096
db.x1e.2xlarge	16384
db.r6i.xlarge	4096
db.r6i.large	2048
db.r6i.2xlarge	8192
db.r7g.4xlarge	16384
db.r7g.2xlarge	8192
db.r7g.xlarge	4096
db.r7g.large	2048
db.r8g.4xlarge	16384
db.r8g.2xlarge	8192
db.r8g.xlarge	4096
db.r8g.large	2048
db.r7i.large	2048
db.r7i.xlarge	4096

Instance Type	Maximum concurrent WebSocket connections
db.r7i.2xlarge	8192

> **Note**
>
> Neptune no longer supports R4 instance types.

When a client properly closes a connection, the closure is immediately reflected in the open connections count.

If the client doesn't close a connection, the connection may be closed automatically after a 20- to 25-minute idle timeout (the idle timeout is the time elapsed since the last message was received from the client). However, as long as the idle timeout is not reached, Neptune keeps the connection open indefinitely.

When IAM authentication is enabled, a WebSocket connection is always disconnected a few minutes more than 10 days after it was established, if it hasn't already been closed by then.

Limits on properties and labels

There is no limit on the number of vertices and edges, or RDF quads you can have in a graph.

There is also no limit on the number of properties or labels that any one vertex or edge can have.

There is a size limit of 55 MB on the size of an individual property or label. In RDF terms, this means that the value in any column (S, P, O or G) of an RDF quad cannot exceed 55 MB.

If you need to associate a larger object such as an image with a vertex or node in your graph, you can store it as a file in Amazon S3 and use the Amazon S3 path as the property or label.

Limits that affect the Neptune bulk loader

You cannot queue up more than 64 Neptune bulk load jobs at a time.

Neptune only keeps track of the most recent 1,024 bulk load jobs.

Neptune only stores the last 10,000 error details per job.

Working with other AWS services

You can use Amazon Neptune in conjunction with many other AWS services:

Neptune integrations with other services

- **AWS Glue** – AWS Glue is a serverless data integration service that helps you perform extract, transform, and load (ETL) jobs on data.

  Neptune provides an open-source library, neptune-python-utilities, that simplifies using Python and Gremlin within a Glue job. The Neo4j Spark Connector is also supported for running Scala and openCypher Glue jobs.

- **Amazon SageMaker AI** – Amazon SageMaker AI is a full-featured machine learning platform for building, training, and deploying high-quality machine learning models.

  Neptune integrates with SageMaker AI in two primary ways:

  - Neptune provides an open-source Python package for Jupyter notebooks which can be found in the Neptune graph notebook project on GitHub. This package contains a set of Jupyter magics, tutorial notebooks, and code samples that provide in an interactive coding environment where you can learn about graph technology and Neptune. Neptune provides a fully managed environment for Jupyter notebooks hosted by SageMaker AI, and automatically links to the notebooks in the open-source Neptune graph notebook project.
  - The Neptune ML feature makes it possible to build and train useful machine learning models on large graphs in hours instead of weeks. To accomplish this, Neptune ML uses graph neural network (GNN) technology powered by Amazon SageMaker AI and the Deep Graph Library (DGL).

- **AWS Lambda** – AWS Lambda functions have many uses in Neptune applications.

  For information about how to use Lambda functions with any of the popular Gremlin drivers and language variants, as well as specific examples of Lambda functions written in Java, JavaScript, and Python, see Using AWS Lambda functions in Amazon Neptune.

- **Amazon Athena** – Amazon Athena is an interactive query service that makes it easy to analyze data in Amazon Simple Storage Service and other federated data sources using standard SQL.

  Neptune provides a connector to Athena that enables Athena to communicate with your data stored in Neptune.

- **AWS Database Migration Service (AWS DMS)** – AWS Database Migration Service is an AWS web service you can use to migrate data from one database to another.

  AWS DMS can load data into Neptune from supported source databases quickly and securely. The source database remains fully operational during the migration, minimizing downtime for applications that rely on it.
- **AWS Backup** – AWS Backup is a fully managed backup service that makes it easy to centralize and automate the backup of data across AWS services in the cloud as well as on premises.

  AWS Backup lets you to create automated periodic snapshots of Neptune clusters using your centralized data protection policy across the supported AWS services for database, storage, and compute.
- **AWS SDK for pandas** – The AWS SDK for pandas (previously known as AWS Data Wrangler, or `awswrangler`), is an AWS Professional Service open-source python initiative that extends the power of the `pandas` Python data analysis library to AWS, connecting `DataFrames` and more than 30 AWS data-related services, including Neptune.

  In addition to the SDK, there is also a tutorial about how to use it with Neptune, and several sample Neptune notebooks, namely Fraud Ring Detection, Synthetic Identity Detection, and Logistics Analysis.
- **JDBC Driver** – The Neptune JDBC driver supports openCypher, Gremlin, SQL-Gremlin, and SPARQL queries.

  JDBC connectivity makes it easy to connect to Neptune with business intelligence (BI) tools such as Tableau.

Neptune tools and utilities

Amazon Neptune provides a number of tools and utilities that can simplify and automate your work with a graph. Among these are the following:

Amazon Neptune tools

- **Amazon Neptune utility for GraphQL** – The Amazon Neptune utility for GraphQL is an open-source Node.js command-line tool that can help you create and maintain a GraphQL API for a Neptune property-graph database. It is a no-code way to create a GraphQL resolver for GraphQL queries that have a variable number of input parameters and return a variable number of nested fields.
- **Nodestream** – Nodestream is a framework for dealing with semantically modeling data as a graph. It is designed to be flexible and extensible, allowing you to define how data is collected and modeled as a graph. It uses a pipeline-based approach to define how data is collected and processed, and it provides a way to define how the graph should be updated when the schema changes.
- **Amazon Neptune MCP Query Servers** – Model Context Protocol (MCP) server for Amazon Neptune that supports both Neptune Database and Neptune Analytics by providing the ability to run graph queries (openCypher and Gremlin) as well as fetch the graph schema.
- **Amazon Neptune MCP Memory Servers** – Model Context Protocol (MCP) server for Amazon Neptune that provides memory to agents, stored in a knowledge graph against Amazon Neptune.

Amazon Neptune utility for GraphQL

The Amazon Neptune utility for GraphQL is an open-source Node.js command-line tool that can help you create and maintain a GraphQL API for a Neptune property-graph database (it does not yet work with RDF data). It is a no-code way to create a GraphQL resolver for GraphQL queries that have a variable number of input parameters and return a variable number of nested fields.

It has been released as an open-source project located at https://github.com/aws/amazon-neptune-for-graphql.

You can install the utility using NPM like this (see Installation and Setup for details):

```
npm i @aws/neptune-for-graphql -g
```

The utility can discover the graph schema of an existing Neptune property graph, including nodes, edges, properties, and edge cardinality. It then generates a GraphQL schema with the directives needed to map the GraphQL types to the nodes and edges the database, and auto-generates resolver code. The resolver code is designed to minimize latency by returning only the data requested by the GraphQL query.

You can also start with an existing GraphQL schema and an empty Neptune database, and let the utility infer the directives needed to map that GraphQL schema to the nodes and edges of data to be loaded into the database. Or, you can start with a GraphQL schema and directives that you've already created or modified.

The utility is capable of creating all the AWS resources it needs for its pipeline, including the AWS AppSync API, the IAM roles, the data source, schema, and resolver, and the AWS Lambda function that queries Neptune.

> **Note**
>
> Command-line examples here assume a Linux console. If you are using Windows, replace the backslashes ('\') at the end of lines with carets ('^').

Topics

- Installing and setting up the Amazon Neptune utility for GraphQL
- Scanning data in an existing Neptune database
- Starting from a GraphQL schema with no directives
- Working with directives for a GraphQL schema
- Command-line arguments for the GraphQL utility

Installing and setting up the Amazon Neptune utility for GraphQL

If you're going to use the utility with an existing Neptune database, you need it to be able to connect to the database endpoint. By default, a Neptune database is accessible only from within the VPC where it is located.

Because the utility is a Node.js command-line tool, you must have Node.js (version 18 or above) installed for the utility to run. To install Node.js on an EC2 instance in the same VPC as your Neptune database, follow the instructions here. The minimum size instance to run the utility is t2.micro. During the creation of the instance select the Neptune database VPC from the **Common Security Groups** pulldown menu.

To install the utility itself on an EC2 instance or your local machine, use NPM:

```
npm i @aws/neptune-for-graphql -g
```

You can then run the utility's help command to check whether it installed properly:

```
neptune-for-graphql --help
```

You may also want to install the AWS CLI to manage AWS resources.

Scanning data in an existing Neptune database

Whether you are familiar with GraphQL or not, the command below is the fastest way to create a GraphQL API. This assumes that you have installed and configured the Neptune utility for GraphQL as described in the installation section, so that it's connected to the endpoint of your Neptune database.

```
neptune-for-graphql \
  --input-graphdb-schema-neptune-endpoint (your neptune database endpoint):(port
 number) \
  --create-update-aws-pipeline \
  --create-update-aws-pipeline-name (your new GraphQL API name) \
  --output-resolver-query-https
```

The utility analyzes the database to discover the schema of the nodes, edges, and properties in it. Based on that schema, it infers a GraphQL schema with associated queries and mutations. Then it creates an AppSync GraphQL API and the required AWS resources to use it. These resources include a pair of IAM roles and a Lambda function containing the GraphQL resolver code.

When the utility has finished you'll find a new GraphQL API in the AppSync console under the name you assigned in the command. To test it, use the AppSync **Queries** option on the menu.

If you run the same command again after adding more data to the database, will update the AppSync API and Lambda code accordingly.

To release all the resources associated with the command, run:

```
neptune-for-graphql \
  --remove-aws-pipeline-name (your new GraphQL API name from above)
```

Starting from a GraphQL schema with no directives

You can start from an empty Neptune database and use a GraphQL schema with no directives to create the data and query it. The command below automatically creates AWS resources to do this:

```
neptune-for-graphql \
  --input-schema-file (your GraphQL schema file)
  --create-update-aws-pipeline \
  --create-update-aws-pipeline-name (name for your new GraphQL API) \
  --create-update-aws-pipeline-neptune-endpoint (your Neptune database endpoint):(port
 number) \
  --output-resolver-query-https
```

The GraphQL schema file must include the GraphQL schema types, as shown in the TODO example below. The utility analyzes your schema and creates an extended version based on your types. It adds queries and mutations for the nodes stored in the graph database, and if your schema has nested types, it adds relationships between the types stored as edges in the database.

The utility creates an AppSync GraphQL API, and all the AWS resources required. These include a pair of IAM roles and a Lambda function that contains the GraphQL resolver code. When the command completes, you can find a new GraphQL API with the name you specified in the AppSync console. To test it, use **Queries** in the AppSync menu.

The example below illustrates how this works:

Todo example, starting from a GraphQL schema with no directives

In this example we start from a Todo GraphQL schema with no directives, which you can find in the *???samples???* directory. It includes these two types:

```
type Todo {
  name: String
  description: String
  priority: Int
  status: String
  comments: [Comment]
```

```
}

type Comment {
  content: String
}
```

This command processes the Todo schema and an endpoint of an empty Neptune database to create a GraphQL API in AWS AppSync:

```
neptune-for-graphql /
  --input-schema-file ./samples/todo.schema.graphql \
  --create-update-aws-pipeline \
  --create-update-aws-pipeline-name TodoExample \
  --create-update-aws-pipeline-neptune-endpoint (empty Neptune database endpoint):(port
 number) \
  --output-resolver-query-https
```

The utility creates a new file in the output folder called `TodoExample.source.graphql`, and the GraphQL API in AppSync. The utility infers the following:

- In the Todo type it added `@relationship` for a new CommentEdge type. This instructs the resolver to connect Todo to Comment using a graph database edge called CommentEdge.
- It added a new input called TodoInput to help the queries and mutations.
- It added two queries for each type (Todo, Comment): one to retrieve a single type using an `id` or any of the type fields listed in the input, and the other to retrieve multiple values, filtered using the input for that type.
- It added three mutations for each type: create, update and delete. The type to delete is specified using an `id` or the input for that type. These mutations affect the data stored in the Neptune database.
- It added two mutations for connections: connect and delete. They take as input the node ids of the from and to vertices used by Neptune and the connection are edges in the database.

The resolver recognizes the queries and mutations by their names, but you can customize them as shown below.

Here is the content of the `TodoExample.source.graphql` file:

```
type Todo {
  _id: ID! @id
```

```
  name: String
  description: String
  priority: Int
  status: String
  comments(filter: CommentInput, options: Options): [Comment] @relationship(type:
 "CommentEdge", direction: OUT)
  bestComment: Comment @relationship(type: "CommentEdge", direction: OUT)
  commentEdge: CommentEdge
}

type Comment {
  _id: ID! @id
  content: String
}

input Options {
  limit: Int
}

input TodoInput {
  _id: ID @id
  name: String
  description: String
  priority: Int
  status: String
}

type CommentEdge {
  _id: ID! @id
}

input CommentInput {
  _id: ID @id
  content: String
}

input Options {
  limit: Int
}

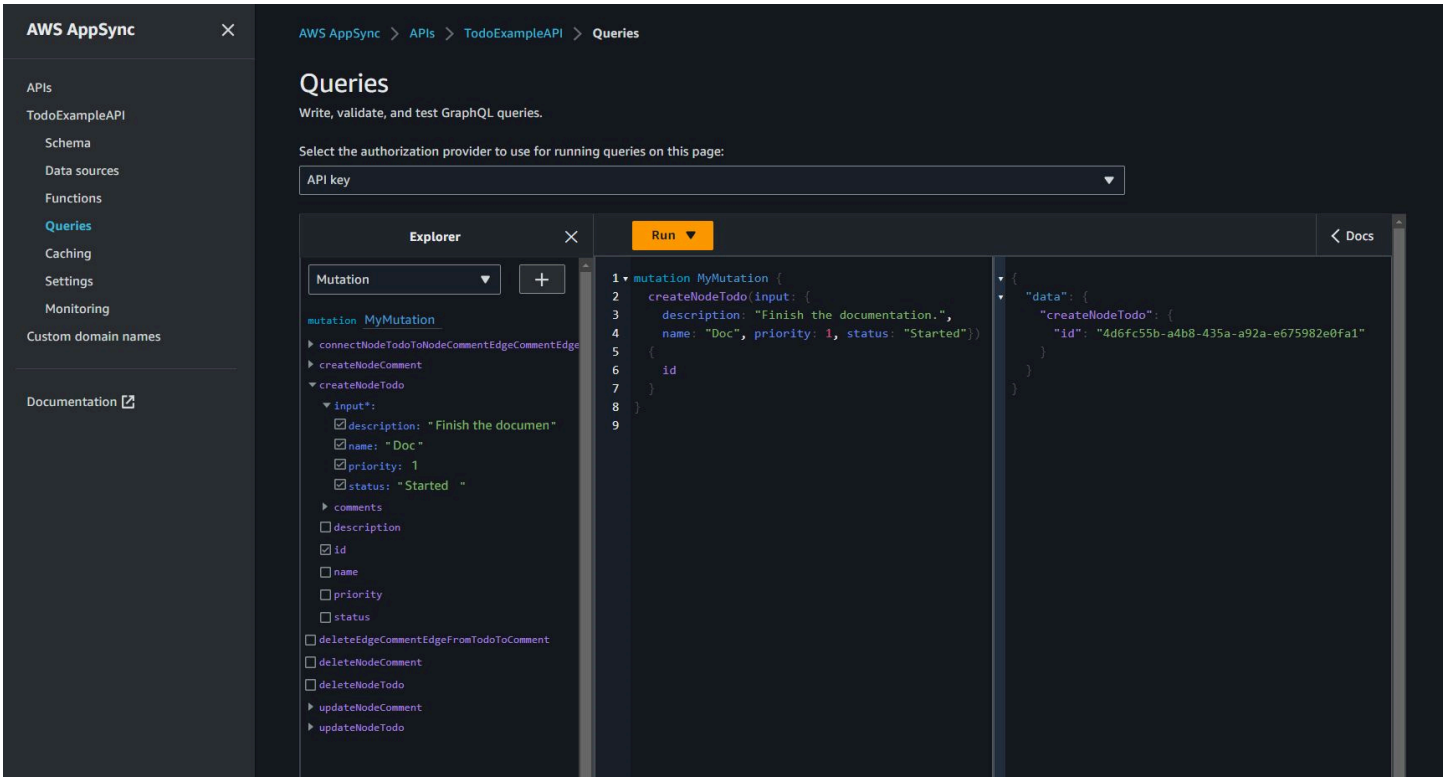
type Query {
  getNodeTodo(filter: TodoInput, options: Options): Todo
  getNodeTodos(filter: TodoInput): [Todo]
  getNodeComment(filter: CommentInput, options: Options): Comment
```

```
  getNodeComments(filter: CommentInput): [Comment]
}

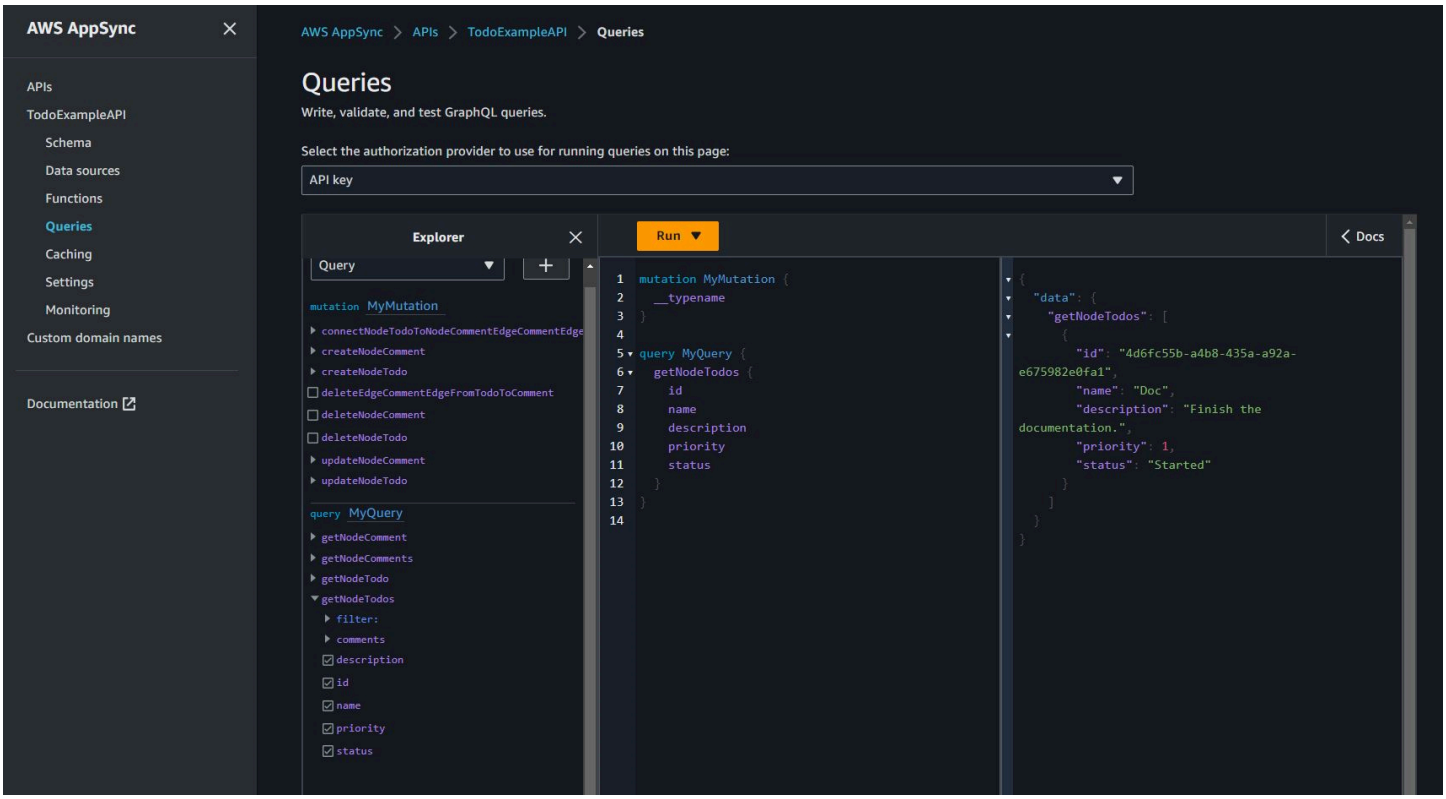
type Mutation {
  createNodeTodo(input: TodoInput!): Todo
  updateNodeTodo(input: TodoInput!): Todo
  deleteNodeTodo(_id: ID!): Boolean
  connectNodeTodoToNodeCommentEdgeCommentEdge(from_id: ID!, to_id: ID!): CommentEdge
  deleteEdgeCommentEdgeFromTodoToComment(from_id: ID!, to_id: ID!): Boolean
  createNodeComment(input: CommentInput!): Comment
  updateNodeComment(input: CommentInput!): Comment
  deleteNodeComment(_id: ID!): Boolean
}

schema {
  query: Query
  mutation: Mutation
}
```

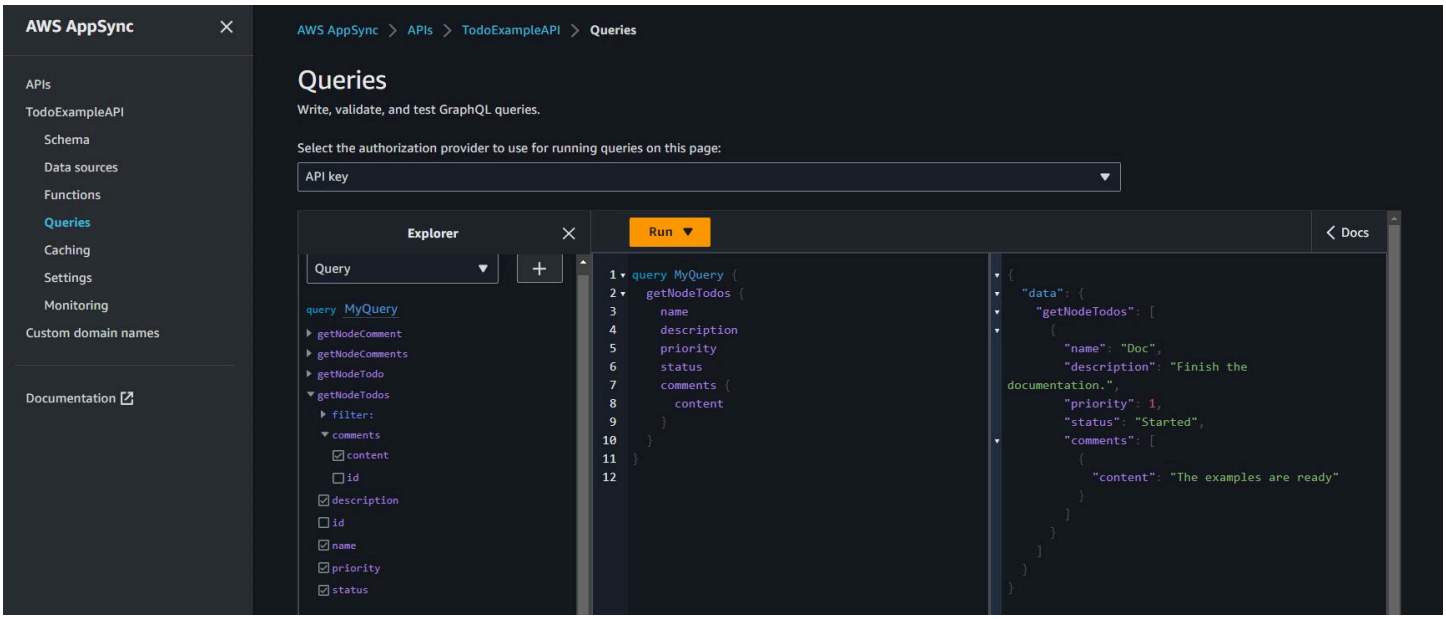
Now you can create and query data. Here is a snapshot of the AppSync **Queries** console used to test the new GraphQL API, named `TodoExampleAPI` in this case. In the middle window, the Explorer shows you a list of queries and mutations from which you can pick a query, the input parameters, and the return fields. This screenshot shows the the creation of a Todo node type using the `createNodeTodo` mutation:



This screenshot shows querying all Todo nodes using the `getNodeTodos` query:



After having created a Comment using `createNodeComment`, you can use the `connectNodeTodoToNodeCommentEdgeCommentEdge` mutation to connect them by specifying their ids. Here is a nested query to retrieve Todos and their attached comments:



If you want to make changes to the `TodoExample.source.graphql` file as described in Working with directives, you can then use the edited schema as input and run the utility again. The utility will then modify the GraphQL API accordingly.

Working with directives for a GraphQL schema

You can start from a GraphQL schema that already has directives, using a command like the following:

```
neptune-for-graphql \
  --input-schema-file (your GraphQL schema file with directives) \
  --create-update-aws-pipeline \
  --create-update-aws-pipeline-name (name for your new GraphQL API) \
  --create-update-aws-pipeline-neptune-endpoint  (empty Neptune database
 endpoint):(port number) \
  --output-resolver-query-https
```

You can modify directives that the utility has created or add your own directives to a GraphQL schema. Here are some of the ways to work with directives:

Running the utility so that it doesn't generate mutations

To prevent the utility from generating muatations in the GraphQL API, use the the `--output-schema-no-mutations` option in the `neptune-for-graphql` command.

The `@alias` directive

The `@alias` directive can be applied to GraphQL schema types or fields. It maps different names between the graph database and the GraphQL schema. The syntax is:

```
@alias(property: (property name))
```

In the example below `airport` is the graph database node label mapped to the `Airport` GraphQL type, and `desc` is the the graph node property mapped to the `description` field (see the Air Routes Example):

```
type Airport @alias(property: "airport") {
  city: String
  description: String @alias(property: "desc")
}
```

Note that standard GraphQL formatting calls for Pascal-casing type namess and camel-casing field names.

The `@relationship` directive

The `@relationship` directive maps nested GraphQL types to graph database edges. The syntax is:

```
@relationship(edgeType: (edge name), direction: (IN or OUT))
```

Here is an example command:

```
type Airport @alias(property: "airport") {
  ...
  continentContainsIn: Continent @relationship(edgeType: "contains", direction: IN)
  countryContainsIn: Country @relationship(edgeType: "contains", direction: IN)
  airportRoutesOut(filter: AirportInput, options: Options): [Airport]
 @relationship(edgeType: "route", direction: OUT)
  airportRoutesIn(filter: AirportInput, options: Options): [Airport]
 @relationship(edgeType: "route", direction: IN)
```

```
}
```

You can find `@relationship` directives in both the Todo example and the Air Routes Example.

The `@graphQuery` and `@cypher` directives

You can define openCypher queries to resolve a field value, add queries or add mutations. For example, this adds a new `outboundRoutesCount` field to the `Airport` type to count the outboud routes:

```
type Airport @alias(property: "airport") {
  ...
  outboundRoutesCount: Int @graphQuery(statement: "MATCH (this)-[r:route]->(a) RETURN
 count(r)")
}
```

Here an example of new queries and mutations:

```
type Query {
  getAirportConnection(fromCode: String!, toCode: String!): Airport \
    @cypher(statement: \
      "MATCH (:airport{code: '$fromCode'})-[:route]->(this:airport)-[:route]-
>(:airport{code:'$toCode'})")
}

type Mutation {
  createAirport(input: AirportInput!): Airport @graphQuery(statement: "CREATE
 (this:airport {$input}) RETURN this")
  addRoute(fromAirportCode:String, toAirportCode:String, dist:Int): Route \
    @graphQuery(statement: \
     "MATCH (from:airport{code:'$fromAirportCode'}),
 (to:airport{code:'$toAirportCode'}) \
      CREATE (from)-[this:route{dist:$dist}]->(to) \
      RETURN this")
}
```

Note that if you omit the `RETURN`, the resolver assumes the keyword `this` is the returning scope.

You can also add a query or mututation using a Gremlin query:

```
type Query {
  getAirportWithGremlin(code:String): Airport \
```

```
    @graphQuery(statement: "g.V().has('airport', 'code', '$code').elementMap()")  #
 single node
  getAirportsWithGremlin: [Airport] \
    @graphQuery(statement: "g.V().hasLabel('airport').elementMap().fold()")       #
 list of nodes
  getCountriesCount: Int \
    @graphQuery(statement: "g.V().hasLabel('country').count()")                   #
 scalar
}
```

At this time Gremlin queries are limited to ones that return scalar values, or `elementMap()` for a single node, or `elementMap().fold()` for a list of nodes.

The `@id` directive

The `@id` directive identifies the field mapped to the `id` graph database entity. Graph databases like Amazon Neptune always have a unique `id` for nodes and edges that is assigned during bulk imports or that is autogenerated. For example:

```
type Airport {
  _id: ID! @id
  city: String
  code: String
}
```

Reserved type, query and mutation names

The utility autogenerates queries and mutations to creeate a working GraphQL API. The pattern of these names is recognized by the resolver and is reserved. Here are examples for the type `Airport` and the connecting type `Route`:

The `Options` type is reserved.

```
input Options {
  limit: Int
}
```

The `filter` and `options` function parameters are reserved.

```
type Query {
  getNodeAirports(filter: AirportInput, options: Options): [Airport]
```

```
}
```

The getNode prefix of query names is reserved, and prefixes of mutations names like `createNode`, `updateNode`, `deleteNode`, `connectNode`, `deleteNode`, `updateEdge`, and `deleteEdge` are reserved.

```
type Query {
  getNodeAirport(id: ID, filter: AirportInput): Airport
  getNodeAirports(filter: AirportInput): [Airport]
}

type Mutation {
  createNodeAirport(input: AirportInput!): Airport
  updateNodeAirport(id: ID!, input: AirportInput!): Airport
  deleteNodeAirport(id: ID!): Boolean
  connectNodeAirportToNodeAirportEdgeRout(from: ID!, to: ID!, edge: RouteInput!): Route
  updateEdgeRouteFromAirportToAirport(from: ID!, to: ID!, edge: RouteInput!): Route
  deleteEdgeRouteFromAirportToAirport(from: ID!, to: ID!): Boolean
}
```

Applying changes to the GraphQL schema

You can modify the GraphQL source schema and run the utility again, getting the latest schema from your Neptune database. Every time the utility discovers a new schema in the database, it generates a new GraphQL schema.

You can also manually edit the GraphQL source schema and run the utility again using the source schema as input instead of the Neptune database endpoint.

Finally, you can put your changes in a file using this JSON format:

```
[
  {
    "type": "(GraphQL type name)",
    "field": "(GraphQL field name)",
    "action": "(remove or add)",
    "value": "(value)"
  }
]
```

For example:

```
[
  {
    "type": "Airport",
    "field": "outboundRoutesCountAdd",
    "action": "add",
    "value":"outboundRoutesCountAdd: Int @graphQuery(statement: \"MATCH (this)-
[r:route]->(a) RETURN count(r)\")"
  },
  {
    "type": "Mutation",
    "field": "deleteNodeVersion",
    "action": "remove",
    "value": ""
  },
  {
    "type": "Mutation",
    "field": "createNodeVersion",
    "action": "remove",
    "value": ""
  }
]
```

Then, as you run the utility on this file using the `--input-schema-changes-file` parameter in the command, the utility applies your changes at once.

Command-line arguments for the GraphQL utility

- **`--help, -h`** – Returns help text for the GraphQL utility to the console.

- **`--input-schema`** ***`(schema text)`*** – A GraphQL schema, with or without directives, to use as input.

- **`--input-schema-file`** ***`(file URL)`*** – The URL of a file containing a GraphQL schema to use as input.

- **`--input-schema-changes-file`** ***`(file URL)`*** – The URL of a file containing changes you want made to a GraphQL schema. If you run the utility against a Neptune database multiple

times, and also manually change the GraphQL source schema, maybe adding a custom query, your manual chnages will be lost. To avoid this, put your changes in a changes file and pass it in using this argument.

The changes file uses the following JSON format:

```
[
  {
    "type": "(GraphQL type name)",
    "field": "(GraphQL field name)",
    "action": "(remove or add)",
    "value": "(value)"
  }
]
```

See the Todo example for more information.

- **`--input-graphdb-schema`** ***`(schema text)`*** – Instead of running the utility against a Neptune database, you can express a graphdb schema in text form to use as input. A graphdb schema has a JSON format like this:

```
{
  "nodeStructures": [
    { "label":"nodelabel1",
      "properties": [
        { "name":"name1", "type":"type1" }
      ]
    },
    { "label":"nodelabel2",
      "properties": [
          { "name":"name2", "type":"type1" }
      ]
    }
  ],
  "edgeStructures": [
    {
      "label":"label1",
      "directions": [
        { "from":"nodelabel1", "to":"nodelabel2", "relationship":"ONE-ONE|ONE-MANY|
MANY-MANY"  }
```

```
      ],
      "properties": [
        { "name":"name1", "type":"type1" }
      ]
    }
  ]
}
```

- **`--input-graphdb-schema-file` *`(file URL)`*** – Instead of running the utility against a Neptune database, you can save a graphdb schema in in a file to use as input. See `--input-graphdb-schema` above for an example of the JSON format for a graphdb schema file.

- **`--input-graphdb-schema-neptune-endpoint` *`(endpoint URL)`*** – The Neptune database enpoint from which the utility should extract the graphdb schema.

- **`--output-schema-file` *`(file name)`*** – The output file name for the GraphQL schema. If not specified the default is `output.schema.graphql`, unless a pipeline name has been set using `--create-update-aws-pipeline-name`, in which case the default file name is *`(pipline name)`*`.schema.graphql`.

- **`--output-source-schema-file` *`(file name)`*** – The output file name for the GraphQL schema with directives. If not specified the default is `output.source.schema.graphql`, unless a pipeline name has been set using `--create-update-aws-pipeline-name`, in which case the default name is *`(pipeline name)`*`.source.schema.graphql`.

- **`--output-schema-no-mutations`** – If this argument is present, the utility generates no mutations in the GraphQL API, only queries.

- **`--output-neptune-schema-file` *`(file name)`*** – The output file name for Neptune graphdb schema that the utility discovers. If not specified the default is `output.graphdb.json`, unless a pipeline name has been set using `--create-update-aws-pipeline-name`, in which case the default file name is *`(pipeline name)`*`.graphdb.json`.

- **`--output-js-resolver-file` *`(file name)`*** – The output file name for a copy of the resolver code. If not specified the default is `output.resolver.graphql.js`, unless a pipeline name has been set using `--create-update-aws-pipeline-name`, in which case the file name is *`(pipeline name)`*`.resolver.graphql.js`.

  This file is zipped in the code package uploaded to the Lambda function that runs the resolver.

- **`--output-resolver-query-sdk`** – This argument specifies that the utility's Lambda function should query Neptune using the Neptune data SDK, which has been available starting with Neptune engine version 1.2.1.0.R5 (this is the default). However, if the utility detects an older Neptune engine version, it suggests using the HTTPS Lambda option instead, which you can invoke using the `--output-resolver-query-https` argument.

- **`--output-resolver-query-https`** – This argument specifies that the utility's Lambda function should query Neptune using the Neptune HTTPS API.

- **`--create-update-aws-pipeline`** – This argument triggers the creation of the AWS resources for the GraphQL API to use, including the AppSync GraphQL API and the Lambda that runs the resolver.

- **`--create-update-aws-pipeline-name` *`(pipeline name)`*** – This argument sets the name for the pipeline, like the `pipeline-name` API for AppSync or `pipeline-name` function for the Lambda function. If a name is not specified, `--create-update-aws-pipeline` uses the `Neptune` database name.
- **`--create-update-aws-pipeline-region` *`(AWS region)`*** – This argument sets the AWS region in which the pipeline for the GraphQL API is created. If not specified, the default region is either `us-east-1` or the region where the Neptune database is located, extracted from the database endpoint.
- **`--create-update-aws-pipeline-neptune-endpoint` *`(endpoint URL)`*** – This argument sets the Neptune database endpoint used by the Lambda function to query the

database. If not set, the endpoint set by `--input-graphdb-schema-neptune-endpoint` is used.

- **`--remove-aws-pipeline-name` *`(pipeline name)`*** – This argument removes a pipeline created using `--create-update-aws-pipeline`. The resources to remove are listed in a file named *`(pipeline name)`*`.resources.json`.
- **`--output-aws-pipeline-cdk`** – This argument triggers the creation of a CDK file that can be used to create the AWS resources for the GraphQL API, including the AppSync GraphQL API and the Lambda function that runs the resolver.
- **`--output-aws-pipeline-cdk-neptume-endpoint` *`(endpoint URL)`*** – This argument sets the Neptune database endpoint used by the Lambda function to query the Neptune database. If not set, the endpoint set by `--input-graphdb-schema-neptune-endpoint` is used.
- **`--output-aws-pipeline-cdk-name` *`(pipeline name)`*** – This argument sets the pipeline name for the AppSync API and the Lambda pipeline-name function to use. If not specified, `--create-update-aws-pipeline` uses the Neptune database name.
- **`--output-aws-pipeline-cdk-region` *`(AWS region)`*** – This sets the AWS region in which the pipeline for the GraphQL API is created. If not specified, it defaults to `us-east-1` or region where the Neptune database is located, extracted from the database endpoint.
- **`--output-aws-pipeline-cdk-file` *`(file name)`*** – This sets the CDK file name. If not set the default is *`(pipeline name)`*`-cdk.js`.

Nodestream

Nodestream is a framework for dealing with semantically modeling data as a graph. It is designed to be flexible and extensible, allowing you to define how data is collected and modeled as a graph. It uses a pipeline-based approach to define how data is collected and processed, and it provides a way to define how the graph should be updated when the schema changes. All of this is done using a simple, human-readable configuration file in yaml format. To accomplish this, Nodestream uses a number of core concepts, including pipelines, extractors, transformers, filters, interpreters, interpretations, and migrations.

Beginning with Nodestream 0.12, Amazon Neptune is supported for both Neptune Database and Neptune Analytics.

Please view the Nodestream documentation for details on how to configure and use Nodestream with Neptune: Nodestream support for Amazon Neptune.

Nodestream with Neptune currently supports standard ETL pipelines as well as time to live (TTL) pipelines. ETL pipelines enable bulk data ingestion into Neptune from a much broader range of data sources and formats than have previously been possible in Neptune including:

- Software Bill of Materials
- Files including CSV, JSON, JSONL, Parquet, txt and yaml
- Kafka
- Athena
- REST APIs

Nodestream fully supports IAM authentication when connecting to Amazon Neptune, as long as credentials are properly configured. See the boto3 credentials guide for more information on correctly configuring credentials.

Nodestream's TTL mechanism also enables new capabilities not previously available in Neptune. By annotating ingested graph elements with timestamps, Nodestream can create pipelines which automatically expire and remove data that has passed a configured lifespan.

Neptune Service Errors

Amazon Neptune has two different sets of errors:

- The graph engine errors that are for the Neptune DB cluster endpoints only.
- The errors that are associated with the APIs for creating and modifying Neptune resources with the AWS SDK and AWS Command Line Interface (AWS CLI).

Topics

- Insufficient DB instances available
- Graph Engine Error Messages and Codes
- DB Cluster Management API Error Messages and Codes
- Neptune Loader Error and Feed Messages

Insufficient DB instances available

The `InsufficientDBInstanceCapacity` error can be returned when you try to create, start, or modify a DB instance. It can also be returned when you try to restore a DB instance from a DB snapshot. When this error is returned, a common cause is that the specific DB instance class isn't available in the requested Availability Zone. You can try one of the following to solve the problem:

- Retry the request with a different DB instance class.
- Retry the request with a different Availability Zone.
- Retry the request without specifying an explicit Availability Zone.

For information about troubleshooting instance capacity issues for Amazon EC2, see Insufficient instance capacity in the Amazon EC2 User Guide.

Graph Engine Error Messages and Codes

Amazon Neptune endpoints return the standard errors for Gremlin and SPARQL when encountered.

Errors that are specific to Neptune can also be returned from the same endpoints. This section documents Neptune error messages, codes, and recommended actions.

> **Note**
>
> These errors are for the Neptune DB cluster endpoints only. The APIs for creating and modifying Neptune resources with the AWS SDK and AWS CLI have a different set of common errors. For information about those errors, see the section called "API Errors".

Graph Engine Error Format

Neptune error messages return a relevant HTTP error code and a JSON-formatted response.

```
HTTP/1.1 400 Bad Request
x-amzn-RequestId: LDM6CJP8RMQ1FHKSC1RBVJFPNVV4KQNSO5AEMF66Q9ASUAAJG
Content-Type: application/x-amz-json-1.0
Content-Length: 465
Date: Thu, 15 Mar 2017 23:56:23 GMT

{
    "requestId": "0dbcded3-a9a1-4a25-b419-828c46342e47",
    "code": "ReadOnlyViolationException",
    "detailedMessage": "The request is rejected because it violates some read-only
 restriction, such as a designation of a replica as read-only."
}
```

Graph Engine Query Errors

The following table contains the error code, message, and HTTP status.

It also indicates whether it is OK to retry the request. Generally, it is OK to retry the request if it might succeed on a new try.

Neptune Service Error Code	HTTP status	Ok to Retry?	Message
`AccessDeniedException`	403	No	Authentication or authorization failure.
`BadRequestException`	400	No	The request could not be completed.

Neptune Service Error Code	HTTP status	Ok to Retry?	Message
`BadRequestException`	400	No	Request size exceeds max allowed value of 157286400 bytes.
`CancelledByUserException`	500	Yes	The request processing was cancelled by an authorized client.
`ConcurrentModificationException`	500	Yes	The request processing did not succeed due to a modification conflict. The client should retry the request.

Neptune Service Error Code	HTTP status	Ok to Retry?	Message
`ConstraintViolationException`	400	Yes	The query engine discovered, during the execution of the request, that the completion of some operation is impossible without violating some data integrity constraints, such as persistence of in- and out-vertices while adding an edge. Such conditions are typically observed if there are concurrent modifications to the graph, and are transient. The client should retry the request.
`FailureByQueryException`	500	Yes	Calling `fail()` caused request processing to fail. The client may retry the request presuming the logic in the query could be expected to succeed on a later attempt. As a result, the choice to retry will be query specific.

Neptune Service Error Code	HTTP status	Ok to Retry?	Message
`InternalFailureException`	500	No	The request processing has failed.
`InvalidNumericDataException`	400	No	Invalid use of numeric data which cannot be represented in 64-bit storage size.
`InvalidParameterException`	400	No	An invalid or out-of-range value was supplied for some input parameter or invalid syntax in a supplied RDF file.
`MalformedQueryException`	400	No	The request is rejected because it contains a query that is syntactically incorrect or does not pass additional validation.
`MemoryLimitExceededException`	500	Yes	The request processing did not succeed due to lack of memory, but can be retried when the server is less busy.

Neptune Service Error Code	HTTP status	Ok to Retry?	Message
`MethodNotAllowedException`	405	No	The request is rejected because the chosen HTTP method is not supported by the used endpoint.
`MissingParameterException`	400	No	A required parameter for the specified action is not supplied.
`QueryLimitExceededException`	500	Yes	The request processing did not succeed due to the lack of a limited resource, but can be retried when the server is less busy.
`QueryLimitException`	400	No	Size of query exceeds system limit.
`QueryTooLargeException`	400	No	The request was rejected because its body is too large.
`ReadOnlyViolationException`	400	No	The request is rejected because it violates some read-only restriction, such as a designation of a replica as read-only.

Neptune Service Error Code	HTTP status	Ok to Retry?	Message
`ThrottlingException`	500	Yes	Rate of requests exceeds the maximum throughput. OK to retry.
`TimeLimitExceededException`	500	Maybe	The request processing timed out.
`TooManyRequestsException`	429	Yes	The rate of requests exceeds the maximum throughput. OK to retry.
`UnsupportedOperationException`	400	No	The request uses a currently unsupported feature or construct.

> **Important**
>
> Whether or not to retry an error often involves important nuances related to the nature of the failure, the workload, and the error patterns you observe. For detailed guidance on developing a practical retry strategy, see the section called “Exceptions and Retries”.

IAM Authentication Errors

These errors are specific to cluster that have IAM authentication enabled.

The following table contains the error code, message, and HTTP status.

Neptune Service Error Code	HTTP status	Message
Incorrect IAM User/Policy	403	You do not have sufficient access to perform this action.
Incorrect or Missing Region	403	Credential should be scoped to a valid Region, not '*region*'.
Incorrect or Missing Service Name	403	Credential should be scoped to correct service: 'neptune-db '.
Incorrect or Missing Host Header / Invalid Signature	403	The request signature we calculated does not match the signature you provided. Check your AWS Secret Access Key and signing method. Consult the service documentation for details. Host header is missing or hostname is incorrect.
Missing X-Amz-Security-Token	403	'x-amz-security-token ' is named as a SignedHeader , but it does not exist in the HTTP request
Missing Authorization Header	403	The request did not include the required authorization header, or it was malformed.
Missing Authentication Token	403	Missing Authentication Token.
Old Date	403	Signature expired: *20181011T213907Z* is now earlier than *20181011T213915Z* (*20181011T214415Z* - *5 min*.)

Neptune Service Error Code	HTTP status	Message
Future Date	403	Signature not yet current: *20500224T213559Z* is still later than *20181108T225925Z* (*20181108T225425Z* + *5 min*.)
Incorrect Date Format	403	Date must be in ISO-8601 'basic format'. Got '*date*'. See https://en.wikipedia.org/wiki/ISO_8601.
Unknown/Missing Access Key or Session Token	403	The security token included in the request is invalid.
Unknown/Missing Secret Key	403	The request signature we calculated does not match the signature you provided. Check your AWS Secret Access Key and signing method. Consult the service documentation for details. Host header is missing or hostname is incorrect.
`TooManyRequestsException`	429	The rate of requests exceeds the maximum throughput. OK to retry.

DB Cluster Management API Error Messages and Codes

These Amazon Neptune errors are associated with the APIs for creating and modifying Neptune resources with the AWS SDK and AWS CLI.

The following table contains the error code, message, and HTTP status.

Neptune Service Error Code	HTTP status	Message
`AccessDeniedException`	403	You do not have sufficient access to perform this action.
`IncompleteSignature`	400	The request signature does not conform to AWS standards.
`InternalFailure`	500	The request processing has failed because of an unknown error, exception, or failure.
`InvalidAction`	400	The action or operation requested is invalid. Verify that the action is typed correctly.
`InvalidClientTokenId`	403	The X.509 certificate or AWS access key ID provided does not exist in our records.
`InvalidParameterCombination`	400	Parameters that must not be used together were used together.
`InvalidParameterValue`	400	An invalid or out-of-range value was supplied for the input parameter.
`InvalidQueryParameter`	400	An invalid or out-of-range value was supplied for the input parameter.
`MalformedQueryString`	400	The query string contains a syntax error.

Neptune Service Error Code	HTTP status	Message
`MissingAction`	400	The request is missing an action or a required parameter.
`MissingAuthenticationToken`	403	The request must contain either a valid (registered) AWS access key ID or X.509 certificate.
`MissingParameter`	400	A required parameter for the specified action is not supplied.
`OptInRequired`	403	The AWS access key ID needs a subscription for the service.
`RequestExpired`	400	The request reached the service more than 15 minutes after the date stamp on the request or more than 15 minutes after the request expiration date (such as for presigned URLs), or the date stamp on the request is more than 15 minutes in the future.
`ServiceUnavailable`	503	The request has failed due to a temporary failure of the server.
`ThrottlingException`	500	The request was denied due to request throttling.
`ValidationError`	400	The input fails to satisfy the constraints specified by an AWS service.

Neptune Loader Error and Feed Messages

The following messages are returned by the `status` endpoint of the Neptune Loader. For more information, see Get-Status API.

The following table contains loader feed code and description.

Error or Feed Code	Description
LOAD_NOT_STARTED	Load has been recorded but not started.
LOAD_IN_PROGRESS	Indicates that loading is in progress and specifies the number of files currently being loaded. When the loader parses a file, it creates one or more chunks to load in parallel. Because a single file can produce multiple chunks, the file count included with this message is usually less than the number of threads being used by the bulk load process.
LOAD_COMPLETED	Load has completed without any errors or errors within an acceptable threshold.
LOAD_CANCELLED_BY_USER	Load has been cancelled by user.
LOAD_CANCELLED_DUE_TO_ERRORS	Load has been cancelled by the system due to errors.
LOAD_UNEXPECTED_ERROR	Load failed with an unexpected error.
LOAD_FAILED	Load failed as a result of one or more errors.
LOAD_S3_READ_ERROR	Feed failed due to intermittent or transient Amazon S3 connectivity issues. If any of the feeds receive this error, overall load status is set to LOAD_FAILED.

Error or Feed Code	Description
LOAD_S3_ACCESS_DENIED_ERROR	Access was denied to the S3 bucket. If any of the feeds receive this error, overall load status is set to LOAD_FAILED.
LOAD_COMMITTED_W_WRITE_CONFLICTS	Loaded data committed with unresolved write conflicts. The loader will try to resolve the write conflicts in separate transactions and update the feed status as the load progresses. If the final feed status is LOAD_COMMITTED_W_WRITE_CONFLICTS, then try resuming the load and it will likely succeed without write conflicts. A write conflict is not usually related to bad input data, but duplicates in data can increase the likelihood of write conflicts.
LOAD_DATA_DEADLOCK	Load was automatically rolled back due to deadlock.
LOAD_DATA_FAILED_DUE_TO_FEED_MODIFIED_OR_DELETED	Feed failed because file was deleted or updated after load start.
LOAD_FAILED_BECAUSE_DEPENDENCY_NOT_SATISFIED	The load request was not executed because its dependency check fails.
LOAD_IN_QUEUE	The load request has been queued up and is waiting to be executed.
LOAD_FAILED_INVALID_REQUEST	The load failed because the request was invalid (for example, the specified source/bucket may not exist, or the file format is invalid).

Engine releases for Amazon Neptune

Amazon Neptune releases engine updates regularly.

You can determine which engine release version you currently have installed using the instance-status API or the Neptune console. The version number tells you whether you are running an original major release, or a minor release, or a patch release.. For more information about release numbering, see Engine version numbers.

For more information about updates in general, see Cluster maintenance.

From engine release 1.3.0.0 going forward, engine versions will have the structure shown in the table below. The minor version number is the one that will be evaluated for `AutoMinorVersionUpgrade` processing.

Version	Prod versi	Majo versi	Minor version	Patch version	Status	Released	End of life	Upgrade to:
1.5.0.0	1	5	0	0	*active*	2026-10-13	TBD	N/A
1.4.8.1	1	4	8	1	*active*	2026-10-06	TBD	1.5.0.0
1.4.8.0	1	4	8	0	*active*	2026-07-27	2027-10-27	1.4.8.1
1.4.7.1	1	4	7	1	*active*	2026-09-16	2027-06-03	1.4.8.0
1.4.7.0	1	4	7	0	*active*	2026-03-03	2027-06-03	1.4.7.1
1.4.6.3	1	4	6	3	*active*	2025-12-18	2027-06-03	1.4.7.0
1.4.6.2	1	4	6	2	*active*	2025-11-18	2027-06-03	1.4.7.0

Version	Prod versi	Majo versi	Minor version	Patch version	Status	Released	End of life	Upgrade to:
1.4.6.1	1	4	6	1	*active*	2025-09-18	2027-03-06	1.4.6.2
1.4.6.0	1	4	6	0	*active*	2025-09-02	2027-03-06	1.4.6.1
1.4.5.1	1	4	5	1	*active*	2025-06-30	2027-03-06	1.4.6.0
1.4.5.0	1	4	5	0	*active*	2025-04-09	2027-03-06	1.4.5.1
1.4.4.0	1	4	4	0	*active*	2025-02-24	2027-03-06	1.4.5.0
1.4.3.0	1	4	3	0	*active*	2025-01-21	2027-03-06	1.4.4.0
1.4.2.0	1	4	2	0	*active*	2024-12-19	2027-03-06	1.4.3.0
1.4.1.0	1	4	1	0	*active*	2024-11-21	2027-03-06	1.4.2.0
1.4.0.0	1	4	0	0	*active*	2024-11-06	2027-03-06	1.4.1.0
1.3.4.0	1	3	4	0	*active*	2024-10-01	2027-03-06	1.4.0.0
1.3.3.0	1	3	3	0	*active*	2024-08-05	2027-03-06	1.3.4.0
1.3.2.1	1	3	2	1	*active*	2024-06-20	2027-03-06	1.3.3.0

Version	Prod versi	Majo versi	Minor version	Patch version	Status	Released	End of life	Upgrade to:
1.3.2.0	1	3	2	0	*active*	2024-06-10	2027-03-06	1.3.2.1
1.3.1.0	1	3	1	0	*active*	2024-03-06	2027-03-06	1.3.2.1
1.3.0.0	1	3	0	0	*active*	2023-11-15	2027-03-06	1.3.2.1

The table below lists all the engine releases since 1.0.1.0, along with information about version end-of-life. You can use the dates in this table to plan your testing and upgrade cycles.

Version	Major versior	Minor versior	Status	Released	End of life	Upgrade to:
1.2.1.2	1.2	1.2	*active*	2024-08-05	2026-12-04	1.3.0.0
1.2.1.1	1.2	1.1	*active*	2024-03-11	2026-12-04	1.3.0.0
1.2.1.0	1.2	1.0	*active*	2023-03-08	2026-12-04	1.3.0.0
1.2.0.2	1.2	0.2	*active*	2022-11-16	2026-12-04	1.3.0.0
1.2.0.1	1.2	0.1	*active*	2022-10-26	2026-12-04	1.3.0.0
1.2.0.0	1.2	0.0	*active*	2022-07-21	2026-12-04	1.3.0.0
1.1.1.0	1.1	1.0	*active*	2022-04-19	2026-12-04	1.2.1.0
1.1.0.0	1.1	0.0	*deprecated*	2021-11-19	2025-03-15	1.1.1.0
1.0.5.1	1.0	5.1	*deprecated*	2021-10-01	2023-01-30	1.1.0.0
1.0.5.0	1.0	5.0	*deprecated*	2021-07-27	2023-01-30	1.1.0.0
1.0.4.2	1.0	4.2	*deprecated*	2021-06-01	2023-01-30	1.1.0.0

Version	Major version	Minor version	Status	Released	End of life	Upgrade to:
1.0.4.1	1.0	4.1	*deprecated*	2020-12-08	2023-01-30	1.1.0.0
1.0.4.0	1.0	4.0	*deprecated*	2020-10-12	2023-01-30	1.1.0.0
1.0.3.0	1.0	3.0	*deprecated*	2020-08-03	2023-01-30	1.1.0.0
1.0.2.2	1.0	2.2	*deprecated*	2020-03-09	2022-07-29	1.0.3.0
1.0.2.1	1.0	2.1	*deprecated*	2019-11-22	2022-07-29	1.0.3.0
1.0.2.0	1.0	2.0	*deprecated*	2019-11-08	2020-05-19	1.0.3.0
1.0.1.2	1.0	1.2	*deprecated*	2019-10-15	—	—
1.0.1.1	1.0	1.1	*deprecated*	2019-08-13	—	—
1.0.1.0.*	1.0	1.0.*	*deprecated*	2019-07-02 and before	—	—

Major engine version end-of-life planning

Neptune engine versions almost always reach their end of life at the end of a calendar quarter. Exceptions occur only when important security or availability issues arise.

When an engine version reaches its end of life, you will be required to upgrade your Neptune database to a newer version.

In general, Neptune engine versions continue to be available as follows:

- **Minor engine versions:** Minor engine versions remain available for at least 6 months following their release.
- **Major engine versions:** Major engine versions remain available for at least 12 months following their release.

At least 3 months before an engine version reaches its end of life, AWS will send an automated email notification to the email address associated with your AWS account and post the same message to your AWS Health Dashboard. This will give you time to plan and prepare to upgrade.

When an engine version reaches its end of life, you will no longer be able to create new clusters or instances using that version, nor will autoscaling be able to create instances using that version.

An engine version that actually reaches its end of life will automatically be upgraded during a maintenance window. The message sent to you 3 months before the engine version's end of life will contain details about what this automatic update would involve, including the version to which you would be automatically upgraded, the impact on your DB clusters, and actions that we recommend.

Important

You are responsible for keeping your database engine versions current. AWS urges all customers to upgrade their databases to the latest engine version in order to benefit from the most current security, privacy, and availability safeguards. If you operate your database on an unsupported engine or software past the deprecation date ("Legacy Engine"), you face a greater likelihood of security, privacy, and operational risks, including downtime events.

Operation of your database on any engine is subject to the Agreement governing your use of the AWS Services. Legacy Engines are not Generally Available. AWS no longer provides support for the Legacy Engine, and AWS may place limits on the access to or use of any Legacy Engine at any time, if AWS determines the Legacy Engine poses a security or liability risk, or a risk of harm, to the Services, AWS, its Affiliates, or any third party. Your decision to continue running Your Content in a Legacy Engine could result in Your Content becoming unavailable, corrupted, or unrecoverable. Databases running on a Legacy Engine are subject to Service Level Agreement (SLA) Exceptions.

DATABASES AND RELATED SOFTWARE RUNNING ON A LEGACY ENGINE CONTAIN BUGS, ERRORS, DEFECTS, AND/OR HARMFUL COMPONENTS. ACCORDINGLY, AND NOTWITHSTANDING ANYTHING TO THE CONTRARY IN THE AGREEMENT OR THE SERVICE TERMS, AWS IS PROVIDING THE LEGACY ENGINE "AS IS."

Amazon Neptune Engine version 1.5.0.0 (2026-10-13)

As of 2026-10-13, engine version 1.5.0.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

New Features in This Engine Release

- Fast bulk load (offline loader) for high-performance initial loads of large Gremlin CSV datasets into empty graphs. See the Fast bulk load (offline loader) documentation.
- Storage management APIs: use the Neptune storage usage breakdown to analyze storage usage, and the Neptune online index rebuild to defragment and reclaim index storage while your database remains fully operational.
- Gremlin upgrade to Apache TinkerPop 3.8.x. This includes updated Gremlin client minimum versions (Java 11+, Python 3.10+, Go 1.25+, .NET 6.0+), step behavior changes that improve consistency of the language, and new Gremlin features. Given the step behavior changes, it is important that you carefully review the TinkerPop Upgrade Documentation to understand the semantics of Gremlin that have changed and how those changes will affect your existing queries. Given that these are semantic differences, the changes may not surface as errors, but rather change query results. You will need to decide if any of your queries are affected by these changes as part of your upgrade.

Improvements in This Engine Release

Gremlin improvements

- Expanded Gremlin step coverage on the DFE query engine offering significantly wider query pattern support aligned with TinkerPop 3.8.x semantics. See Gremlin step coverage in DFE.

openCypher improvements

- Added native support for the `all()`, `any()`, `none()`, and `single()` predicate functions. These functions previously required list comprehension rewrites (see Rewriting `None`, `All`, and `Any` predicate functions).

Gremlin and openCypher improvements

- Added DFE support for regex predicates in Gremlin and openCypher

General improvements

- Improved query performance for openCypher and SPARQL queries that use variable assignments (such as `WITH` clauses or `BIND`) across join groups, by enabling the query optimizer to make better join ordering decisions based on variables produced in earlier query stages
- Improved slow query logging for openCypher queries executed via Bolt protocol with explicit transactions
- Improved memory efficiency for DFE queries that process large intermediate result sets
- `pg_schema` computation that is interrupted by an engine restart or failover transitions to `Paused` (rather than `Stopped`)

Defects Fixed in This Engine Release

- openCypher `MERGE` clauses now reject null property values instead of silently ignoring them
- Fixed query timeout being incorrectly reported as an internal failure error
- Fixed a query cancellation issue that could reduce the available query-memory budget
- Fixed an issue where Neptune Serverless instances may not scale down after running queries

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.5.0.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.8.0`
- *Gremlin latest version supported:* `3.8.1`
- *openCypher version:* `Neptune-9.0.20190305-1.1`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.5.0.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.5.0.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.5.0.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.8.1 (2026-10-06)

As of 2026-10-06, engine version 1.4.8.1 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Improvements in This Engine Release

General improvements:

- Added support for specifying a lock-wait timeout through a query hint.

Defects fixed in this engine release (1.4.8.1)

General fixes

- Fixed a query cancellation issue that could reduce the available query-memory budget.
- Various stability and correctness fixes for the DFE query engine.

Gremlin fixes

- General stability and security improvements for Gremlin data loading.

SPARQL fixes

- Fixed an issue where a SPARQL query using a predicate alternation (for example, `(:p1|:p2|:p3)`) inside a multi-hop property path (* or +) could return incomplete results.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.8.1, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.8.1

You can upgrade to this release from engine release 1.2.0.0 or above.

> **Upgrading global database clusters to this release**
>
> Minor version upgrades to engine version 1.4.8.1 from versions prior to 1.4.7.0 are not supported for Neptune clusters that are part of a global database. Non-global database clusters and major version upgrades are not affected.
>
> To minor version upgrade a global database cluster running a version prior to 1.4.7.0 to 1.4.8.1, you must first remove the secondary clusters from the global database (see ???), upgrade the primary cluster to 1.4.8.1, and then create new secondary clusters in the global database.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
```

```
    --engine-version 1.4.8.1 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.8.1 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.8.0 (2026-07-27)

As of 2026-07-27, engine version 1.4.8.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

New Features in This Engine Release

- Native RDF export. You can now export RDF data from your Neptune cluster directly to Amazon Simple Storage Service (Amazon S3) using the export API. This feature also supports filtering exports by named graph URIs using the `exportFilter` parameter. See the Native export for RDF data documentation.
- Property graph schema. You can now retrieve the schema of your property graph data, including node labels, edge labels, and their associated properties. See the Property graph schema documentation.
- Dictionary garbage collection now works on clusters with Neptune Streams enabled.

Improvements in This Engine Release

Gremlin improvements:

- Reduced the chance that a Gremlin query will encounter an out-of-memory condition when a serverless instance has scaled down to a low NCU value.

SPARQL improvements:

- Improved performance of very large, multi-operation SPARQL Update requests.
- Improved performance of SPARQL alternative property paths (for example, `?s <a>|<b> ?o`).

Defects Fixed in This Engine Release

General fixes:

- Fixed a rare condition that could prevent the engine from starting successfully under certain configurations.
- Fixed an issue where rolled-back transactions could lead to gradual memory growth over time.
- Various stability and correctness fixes for the DFE query engine.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.8.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.8.0

You can upgrade to this release from engine release 1.2.0.0 or above.

> **Upgrading global database clusters to this release**
>
> Minor version upgrades to engine version 1.4.8.0 from versions prior to 1.4.7.0 are not supported for Neptune clusters that are part of a global database. Non-global database clusters and major version upgrades are not affected.
> To minor version upgrade a global database cluster running a version prior to 1.4.7.0 to 1.4.8.0, you must first remove the secondary clusters from the global database (see ???), upgrade the primary cluster to 1.4.8.0, and then create new secondary clusters in the global database.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.8.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
```

```
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.8.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.7.1 (2026-09-18)

As of 2026-09-18, engine version 1.4.7.1 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects fixed in this engine release (1.4.7.1)

General fixes

- Fixed an issue where the `VolumeBytesLeftTotal` Amazon CloudWatch metric reported incorrect values in engine version 1.4.7.0.
- Various stability and correctness fixes for the DFE query engine.

Gremlin fixes

- Fixed an issue that could cause some Gremlin queries to fail using Neptune serverless when the instance has scaled down close to the minimum NCU value.
- General stability and security improvements for Gremlin data loading.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.7.1, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.7.1

You can upgrade to this release from engine release 1.2.0.0 or above.

> **Upgrading global database clusters to this release**
>
> Minor version upgrades to engine version 1.4.7.1 from versions prior to 1.4.7.0 are not supported for Neptune clusters that are part of a global database. Non-global database clusters and major version upgrades are not affected.
>
> To minor version upgrade a global database cluster running a version prior to 1.4.7.0 to 1.4.7.1, you must first remove the secondary clusters from the global database (see ???), upgrade the primary cluster to 1.4.7.1, and then create new secondary clusters in the global database.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.7.1 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.7.1 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.7.0 (2026-03-03)

As of 2026-03-03, engine version 1.4.7.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

New Features in This Engine Release

- openCypher read from S3 support for Parquet and CSV files via OC. See the the section called "neptune.read()" documentation.
- openCypher geospatial query functions. This release includes 12 Spatial Types functions based on the ISO/IEC 13249-3:2016 standard, a new Geometry property type for POINT stored in a new geopetric index for fast retrival, and support for Well-Known Text (WKT) format. See the Spatial Data and the Spatial Functions documentation.

Improvements in This Engine Release

- Improved query performance for SPARQL subqueries that return small result sets, including those with small LIMIT values
- Improved query performance in cases where variables are constrained by a very large number of constant values (for example, by a SPARQL VALUES clause, or an OpenCypher UNWIND clause)
- Improvements for low latency insert queries via some optimizations to dictionary inserts
- Added new Gremlin language steps into the DFE engine (see Gremlin step coverage in DFE).
  - Path and traversal steps: `order(local)`
  - Aggregate and collection steps: `dedup(local)`
- Performance improvement for OpenCypher queries including `COLLECT(DISTINCT ...)`. The rewrite described in Rewriting COLLECT(DISTINCT ...) queries is no longer needed when using engine version 1.4.7.0 or later.

Defects Fixed in This Engine Release

General fixes:

- Fix for bulk load becoming unresponsive when loading a large number of edge files
- Fix a global database cluster patching issue that affected secondary cluster updates from releases 1.4.0.0, 1.4.1.0, and 1.4.2.0.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.7.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.7.0

You can upgrade to this release from engine release 1.2.0.0 or above.

> **Upgrading global database clusters to this release**
>
> Minor version upgrades to engine version 1.4.7.0 are not supported for Neptune clusters that are part of a global database. Non-global database clusters and major version upgrades are not affected.
>
> To minor version upgrade a global database cluster to 1.4.7.0, you must first remove the secondary clusters from the global database (see ???), upgrade the primary cluster to 1.4.7.0, and then create new secondary clusters in the global database.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.7.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.7.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.6.3 (2025-12-18)

As of 2025-12-18, engine version 1.4.6.3 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects fixed in this engine release (1.4.6.3)

General fixes

- Some exceptions that were erroneously classified as Internal Server Error are now correctly reported as Out of Memory Exceptions during the Data Flow Engine (DFE) execution.

- Fixed an issue that could cause Neptune Database instances to fail during startup when inline edge ID validation encountered unexpected dictionary entry types. Inline edge ID validation now only runs for Gremlin query engine configurations and gracefully handles non-URI dictionary entries.
- Fixed an issue where the Neptune bulk loader could fail to connect to S3 in certain regions due to incorrect endpoint resolution.

openCypher fixes

- Fixed an engine crash that sometimes occurred in queries using `CALL`.
- Fixed timeout and cancellation handling for mutation queries.
- Fixed a bug where the `MERGE` clause gave incorrect results when one of the property values passed to `MERGE` has null value.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.6.3, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.6.3

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.6.3 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.6.3 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.6.2 (2025-11-18)

As of 2025-11-18, engine version 1.4.6.2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects fixed in this engine release

General fixes

- Some exceptions that were erroneously classified as Internal Failure Exceptions (IFEs) are now correctly reported as Concurrent Modification Exceptions (CMEs) during the Data Flow Engine (DFE) execution.

Gremlin fixes

- Improved the engine stability for Gremlin over DFE

openCypher fixes

- Fixed an issue introduced in Neptune engine release 1.4.6.0 where openCypher MERGE queries incorrectly handled incoming relationship directions (←). Queries using patterns like MERGE (n)←[:type]-(m) now create relationships with the correct direction.
- Fixed a failed commit operation on the writer transactions that were not properly rolled back.
- Optimized query plans are now supported for MERGE queries that reference parametrized values.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.6.2, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.6.2

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.6.2 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.6.2 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.6.1 (2025-09-18)

As of 2025-09-18, engine version 1.4.6.1 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects fixed in this engine release

General fixes

- Removed the network checks for clusters that are using non-RFC 1918 conforming private IP ranges that was introduced in Release Amazon Neptune Engine version 1.4.6.0 (2025-09-02).

Gremlin fixes

- Fixed websocket connection management issue with transactions.
- Fixed a rare instance restart issue when using gremlin DFE mode

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.6.1, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.6.1

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.6.1 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.6.1 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.6.0 (2025-09-02)

As of 2025-09-02, engine version 1.4.6.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Warning

The 1.4.6.0 engine version includes new network checks for clusters that are using non-RFC 1918 conforming private IP ranges for database VPC without IAM authentication. If you have this VPC and IAM configuration, you will need to update your database VPC to use RFC 1918 private IP ranges and/or enable IAM authentication to avoid errors with queries after upgrading to 1.4.6.0.

New features in this engine release

- Connect to Neptune using public endpoints. For more information see Neptune public endpoints.

Improvements in this engine release

General Improvements

- Improved SPARQL performance for update operations.
- Improved OpenCypher performance for `CREATE`, `MERGE`, and `SET` (mutations) operations.
- Improved OpenCypher performance for CALL Subquery operations.

openCypher improvements

- Added new query hint to support query level timeout.

Defects fixed in this engine release

Gremlin fixes

- Connections to Gremlin sessions must occur on the same channel that created them, meaning that it is not possible to connect multiple client instances to the same session.
- Gremlin sessions have always closed when the client closes, but they will now also close for a server initiated close of the connection which prevents an unintended or unexpected re-connection.
- Fixed memory leaks for Gremlin queries reading large blob type data.

openCypher fixes

- Fixed variable handling after a `CALL` subquery.
- Fixed an issue with `reduce` function to correctly handle arithmetic overflow scenarios.
- Fixed a memory leak affecting parameterized queries when the Query Plan Cache is enabled.
- Fixed an issue with `NOT EXISTS` used in complex `WHERE` clauses.
- Fix for Cuncurrent Memory Exception (CMEs) being misreported as BadRequestException.

SPARQL fixes

- Fixed an error message for SPARQL `LOAD/UNLOAD` when the remote source is unavailable.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.6.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`

- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.6.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.6.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.6.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
--db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> ```

```
proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.5.1 (2025-06-30)

As of 2025-06-30, engine version 1.4.5.1 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects fixed in this engine release

General fixes

- Fixed an issue with serverless scaling that was impacting all query languages.
- Fixed an issue where an internal error was returned on some OpenCypher queries when running `collect(distinct())` on a list.
- Fixed an issue where a concurrent FTS query causes other concurrent FTS queries to be terminated when using OpenSearch serverless.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.5.1, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.5.1

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.5.1 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.5.1 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.5.0 (2025-04-09)

As of 2025-04-09, engine version 1.4.5.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Warning

We have temporarily paused upgrades to 1.4.5.0 due to issues that can occur in certain Serverless configurations. We recommend upgrading to the 1.4.5.1 engine version. Upgrades to 1.4.5.0 have been temporarily disabled.

New features in this engine release

- Added new Gremlin language steps into the DFE engine.
  - **Path and traversal steps:** asDate(), dateAdd(), dateDiff(), fail(), Inject(), label(), path(), project(), repeat(), sack(), select(), unfold(), disjunct(), drop(), identity(), intersect(), length(), loops(), barrier(), order(), range(), reverse(), sample(), cap(), split(), filter(), flatMap(), map(), sideEffect(), union(), index()
  - **Aggregate and collection steps:** aggregate(global), combine(), count(), dedup(global), fold(), group(), groupCount()
  - **Mathematical steps:** max(), mean(), min(), sum()
  - **Element steps:** otherV(), elementMap(), element(), V(), out(), in(), both(), outE(), inE(), bothE(), outV(), inV(), bothV(), otherV()
  - **Property steps:** properties(), key(), valueMap(), value()
  - **Filter steps:** and(), coalesce(), coin(), is(), local(), none(), not(), or(), where()
  - **String manipulation steps:** concat(), lTrim(), rTrim(), substring(), toLower(), toUpper(), trim()

- **Predicates:**
  - Compare: eq, neq, lt, lte, gt, gte
  - Contains: within, without
  - TextP: endingWith, containing, notStartingWith, notEndingWith, notContaining
  - P: and, or, between, outside, inside

For details on all available Gremlin steps in DFE, refer to Gremlin step coverage in DFE.

Improvements in this engine release

General Improvements

- Slow query log lock wait time improvement. Slow query logs now includes wait time metrics for shared and exclusive locks. They are stored as part of each transaction in case of lazy read-write promotion. These metrics appear in the storageCounters section of the slow query logs.
- Dropped support for the following cipher suites:
  - TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
  - TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
  - TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
  - TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384

Gremlin Improvements

- Added many new steps to the Gremlin language. For more information, see Gremlin step coverage in DFE.

openCypher improvements

- CREATE, MERGE, and SET (mutations) performance improvements.
- CALL Subquery performance improvements.
- Support for HTTP trailing header support for multi-part openCypher responses. For more information, see Optional HTTP trailing headers.
- Added day, month, and year temporal function to openCypher. For more information, see Temporal functions.

```
RETURN day(datetime('2021-06-03T01:48:14Z'))
{
  "results": [{
      "day(datetime('2021-06-03T01:48:14Z'))": 3
    }]
}
```

Defects fixed in this engine release

General fixes

- Fixed issue that dropped the Audit/SlowQueryLog log files.

Gremlin fixes

- Fixed an issue with Gremlin queries running with Result Cache feature disabled. A query ending with iterate() was returning results instead of returning empty response.
- Fixed an issue with Gremlin Result Cache caused by concurrent queries with the same key. One of the concurrent running queries incorrectly returned results instead of returning empty results.
- Fixed an issue with Amazon S3 export queries that would cause a query to fail an Amazon S3 multi-part upload due to timeouts or cancellation, by increasing clean up time.
- Fixed a permission issue related to Gremlin Amazon S3 export.

SPARQL fixes

- Fixed an issue in the handling of SPARQL queries that declare multiple base IRIs that would cause only the initial declaration to be used.
- Fixed an issue in the handling of SPARQL `REPLACE` functions using invalid pattern strings that would cause an error to be returned.
- Fixed an issue in the handling of SPARQL `REPLACE` functions using the case-insensitivity (`"i"`) flag with unicode data.
- Fixed an issue in the parsing of SPARQL queries using invalid `\u` and `\U` codepoint escape sequences which could lead to no response being returned.

- Fixed an issue in the SPARQL `IRI` function that did not always correctly resolve relative IRIs against the current base IRI.
- Fixed an issue that caused `SPARQL INSERT DATA` and `DELETE DATA` updates using prefixed names to fail to correctly resolve relative IRIs against the current base IRI.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.5.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.5.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.5.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
```

```
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.5.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.4.0 (2025-02-24)

As of 2025-02-24, engine version 1.4.4.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Warning**
>
> The query plan cache is temporarily unsupported for the use case of executing parameterized queries involving numeric parameter values, due to a bug in handling duplicate usages of a numeric type parameter in the query. For example:
>
> ```
> MATCH (n:movie) WHERE n.runtime>=$minutes RETURN n
>  UNION
> ```

```
MATCH (n:show) WHERE n.duration>=$minutes RETURN n

parameters={"minutes":130}
```

Queries that do a lot of index searches on statements or dictionary indices could see a 5% performance regression. For example - getting a count of all vertices, or getting the `id` of all vertices would not be affected. Getting all properties of all vertices could see up to a 5% regression.

Defects fixed in this engine release

General fixes

- Fixed an issue where the `WITH` clause containing an asterisk (`*`) and aliasing expressions caused incorrect query parsing in openCypher.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.4.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.4.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.4.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.4.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.3.0 (2025-01-21)

As of 2025-01-21, engine version 1.4.3.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Warning**
>
> The query plan cache is temporarily unsupported for the use case of executing parameterized queries involving numeric parameter values, due to a bug in handling duplicate usages of a numeric type parameter in the query. For example:
>
> ```
> MATCH (n:movie) WHERE n.runtime>=$minutes RETURN n
>  UNION
>  MATCH (n:show) WHERE n.duration>=$minutes RETURN n
>
>  parameters={"minutes":130}
> ```
>
> Queries that do a lot of index searches on statements or dictionary indices could see a 5% performance regression. For example - getting a count of all vertices, or getting the `id` of all vertices would not be affected. Getting all properties of all vertices could see up to a 5% regression.

New features in this engine release

- Exporting Gremlin query results to Amazon S3. Exporting Gremlin query results directly to Amazon S3. This feature allows you to handle large query results efficiently by exporting them to an Amazon S3 bucket, instead of returning them as query response.

  ```
  g.V().
      hasLabel('Comment').
      valueMap().
      call('neptune.query.exportToS3', [
      'destination': 's3://your-bucket/path/result.json',
      'format': 'GraphSONv3',
      'keyArn': 'optional-kms-key-arn'
    ])
  ```

- **R7i instances**. R7i instance family, up to 48xlarge, are now available in the following regions:

- ap-northeast-1 - Asia Pacific (Tokyo)
- ap-northeast-2 - Asia Pacific (Seoul)
- ap-south-1 - Asia Pacific (Mumbai)
- ap-southeast-1 - Asia Pacific (Singapore)
- ap-southeast-2 - Asia Pacific (Sydney)
- ap-southeast-3 - Asia Pacific (Jakarta)
- ca-central-1 - Canada (Central)
- eu-central-1 - Europe (Frankfurt)
- eu-north-1 - Europe (Stockholm)
- eu-south-2 - Europe (Spain)
- eu-west-1 - Europe (Ireland)
- eu-west-2 - Europe (London)
- eu-west-3 - Europe (Paris)
- us-east-1 - US East (N. Virginia)
- us-east-2 - US East (Ohio)
- us-west-1 - US West (N. California)
- us-west-2 - US West (Oregon)

Improvements in this engine release

General Improvements

- Lab mode support for dictionary garbage collection (GC).

  When enabled, the unused dictionary entries are cleaned up by a background job. It does not reduce `VolumeBytesUsed`, it frees up space in the index for new inserts. The rate of growth in `VolumeBytesUsed` is likely to be less when dictionary GC is enabled relative to when it is not. This works for property graph data (inserted via Gremlin or openCypher) when the `neptune_streams` parameter is not enabled. For more information, see Neptune dictionary garbage collection

Defects fixed in this engine release

General fixes

- Fixed two memory leak issues affecting FreeableMemory when DFE engine used.

openCypher fixes

- Resolve issue with MERGE ON MATCH / ON CREATE for duplicate rows.

```
UNWIND [1, 1] AS id
MERGE (n:Person {id: id})
  ON CREATE SET n.p = 5
  ON MATCH SET n.p = 6
```

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.3.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.3.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.3.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.3.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.2.0 (2024-12-19)

As of 2024-12-19, engine version 1.4.2.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Warning

The query plan cache is temporarily unsupported for the use case of executing parameterized queries involving numeric parameter values, due to a bug in handling duplicate usages of a numeric type parameter in the query. For example:

```
MATCH (n:movie) WHERE n.runtime>=$minutes RETURN n
      UNION
      MATCH (n:show) WHERE n.duration>=$minutes RETURN n

      parameters={"minutes":130}
```

Queries that do a lot of index searches on statements or dictionary indices could see a 5% performance regression. For example - getting a count of all vertices, or getting the `id` of all vertices would not be affected. Getting all properties of all vertices could see up to a 5% regression.

Defects fixed in this engine release

General fixes

- Fixed execution plan and performance for queries with multiple property accesses over the same variable inside OPTIONAL MATCH and list comprehension. For example:

```
MATCH (n)
  WHERE n.name = 'A'
OPTIONAL MATCH (n)-[:knows]->(m)
  WHERE m.name = 'C' or m.city = 'B'
RETURN m
```

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.2.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`

- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.2.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.2.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.2.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
--db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> ```

```
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.1.0 (2024-11-21)

As of 2024-11-21, engine version 1.4.1.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Warning

The query plan cache is temporarily unsupported for the use case of executing parameterized queries involving numeric parameter values, due to a bug in handling duplicate usages of a numeric type parameter in the query. For example:

```
MATCH (n:movie) WHERE n.runtime>=$minutes RETURN n
      UNION
      MATCH (n:show) WHERE n.duration>=$minutes RETURN n

      parameters={"minutes":130}
```

Queries that do a lot of index searches on statements or dictionary indices could see a 5% performance regression. For example - getting a count of all vertices, or getting the `id` of all vertices would not be affected. Getting all properties of all vertices could see up to a 5% regression.

New features in this engine release

- Added support for `CALL` subquery with a read-only subquery, allowing execution of operations within a defined scope. A `CALL` subquery is executed once for each incoming row and the

variables returned in a subquery are available to the outer scope of the enclosing query. Variables from outer scope can be imported into a `CALL` subquery using an importing `WITH` clause. For more information, see CALL subquery support in Neptune.

```
MATCH (origin:airport {code:"AUS"})-[:route]->(stopover)
CALL {
  WITH stopover
  MATCH (stopover)-[r:route]->(destination)
  RETURN destination
  ORDER BY r.dist DESC LIMIT 2
}
RETURN stopover, destination
```

- Added openCypher functions. We are introducing eight new functions to help with strings, collections operations and collection sorting. These include: `textIndexOf`, `collToSet`, `collSubtract`, `collIntersection`, `collSort`, `collSortMaps`, `collSortMulti`, and `collSortNodes`. See Neptune openCypher functions for the description, input parameters, output and examples.

Improvements in this engine release

Gremlin Improvements

- New lab mode parameter `AccurateQRCMemoryEstimation`. Gremlin query result cache, when enabled, allows caching of query results on the database. By default an approximate estimate is used to determine the size of the result cached. With this lab mode param `AccurateQRCMemoryEstimation` enabled, the size estimation for cached results will use an accurate size estimate instead of approximate.
- Fixed an issue with "not" filter optimization in Gremlin queries executing on default execution engine. This issue affected queries when edges are filtered using not() step combined with either of outV()/inV()/otherV() steps. Sample queries include:
  - `g.E().hasLabel("knows").not(outV().hasId("5"))`
  - `g.V().has('airport','code','SDF').outE().where(not(otherV().has(id, within('1','5','7')))).count()`

openCypher improvements

- Improved performance for queries which use large static lists or maps. Certain queries with UNWIND over a large list of nested maps used to insert / upsert a node with properties see significant performance improvements.
- Introduces a new openCypher query hint to instruct engine to assume consistent datatypes for values used in the query. See AssumeConsistentDataTypes for details about the new openCypher query hint.
- Introduces a set of new openCypher functions for handling text and collection values.

Defects fixed in this engine release

Gremlin fixes

- Fixed an issue in TinkerPop OSS code path to build a Bytecode representation of a traversal query when any of `withStrategies()/withoutStrategies()/with()` steps are used on `GraphTraversalSource` "g" object. The issue incorrectly appended new instructions to Bytecode instead of replacing existing instructions for the same strategy and caused a cache key mismatch during result cache invalidation to clear the stored results.

openCypher fixes

- Corrected behavior for `` `~id`match `` in CREATE/MERGE/MATCH clauses. When using an invalid `` `~id` `` value like null or non-string types, a correct exception is now thrown for CREATE/MERGE clauses and zero result is returned for a `MATCH` clause.
- Fixed IFE when user uses a value of unsupported type with aggregation functions (i.e. sum(<string>)).
- Fixed an issue where some low-latency mutation queries from a large workload of queries failed with an OutOfMemory error.

SPARQL fixes

- Fixed an audit log issue when handling SPARQL queries that contain the `'%'` character.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.1.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.1.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.1.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.1.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
--db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
--db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that

begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.4.0.0 (2024-11-06)

As of 2024-11-06, engine version 1.4.0.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Note**
>
> Engine release 1.3.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.3.0.0 to engine version 1.3.0.0 or above, you must re-create all of your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.3`. Earlier releases used parameter group family `neptune1`, or `neptune1.2`. and those parameter groups won't work with release 1.3.0.0 and above. Similarly, you should use 1.4.0.0 cluster parameter groups for engine versions 1.4.0.0 and above. See Amazon Neptune parameter groups for more information.

Warning

The query plan cache is temporarily unsupported for the use case of executing parameterized queries involving numeric parameter values, due to a bug in handling duplicate usages of a numeric type parameter in the query. For example:

```
MATCH (n:movie) WHERE n.runtime>=$minutes RETURN n
      UNION
      MATCH (n:show) WHERE n.duration>=$minutes RETURN n

      parameters={"minutes":130}
```

Queries that do a lot of index searches on statements or dictionary indices could see a 5% performance regression. For example - getting a count of all vertices, or getting the `id` of all vertices would not be affected. Getting all properties of all vertices could see up to a 5% regression.

New features in this engine release

- When an edge is added to a property graph without an explicit ID, by default the server assigns a UUID-based edge ID, which is stored in the dictionary. Now by setting a new cluster parameter, `neptune_enable_server_generated_edge_id = 1`, the server will assign IDs using an internally managed 8-byte integer, without any dictionary overhead. This results in storage savings and improved query performance without any changes to queries. This feature is currently supported for inserts via the Gremlin query language only.
- Added support for Gremlin limit() step execution in nested traversals for DFE engine.

```
g.V().project("foo").by(out().order().by(T.id).limit(1))
```

Improvements in this engine release

General Improvements

- Neptune will automatically reclaim undo storage held by large transactions once the transaction is complete and logs are no longer needed for recovery.

- Support for global database survivable replica. This feature allows secondary cluster to still serve read requests during a writer instance restart on the primary cluster. Previously, when a writer instance restarted, all reader instances in a secondary cluster also restarted. With this release, secondary cluster reader instances continue to serve read requests during a writer instance restart, improving read availability in the cluster.
- Audit logs are now written synchronously, which guarantees that every query is logged. This may affect performance for particularly large queries (>100kb) or high-throughput workloads (>1000 qps).

Gremlin Improvements

- The per-query timeout is by default enforced to be smaller than the cluster-level timeout. In a previous release, this check was introduced but needed to be explicitly enabled via the lab-mode parameter 'StrictTimeoutValidation'. With this release, 'StrictTimeoutValidation' will be enabled by default and must be disabled explicitly to keep the old behavior.

openCypher improvements

- In a previous release we introduced extended datetime format support, enabled via a lab mode parameter `DatetimeMillisecond`. This extended datetime format support is now enabled by default.

SPARQL improvements

- New explicit IAM actions for query permissions.

```
Previously:
COPY: WriteDataViaQuery & ReadDataViaQuery
MOVE: WriteDataViaQuery & DeleteDataViaQuery
DELETEINSERT: ReadDataViaQuery & DeleteDataViaQuery

 Now,
COPY: WriteDataViaQuery & ReadDataViaQuery & DeleteDataViaQuery
MOVE: WriteDataViaQuery & ReadDataViaQuery & DeleteDataViaQuery
DELETEINSERT: ReadDataViaQuery, WriteDataViaQuery if there is INSERT clause,
 DeleteDataViaQuery if there is DELETE clause.
```

Defects fixed in this engine release

General fixes

- Fixed an issue with Serverless instances that could lead to a database restart while scaling up.
- Fixed an issue related to audit log file management that could cause log files to be inaccessible for download or rotation, and in some cases increase CPU usage.
- Fixed a query issue related to optimization delaying map output generation in the DFE engine.
- Fixed an issue which caused mismatched timestamps between the audit logs and slow query logs.

Gremlin fixes

- Resolved an issue in Gremlin WebSocket connection management where queries running for time exceeding the connection idle timeout were prematurely terminated. This specifically affected Python Gremlin clients using the AIOHTTP transport.

openCypher fixes

- Fixed an issue in collect step which caused an internal failure exception when null values are present during the collect(distinct(n)) query construct.
- Fixed an issue where a `NullPointerException` might occur in queries when the query plan cache is enabled.
- Fixed an issue which evaluated more data than needed when a query contains LIMIT clause.
- Fixed an issue where using range operations (<, <=, >, >=) in a parameterized query with a query plan cache would produce duplicate results.
- Fixed an issue that transposes result columns when UNION and UNION ALL operations are performed using bolt connections.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.4.0.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`

- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.4.0.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.4.0.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.4.0.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
--db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> ```

```
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.3.4.0 (2024-10-01)

As of 2024-10-01, engine version 1.3.4.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

Engine release 1.3.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.3.0.0 to engine version 1.3.0.0 or above, you must re-create all of your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.3`. Earlier releases used parameter group family `neptune1`, or `neptune1.2`. and those parameter groups won't work with release 1.3.0.0 and above. Similarly, you should use 1.4.0.0 cluster parameter groups for engine versions 1.4.0.0 and above. See Amazon Neptune parameter groups for more information.

Warning

An issue was detected with SPARQL 1.1 update operations that may be present under certain conditions when update operators are used with action-based authorization policies. If you are using the SPARQL 1.1 update operations with action-based authorization policies, we recommend to upgrade to the latest Neptune minor engine version (at least 1.3.4.0) which includes a fix for this issue.

The query plan cache was temporarily disabled for parameterized queries involving numeric parameter values due to an issue in handling duplicate usages of a numeric type parameter as in the following query:

```
MATCH (n:movie) WHERE n.runtime>=$minutes RETURN n
UNION
MATCH (n:show) WHERE n.duration>=$minutes RETURN n

parameters={"minutes":130}
```

Improvements in this engine release

- Added support for Gremlin limit() step execution in nested traversals for DFE engine.
- Added CloudWatch metrics related to Gremlin result cache as listed below, which can be useful in diagnosing and tuning result cache latency. See Neptune Metrics for details.

```
NumResultCacheHit
NumResultCacheMiss
ResultCacheSizeInBytes
ResultCacheItemCount
ResultCacheOldestItemTimestamp
ResultCacheNewestItemTimestamp
```

Defects fixed in this engine release

General improvements

- Fixed an issue where in rare cases, the engine crashes instead of returning a query error.

Gremlin fixes

- We have improved request handling and error reporting when a client or a proxy sends a websocket upgrade request over an established/used HTTP connection (previous to that 400 responses with error "no gremlin script supplied, code MissingParameterException" were returned).

- Optimized the handling of `mergeV` steps with single cardinality property value updates. For example, the query below is now natively supported in Neptune.

```
g.mergeV([(T.id): 1234]). option(onMatch, ['age': single(20), 'name':
 single('alice'), 'city': set('miami')])
```

- Fixed a Gremlin DFE query evaluation issue that caused queries to fail with an `InternalFailureException`. This error occurred with certain patterns of `select` as shown in the following example:

```
g.V("1").as("a").as("b").select("a","b").dedup()
```

openCypher fixes

- Fixed an issue where running `collect(distinct())` with null values present caused an error to be returned.
- Fixed an issue where running a parameterized query containing range filter (</<=/>/>= against the parameter value) leads to duplicate/missing results.
- Fixed a bug where DFE engine produced more output than requested in limit queries, which could lead to out of memory errors.

SPARQL fixes

- Fixed an issue where running a federated SPARQL update query on IAM authentication-enabled clusters caused an error to be returned.
- Fixed action-based permissions for SPARQL 1.1 update operations.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.3.4.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.3.4.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.3.4.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.3.4.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.3.3.0 (2024-08-05)

As of 2024-08-05, engine version 1.3.3.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

Engine release 1.3.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.3.0.0 to engine version 1.3.0.0 or above, you must re-create all of your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.3`. Earlier releases used parameter group family `neptune1`, or `neptune1.2`. and those parameter groups won't work with release 1.3.0.0 and above. Similarly, you should use 1.4.0.0 cluster parameter groups for engine versions 1.4.0.0 and above. See Amazon Neptune parameter groups for more information.

Warning

Engine release 1.3.3.0 introduced some potential issues that you should be aware of. See the section below on Mitigating issues in release 1.3.3.0 for more information.

Defects fixed in this engine release

General improvements

- Fixed an issue where the engine becomes unstable when there are a high number of predicates in the predicate cache.

openCypher fixes

- Fixed an issue where query execution can remain stuck after an internal exception is thrown.
- Fixed an issue where a query can fail with an internal exception when using query plan cache.

SPARQL fixes

- Fixed an issue with the SPARQL 1.1 Graph Store HTTP Protocol (GSP) that may be present under certain conditions when GSP is used with action-based authorization policies.

Mitigating issues in release 1.3.3.0

- Queries using numerical filter values can return incorrect results when using the query plan cache. To avoid the issue, use the query hint `QUERY:PLANCACHE "disabled"` to skip the query plan cache. For example, use:

```
USING QUERY:PLANCACHE "disabled"
MATCH (n:person)
WHERE n.yearOfBirth > $year
RETURN n

parameters={"year":1950}
```

- Queries using the same parameter name multiple times can fail with the error `Parameter name should not be a number and/or contain _internal_ or _modified_user_ string within it. These are reserved for planCache. Otherwise, rerun with HTTP parameter planCache=disabled`. Either skip the query plan cache like above in such cases, or duplicate the parameters as in this example:

```
MATCH (n:movie) WHERE n.runtime>=$minutes RETURN n
UNION
MATCH (n:show) WHERE n.duration>=$minutes RETURN n

parameters={"minutes":130}
```

  Use the hint `QUERY:PLANCACHE "disabled"` or modify the parameters:

```
MATCH (n:movie) WHERE n.runtime>=$rt_min RETURN n
UNION
```

```
MATCH (n:show) WHERE n.duration>=$dur_min RETURN n

parameters={"rt_min":130, "dur_min":130}
```

- Queries executed with the Bolt protocol can produce incorrect results if the query is a UNION or UNION ALL query. To avoid the issue, consider executing the particular query with the HTTP endpoint. Alternatively, execute each part of the union separately when using the Bolt protocol.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.3.3.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.3.3.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.3.3.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.3.3.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.3.2.1 (2024-06-20)

As of 2024-06-20, engine version 1.3.2.1 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Note**
>
> Engine release 1.3.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.3.0.0 to engine version 1.3.0.0 or above, you must re-create all of your

existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.3`. Earlier releases used parameter group family `neptune1`, or `neptune1.2`. and those parameter groups won't work with release 1.3.0.0 and above. Similarly, you should use 1.4.0.0 cluster parameter groups for engine versions 1.4.0.0 and above. See Amazon Neptune parameter groups for more information.

Warning

Engine release 1.3.2.1 introduced some potential issues that you should be aware of. See the section below on Mitigating issues in release 1.3.2.1 for more information.

Defects fixed in this engine release

openCypher fixes

- A bug was detected in the query plan cache feature for parameterized queries that contain an inner `WITH` clause having `SKIP` and `LIMIT` as parameters. The SKIP/LIMIT values were not properly parameterized, and as a result, subsequent executions of the same cached query plan with different parameter values would still return the same results as the first execution. This has been fixed.

```
# insert some nodes
UNWIND range(1, 10) as i CREATE (s {name: i}) RETURN s

# sample query
MATCH (p)
WITH p ORDER BY p.name SKIP $s LIMIT $l
RETURN p.name as res

# first time executing with {"s": 2, "l": 1}
{
  "results" : [ {
    "res" : 3
  } ]
}

# second time executing with {"s": 2, "l": 10}
# due to bug, produces
```

```
{
  "results" : [ {
    "res" : 3
  } ]
}
# with fix, produces correct results:
{
  "results" : [ {
    "res" : 3
  }, {
    "res" : 4
  }, {
    "res" : 5
  }, {
    "res" : 6
  }, {
    "res" : 7
  }, {
    "res" : 8
  }, {
    "res" : 9
  }, {
    "res" : 10
  } ]
}%
```

- Fixed a bug where parameterized mutation queries throw an `InternalFailureException` when the parameter that was passed is not already present in the database.
- Fixed a bug where parameterized Bolt queries get stuck after hitting a race condition during query resource cleanup.

Changes in 1.3.2.1 carried over from 1.3.2.0

Improvements carried over from engine release 1.3.2.0

General improvements

- Support for TLS version 1.3 including cipher suites TLS_AES_128_GCM_SHA256 and TLS_AES_256_GCM_SHA384. TLS 1.3 is an option - TLS 1.2 is still the minimum.
- openCypher extended support for dateime format is in lab_mode for this version. We encourage you to test it.

Gremlin improvements

- TinkerPop 3.7.x upgrade
  - Provides a large expansion of the Gremlin language.
    - New steps for processing strings, lists and dates.
    - New syntax for specifying cardinality within the `mergeV()` step.
    - `union()` can now be used as a start step.
    - To learn more about the changes in 3.7.x, see the TinkerPop upgrade documentation.
  - When upgrading client Gremlin language drivers for Java, note that the serializer classes have ungone some renaming. You will need to update package and class naming in your configuration files and in code, if specified.
- `StrictTimeoutValidation` (only when enabled via labmode `StrictTimeoutValidation` by including `StrictTimeoutValidation=enabled`): When the `StrictTimeoutValidation` parameter has a value of `enabled`, a per-query timeout value specified as a request option or a query hint cannot exceed the value set globally in the parameter group. In such a case, Neptune will throw a `InvalidParameterException`. This setting can be confirmed in a response on the `/status` endpoint when the value is `disabled`, and in Neptune versions 1.3.2.0 and 1.3.2.1 the default value of this parameter is `Disabled`.

openCypher improvements

- Amazon Neptune engine version 1.3.2.0 delivers up to 9x faster and 10x higher throughput for openCypher query performance vs. previous engine releases.
- Low latency queries and throughput performance improvement: Overall performance improvements for low latency openCypher queries. The new version also improves the throughput for such queries. The improvements are more significant when parameterized queries are used.
- Support for Query Plan Cache: When a query is submitted to Neptune, the query string is parsed, optimized, and transformed into a query plan, which then gets executed by the engine. Applications are often backed by common query patterns that are instantiated with different values. Query plan cache can reduce the overall latency by caching the query plans and thereby avoiding parsing and optimization for such repeated patterns. See Query plan cache in Amazon Neptune for more details.
- Performance Improvement for DISTINCT aggregation queries.
- Performance improvement for joins involving nullable variables.

- Performance improvement for queries involving not equals to id(node/relationship) predicate.
- Extended support for datetime functionality (Only enabled via lab mode `DatetimeMillisecond` by including `DatetimeMillisecond=enabled`. For more information, see Temporal support in the Neptune openCypher implementation (Neptune Analytics and Neptune Database 1.3.2.0 and above).

Defect fixes carried over from engine release 1.3.2.0

General improvements

- Updated the NeptuneML error message when validating access to Graphlytics buckets.

Gremlin fixes

- Fixed missing label information in DFE query translation, for scenarios where non-path contributing steps contain labels. For example:

```
g.withSideEffect('Neptune#useDFE', true).
  V().
  has('name', 'marko').
  has("name", TextP.regex("mark.*")).as("p1").
  not(out().has("name", P.within("peter"))).
  out().as('p2').
  dedup('p1', 'p2')
```

- Fixed a `NullPointerException` bug in the DFE query translation, which occurs when a query is executed in two DFE fragments, and the first fragment is optimized to an unsatisfiable node. For example:

```
g.withSideEffect('Neptune#useDFE', true).
  V().
  has('name', 'doesNotExists').
  has("name", TextP.regex("mark.*")).
  inject(1).
  V().
  out().
  has('name', 'vadas')
```

- Fixed a bug where Neptune could throw an `InternalFailureException` when a query contains ValueTraversal inside by() modulator and its input is Map. For example:

```
g.V().
  hasLabel("person").
  project("age", "name").by("age").by("name").
  order().by("age")
```

openCypher fixes

- Improved UNWIND operations (e.g. expand a list of values into individual values) to help prevent out of memory (OOM) situations. For example:

```
MATCH (n)-->(m)
WITH collect(m) AS list
UNWIND list AS m
RETURN m, list
```

- Fixed custom id optimization in case of multiple MERGE operations where id is injected via UNWIND. For example:

```
UNWIND [{nid: 'nid1', mid: 'mid1'}, {nid: 'nid2', mid: 'mid2'}] as ids
MERGE (n:N {`~id`: ids.nid})
MERGE (m:M {`~id`: ids.mid})
```

- Fixed memory explosion while planning for complex queries with property access and multiple hops with bi-directional relationships. For example:

```
MATCH (person1:person)-[:likes]->(res)-[:partOf]->(group)-[:knows]-(:entity {name:
 'foo'}),
       (person1)-[:knows]->(person2)-[:likes]-(res2), (comment)-[:presentIn]->(:Group
 {name: 'barGroup'}),
       (person1)-[:commented]->(comment2:comment)-[:partOf]->(post:Post), (comment2)-
[:presentIn]->(:Group {name: 'fooGroup'}),
       (comment)-[:contains]->(info:Details)-[:CommentType]->(:CommentType {name:
 'Positive'}),
       (comment2)-[:contains]->(info2:Details)-[:CommentType]->(:CommentType {name:
 'Positive'})
WHERE datetime('2020-01-01T00:00') <= person1.addedAfter <=
 datetime('2023-01-01T23:59') AND comment.approvedBy = comment2.approvedBy
MATCH (comment)-[:contains]->(info3:Details)-[:CommentType]->(:CommentType {name:
 'Neutral'})
RETURN person1, group.name, info1.value,  post.ranking, info3.value
```

- Fixed aggregation queries with null as group by variables. For example:

```
MATCH (n)
RETURN null AS group, sum(n.num) AS result
```

SPARQL fixes

- Fixed the SPARQL parser to improve parsing time for large queries like INSERT DATA containing many triples and large tokens.

Mitigating issues in release 1.3.2.1

- Queries using numerical filter values can return incorrect results when using the query plan cache. To avoid the issue, use the query hint `QUERY:PLANCACHE "disabled"` to skip the query plan cache. For example, use:

```
USING QUERY:PLANCACHE "disabled"
MATCH (n:person)
WHERE n.yearOfBirth > $year
RETURN n

parameters={"year":1950}
```

- Queries using the same parameter name multiple times can fail with the error `Parameter name should not be a number and/or contain _internal_ or _modified_user_ string within it. These are reserved for planCache. Otherwise, rerun with HTTP parameter planCache=disabled`. Either skip the query plan cache like above in such cases, or duplicate the parameters as in this example:

```
MATCH (n:movie) WHERE n.runtime>=$minutes RETURN n
UNION
MATCH (n:show) WHERE n.duration>=$minutes RETURN n

parameters={"minutes":130}
```

  Use the hint `QUERY:PLANCACHE "disabled"` or modify the parameters:

```
MATCH (n:movie) WHERE n.runtime>=$rt_min RETURN n
```

```
UNION
MATCH (n:show) WHERE n.duration>=$dur_min RETURN n

parameters={"rt_min":130, "dur_min":130}
```

- Queries executed with the Bolt protocol can produce incorrect results if the query is a UNION or UNION ALL query. To avoid the issue, consider executing the particular query with the HTTP endpoint. Alternatively, execute each part of the union separately when using the Bolt protocol.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.3.2.1, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.3.2.1

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.3.2.1 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.3.2.1 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.3.2.0 (2024-06-10)

As of 2024-06-10, engine version 1.3.2.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Note**
>
> Engine release 1.3.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.3.0.0 to engine version 1.3.0.0 or above, you must re-create all of your

existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.3`. Earlier releases used parameter group family `neptune1`, or `neptune1.2`. and those parameter groups won't work with release 1.3.0.0 and above. Similarly, you should use 1.4.0.0 cluster parameter groups for engine versions 1.4.0.0 and above. See Amazon Neptune parameter groups for more information.

Warning

Engine release 1.3.2.0 introduced some potential issues that you should be aware of. See the section below on Mitigating issues in release 1.3.2.0 for more information.

Improvements in this engine release

General improvements

- Support for TLS version 1.3 including cipher suites TLS_AES_128_GCM_SHA256 and TLS_AES_256_GCM_SHA384. TLS 1.3 is an option - TLS 1.2 is still the minimum.

Gremlin improvements

- TinkerPop 3.7.x upgrade
  - Provides a large expansion of the Gremlin language.
    - New steps for processing strings, lists and dates.
    - New syntax for specifying cardinality withing the `mergeV()` step.
    - `union()` can now be used as a start step.
    - To learn more about the changes in 3.7.x, see the TinkerPop upgrade documentation.
  - When upgrading client Gremlin language drivers for Java, note that the serializer classes have ungone some renaming. You will need to update package and class naming in your configuration files and in code, if specified.
- `StrictTimeoutValidation` (only when enabled via labmode `StrictTimeoutValidation` by including `StrictTimeoutValidation=enabled`): When the `StrictTimeoutValidation` parameter has a value of `enabled`, a per-query timeout value specified as a request option or a query hint cannot exceed the value set globally in the parameter group. In such a case, Neptune will throw a `InvalidParameterException`. This setting can be confirmed in a response on

the `/status` endpoint when the value is `disabled`, and in Neptune version 1.3.2.0 the default value of this parameter is `Disabled`.

openCypher improvements

- Amazon Neptune engine version 1.3.2.0 delivers up to 9x faster and 10x higher throughput for openCypher query performance vs. previous engine releases.
- Low latency queries and throughput performance improvement: Overall performance improvements for low latency openCypher queries. The new version also improves the throughput for such queries. The improvements are more significant when parameterized queries are used.
- Support for Query Plan Cache: When a query is submitted to Neptune, the query string is parsed, optimized, and transformed into a query plan, which then gets executed by the engine. Applications are often backed by common query patterns that are instantiated with different values. Query plan cache can reduce the overall latency by caching the query plans and thereby avoiding parsing and optimization for such repeated patterns.
- Performance Improvement for DISTINCT aggregation queries.
- Performance improvement for joins involving nullable variables.
- Performance improvement for queries involving not equals to id(node/relationship) predicate.
- Extended support for datetime functionality (Only enabled via lab mode `DatetimeMillisecond` by including `DatetimeMillisecond=enabled`. For more information, see Temporal support in the Neptune openCypher implementation (Neptune Analytics and Neptune Database 1.3.2.0 and above).

Defects fixed in this engine release

General improvements

- Updated the NeptuneML error message when validating access to Graphlytics buckets.

Gremlin fixes

- Fixed missing label information in DFE query translation, for scenarios where non-path contributing steps contain labels. For example:

```
g.withSideEffect('Neptune#useDFE', true).
```

```
  V().
  has('name', 'marko').
  has("name", TextP.regex("mark.*")).as("p1").
  not(out().has("name", P.within("peter"))).
  out().as('p2').
  dedup('p1', 'p2')
```

- Fixed a `NullPointerException` bug in the DFE query translation, which occurs when a query is executed in two DFE fragments, and the first fragment is optimized to an unsatisfiable node. For example:

```
g.withSideEffect('Neptune#useDFE', true).
  V().
  has('name', 'doesNotExists').
  has("name", TextP.regex("mark.*")).
  inject(1).
  V().
  out().
  has('name', 'vadas')
```

- Fixed a bug where Neptune could throw an `InternalFailureException` when a query contains ValueTraversal inside by() modulator and its input is Map. For example:

```
g.V().
  hasLabel("person").
  project("age", "name").by("age").by("name").
  order().by("age")
```

openCypher fixes

- Improved UNWIND operations (e.g. expand a list of values into individual values) to help prevent out of memory (OOM) situations. For example:

```
MATCH (n)-->(m)
WITH collect(m) AS list
UNWIND list AS m
RETURN m, list
```

- Fixed custom id optimization in case of multiple MERGE operations where id is injected via UNWIND. For example:

```
UNWIND [{nid: 'nid1', mid: 'mid1'}, {nid: 'nid2', mid: 'mid2'}] as ids
MERGE (n:N {`~id`: ids.nid})
MERGE (m:M {`~id`: ids.mid})
```

- Fixed memory explosion while planning for complex queries with property access and multiple hops with bi-directional relationships. For example:

```
MATCH (person1:person)-[:likes]->(res)-[:partOf]->(group)-[:knows]-(:entity {name:
 'foo'}),
      (person1)-[:knows]->(person2)-[:likes]-(res2), (comment)-[:presentIn]->(:Group
 {name: 'barGroup'}),
      (person1)-[:commented]->(comment2:comment)-[:partOf]->(post:Post), (comment2)-
[:presentIn]->(:Group {name: 'fooGroup'}),
      (comment)-[:contains]->(info:Details)-[:CommentType]->(:CommentType {name:
 'Positive'}),
      (comment2)-[:contains]->(info2:Details)-[:CommentType]->(:CommentType {name:
 'Positive'})
WHERE datetime('2020-01-01T00:00') <= person1.addedAfter <=
 datetime('2023-01-01T23:59') AND comment.approvedBy = comment2.approvedBy
MATCH (comment)-[:contains]->(info3:Details)-[:CommentType]->(:CommentType {name:
 'Neutral'})
RETURN person1, group.name, info1.value,  post.ranking, info3.value
```

- Fixed aggregation queries with null as group by variables. For example:

```
MATCH (n)
RETURN null AS group, sum(n.num) AS result
```

SPARQL fixes

- Fixed the SPARQL parser to improve parsing time for large queries like INSERT DATA containing many triples and large tokens.

Mitigating issues in release 1.3.2.0

- For version 1.3.2.0, we have detected an issue in query plan cache when `skip` or `limit` is used in an inner `WITH` clause and are parameterized. For example:

```
MATCH (n:Person)
```

```
WHERE n.age > $age
WITH n skip $skip LIMIT $limit
RETURN n.name, n.age

parameters={"age": 21, "skip": 2, "limit": 3}
```

In this case, the parameter values for skip and limit from the first plan will be applied to subsequent queries, too, leading to unexpected results.

Mitigation

To prevent this issue, add the query hint `QUERY:PLANCACHE "disabled"` when submitting a query that includes a parameterized skip and/or limit sub-clause. Alternatively, you can hard-code the values into the query.

Option 1: Using the Query Hint to disable plan cache:

```
Using QUERY:PLANCACHE "disabled"
MATCH (n:Person) WHERE n.age > $age
WITH n skip $skip LIMIT $limit
RETURN n.name, n.age

parameters={"age": 21, "skip": 2, "limit": 3}
```

Option 2: Using hard-coded values for skip and limit:

```
MATCH (n:Person)
WHERE n.age > $age
WITH n skip 2 LIMIT 3
RETURN n.name, n.age

parameters={"age": 21}
```

- Queries using numerical filter values can return incorrect results when using the query plan cache. To avoid the issue, use the query hint `QUERY:PLANCACHE "disabled"` to skip the query plan cache. For example, use:

```
USING QUERY:PLANCACHE "disabled"
MATCH (n:person)
WHERE n.yearOfBirth > $year
RETURN n
```

```
parameters={"year":1950}
```

- Queries using the same parameter name multiple times can fail with the error `Parameter name should not be a number and/or contain _internal_ or _modified_user_string within it. These are reserved for planCache. Otherwise, rerun with HTTP parameter planCache=disabled`. Either skip the query plan cache like above in such cases, or duplicate the parameters as in this example:

```
MATCH (n:movie) WHERE n.runtime>=$minutes RETURN n
UNION
MATCH (n:show) WHERE n.duration>=$minutes RETURN n

parameters={"minutes":130}
```

  Use the hint `QUERY:PLANCACHE "disabled"` or modify the parameters:

```
MATCH (n:movie) WHERE n.runtime>=$rt_min RETURN n
UNION
MATCH (n:show) WHERE n.duration>=$dur_min RETURN n

parameters={"rt_min":130, "dur_min":130}
```

- Queries executed with the Bolt protocol can produce incorrect results if the query is a UNION or UNION ALL query. To avoid the issue, consider executing the particular query with the HTTP endpoint. Alternatively, execute each part of the union separately when using the Bolt protocol.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.3.2.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.7.1`
- *Gremlin latest version supported:* `3.7.1`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.3.2.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.3.2.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.3.2.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine version 1.3.1.0 (2024-03-06)

As of 2024-03-06, engine version 1.3.1.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Note**
>
> Engine release 1.3.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.3.0.0 to engine version 1.3.0.0 or above, you must re-create all of your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.3`. Earlier releases used parameter group family `neptune1`, or `neptune1.2`. and those parameter groups won't work with release 1.3.0.0 and above. Similarly, you should use 1.4.0.0 cluster parameter groups for engine versions 1.4.0.0 and above. See Amazon Neptune parameter groups for more information.

Improvements in this engine release

General improvements

- Neptune has improved the warning shown in profile/explain.
- Removed obsolete NIST EC curves from the default named groups used during TLS negotiation. The curves removed are sect409k1, sect409r1, and sect571k1.

Gremlin improvements

- Improved DFE statistics computation to avoid very high NCUs of Serverless instance.

- Gremlin performance improvement for WITHIN.

Defects fixed in this engine release

Gremlin fixes

- Miscellaneous improvements to Gremlin DFE query plans.
- Bug fix for Gremlin queries with an optional traversal, e.g., for queries of the form `g.V().hasLabel('person').group().by(id()).by(__.in('friend').id().fold())`, where no persons without friend edges got grouped.
- Fixed a bug where Gremlin queries containing coalesce steps inside by modulators caused an error to be returned if executed using the DFE engine.
- Fixed a bug that prevented read-only queries running in a Gremlin session from working when connected to a read replica.
- Bug fix where IAM ARN was not present in audit log for a successful initial websocket connection request for Gremlin.
- Coalesce step, identify step coverage with DFE.
- Characteristic set optimization for whole DFE plans.

openCypher fixes

- Bug fixes in openCypher SET clause to allow setting on non-variable expression (ie: match(n:TEST) set(case when n.prop = 2 then n end).prop = 3 return n.prop.
- Bug fix for failing openCypher queries involving aggregation and order by.
- Improved UNWIND of large list containing static maps.
- Bug fix openCypher MERGE query using custom id with duplicate values.

SPARQL fixes

- Fixed a SPARQL bug about the variable scope in optional patterns.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.3.1.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.6.2`
- *Gremlin latest version supported:* `3.6.5`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.3.1.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.3.1.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.3.1.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
--db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
--db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.3.0.0 (2023-11-15)

As of 2023-11-15, engine version 1.3.0.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Note**
>
> Engine release 1.3.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.3.0.0 to engine version 1.3.0.0 or above, you must re-create all of your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.3`. Earlier releases used parameter group family `neptune1`, or `neptune1.2`. and those parameter groups won't work with release 1.3.0.0 and above. Similarly, you should use 1.4.0.0 cluster parameter groups for engine versions 1.4.0.0 and above. See Amazon Neptune parameter groups for more information.

New Features in This Engine Release

- Released the Neptune data API.

  The Amazon Neptune data API provides SDK support for more than 40 of Neptune's data operations, including data loading, query execution, data inquiry, and machine learning. It

supports all three Neptune query languages (Gremlin, openCypher and SPARQL), and is available in all SDK languages. It automatically signs API requests and greatly simplifies integrating Neptune into your applications.

- Added support for integrating OpenSearch Serverless with Neptune.

Improvements in This Engine Release

Improvements to Neptune engine updates

Neptune has changed the way it releases engine updates so that you can have more control over the update process. Instead of releasing patches for non-breaking changes, Neptune now releases minor versions that can be controlled using the `AutoMinorVersionUpgrade` instance field, and about which you can receive notifications by subscribing to the `RDS-EVENT-0156` event.

See Maintaining your Amazon Neptune DB Cluster for more information about these changes.

Encryption in transit improvement

Neptune no longer supports the following cipher suites:

- `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA`
- `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA`

Neptune only supports the following strong cipher suites with TLS 1.2:

- `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`
- `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`
- `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384`
- `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256`

Gremlin Improvements

- Added support in the DFE engine for the following Gremlin steps:
  - `FoldStep`
  - `GroupStep`
  - `GroupCountStep`

  - `TraversalMapStep`
  - `UnfoldStep`
  - `LabelStep`
  - `PropertyKeyStep`
  - `PropertyValueStep`
  - `AndStep`
  - `OrStep`
  - `ConstantStep`
  - `CountGlobalStep`
- Optimized Gremlin DFE query plans to avoid full vertex scans when using `by()` modulation.
- Significantly improved performance of low cardinality and low latency queries.
- Added DFE support for TinkerPop `Or` filter predicates.
- Improved DFE support of traversal for filters on the same key, for queries like the following:

```
g.withSideEffect("Neptune#useDFE", true)
 .V()
 .has('name', 'marko')
 .and(
   or(
     has('name', eq("marko")),
     has('name', eq("vardas"))
   )
 )
```

- Improved error handling for the `fail()` step.

openCypher improvements

- Significantly improved performance of low cardinality and low latency queries.
- Improved query planning performance when query contains many node types.
- Reduced latency of all VLP queries.
- Improved performance by removing redundant pipeline joins for single node pattern queries.
- Improved performance for queries that contain multi-hop patterns with cycles, like this one:

```
MATCH (n)-->()-->()-->(m)
```

```
RETURN n m
```

SPARQL improvements

- Introduced a new SPARQL operator: `PipelineHashIndexJoin`.
- Improved performance of URI validation for SPARQL queries.
- Improved performance of SPARQL full-text search queries by batch resolving dictionary terms.

Defects Fixed in This Engine Release

Gremlin fixes

- Fixed a Gremlin bug where a transaction leak would occur when checking the Gremlin query status endpoint for queries with predicates in child traversals for steps that are not processed natively in the DFE engine.
- Fixed a Gremlin bug where `valueMap()` was not optimized in the DFE engine under `by()` traversals.
- Fixed a Gremlin bug where a step label attached to `UnionStep` was not propagated to the last path element of its child traversals respectively.
- Fixed a Gremlin bug where a query would fail because it contained too many TinkerPop steps and then would not be cleaned up.
- Fixed a Gremlin bug where a `NullPointerException` would be thrown in `mergeV` and `mergeE` steps.
- Fixed a Gremlin bug where `order()` would not properly sort string outputs when some of them contained a space character.
- Fixed a Gremlin correctness issue that occurred when the `valueMap` step was processed in the DFE engine.
- Fixed a Gremlin correctness issue that occurred when `GroupStep` or `GroupCountStep` was nested in a key traversal.

openCypher fixes

- Fixed an openCypher bug involving error handling around NULL characters.
- Fixed a bug in openCypher Bolt transaction handling.

SPARQL fixes

- Fixed a SPARQL bug where values inside recursive functions were not properly resolved.
- Fixed a SPARQL bug that caused performance degredation when a large number of values were injected using the `VALUES` clause.
- Fixed a SPARQL bug where a call to the `REGEX` operator on a language-tagged literal would never succeed.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.3.0.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.6.2`
- *Gremlin latest version supported:* `3.6.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.3.0.0

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.3.0.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.3.0.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.1.2 (2024-08-05)

As of 2024-08-05, engine version 1.2.1.2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Note**
>
> **If upgrading from an engine version earlier than 1.2.0.0:**
>
> - Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier

than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.

- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

  If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.
- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Warning

An issue was detected with the SPARQL 1.1 Graph Store HTTP Protocol (GSP) that may be present under certain conditions when GSP is used with action-based authorization policies. If you are using the SPARQL 1.1 Graph Store HTTP Protocol with action-based authorization policies, we recommend to upgrade to the latest minor Neptune engine version (at least 1.2.1.2) which includes a fix for this issue.

Defects Fixed in This Engine Release

SPARQL fixes

- Fixed an issue with the SPARQL 1.1 Graph Store HTTP Protocol (GSP) that may be present under certain conditions when GSP is used with action-based authorization policies.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.1.2, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.6.2`
- *Gremlin latest version supported:* `3.6.2`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.1.2

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.1.2 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.1.2 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.1.1 (2024-03-11)

As of 2024-03-11, engine version 1.2.1.1 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Note**
>
> **If upgrading from an engine version earlier than 1.2.0.0:**
>
> - Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier

than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.

- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

  If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.
- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Improvements in This Engine Release

General improvements

Neptune has improved the warning shown in profile/explain.

Gremlin Improvements

- Improved DFE statistics computation to avoid very high NCUs of Serverless instance.
- Gremlin performance improvement for WITHIN.

Defects Fixed in This Engine Release

Gremlin fixes

- Bug fixes with ordering of Gremlin DFE engine query plan.
- Bug fix with Gremlin out-of-memory error when originally reported as InternalFailureException.
- Bug fix where IAM ARN was not present in audit log for a successful initial websocket connection request.
- Bug fix for Gremlin queries with TinkerPop session enabled when queries in a session fail even when all of them are read only and connect to a reader instance.

openCypher fixes

- Bug fixes in openCypher SET clause to allow setting on non-variable expression (ie: match(n:TEST) set(case when n.prop = 2 then n end).prop = 3 return n.prop.
- Bug fix for failing openCypher queries involving aggregation and order by.
- Improved UNWIND of large list containing static maps.
- Bug fix openCypher MERGE query using custom id with duplicate values.

SPARQL fixes

- Bug fixes in SPARQL DFE query planner.
- Bug fix for SPARQL when used with BIND and OPTIONAL keywords.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.1.1, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.6.2`
- *Gremlin latest version supported:* `3.6.2`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.1.1

You can upgrade to this release from engine release 1.2.0.0 or above.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.1.1 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.1.1 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.1.0 (2023-03-08)

As of 2023-03-08, engine version 1.2.1.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.

- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Subsequent Patch Releases for This Release

- Release: 1.2.1.0.R2 (2023-05-02)
- Release: 1.2.1.0.R3 (2023-06-13)
- Release: 1.2.1.0.R4 (2023-08-10)
- Release: 1.2.1.0.R5 (2023-09-02)
- Release: 1.2.1.0.R6 (2023-09-12)
- Release: 1.2.1.0.R7 (2023-10-06)

New Features in This Engine Release

- Added support for TinkerPop 3.6.2, which adds many new Gremlin features such as the new `mergeV()`, `mergeE()`, `element()`, and `fail()` steps. The `mergeV()` and `mergeE()` steps are of particular note as they offer a long-awaited declarative option for performing upsert-like operations, which should greatly simplify existing code patterns and make Gremlin easier to read. The 3.6.x version also added regex predicates, a new overload to the `property()` step which takes a `Map`, and a major revision of `by()` modulation behavior that is far more consistent across all steps which use it.

  See the TinkerPop change log and upgrade page for information about the changes in version 3.6 and things to consider when upgrading.

  If you are using `fold().coalesce(unfold(), <mutate>)` for conditional inserts, we recommend that you migrate to the new `mergeV/E()` syntax, described here and here. Neptune

uses a narrower locking pattern for `Merge` than for `Coalesce`, which can reduce concurrent modification exceptions (CMEs).

For more information about the new features available in this TinkerPop release, see Stephen Mallette's blog, Exploring new features of Apache TinkerPop 3.6.x in Amazon Neptune.

- Added support for R6i instance types, powered by 3rd-generation Intel Xeon Scalable processors. These are an ideal fit for memory-intensive workloads and offer up to 15% better compute/price performance and up to 20% higher memory bandwidth per vCPU than comparable R5 instance types.
- Added graph summary API endpoints for both property graphs and RDF graphs, that let you get a fast summary report about your graph.

  For property (PG) graphs, the graph summary API provides a read-only list of node and edge labels and property keys, along with counts of nodes, edges, and properties. For RDF graphs, it provides a list of classes and predicate keys, along with counts of quads, subjects, and predicates.

  The following changes went along with the new graph summary API:

  - Added a new GetGraphSummary dataplane action.
  - Added a new `rdf/statistics` endpoint to replace the `sparql/statistics` endpoint, which is now deprecated.
  - Changed the name of the `summary` field in the statistics status response to `signatureInfo`, so as not to confuse it with graph summary information. Previous engine versions continue to use `summary` in the JSON response.
  - Changed the precision of the `date` field in the statistics status response from minute to millisecond. The previous format was `2020-05-07T23:13Z` (minute precision), while the new format is `2023-01-24T00:47:43.319Z` (millisecond precision). Both are ISO 8601 compliant, but this change may break existing code, depending on how the date is being parsed.
  - Added a new `%statistics` line magic in the Workbench that lets you retrieve DFE engine statistics.
  - Added a new `%summary` line magic in the Workbench that lets you retrieve graph summary information.
- Added slow-query logging to log queries that take longer to execute than a specified threshold. You enable and control slow-query logging using the two new dynamic parameters, namely neptune_enable_slow_query_log, and neptune_slow_query_log_threshold.

- Added support for two dynamic parameters, namely the new cluster parameters, neptune_enable_slow_query_log, and neptune_slow_query_log_threshold. When you make a change to a dynamic parameter, it takes effect immediately, without requiring any instance reboot.
- Added a Neptune-specific openCypher removeKeyFromMap() function that removes a specified key from a map and returns the resulting new map.

Improvements in This Engine Release

- Extended Gremlin DFE support to `limit` steps with local scope.
- Added `by()` modulation support for the Gremlin `DedupGlobalStep` in the DFE engine.
- Added DFE support for Gremlin `SelectStep` and `SelectOneStep`.
- Performance improvements and correctness fixes for various Gremlin operators, including `repeat`, `coalesce`, `store`, and `aggregate`.
- Improved performance of openCypher queries involving `MERGE` and `OPTIONAL MATCH`.
- Improved performance of openCypher queries involving `UNWIND` of a list of maps of literal values.
- Improved performance of openCypher queries that have an `IN` filter for `id`. For example:

```
MATCH (n) WHERE id(n) IN ['1', '2', '3'] RETURN n
```

- Added the ability to specify the base IRI for SPARQL queries using the BASE statement (see Default base IRI for queries and updates).
- Shortened the load processing wait time for Gremlin and openCypher edge-only bulk loads.
- Made bulk loads resume asynchronously when Neptune restarts to avoid a lengthy wait time caused by Amazon S3 connectivity issues before failing resume attempts.
- Improved handling of SPARQL DESCRIBE queries that have the describeMode query hint set to `"CBD"` (concise bounded description) and that involve a large number of blank nodes.

Defects Fixed in This Engine Release

- Fixed an openCypher bug where queries returned the string, `"null"`, instead of a null value in Bolt and SPARQL-JSON.

- Fixed an openCypher bug in list comprehension that produced a null value rather than the values provided for the list elements.
- Fixed an openCypher bug where byte values were not correctly serialized.
- Fixed a Gremlin bug in `UnionStep` that occurred when an input was an edge traversing to a vertex inside a child traversal.
- Fixed a Gremlin bug that caused a step label associated with `UnionStep` not to propagate correctly to the last step of each child traversal.
- Fixed a Gremlin bug for the `dedup` step with labels following a `repeat` step, where the labels attached to the `dedup` step weren't available to use in query further.
- Fixed a Gremlin bug where translating the `repeat` step inside a `union` step failed with an internal error.
- Fixed Gremlin correctness issues for DFE queries with `limit` as a child traversal of non-union steps by falling back to Tinkerpop. Queries in a form like this are affected:

```
g.withSideEffect('Neptune#useDFE', true).V().as("a").select("a").by(out().limit(1))
```

- Fixed a SPARQL bug where `SPARQL GRAPH` patterns would not consider the dataset supplied by a `FROM NAMED` clause.
- Fixed a SPARQL bug where SPARQL `DESCRIBE` with some `FROM` and/or `FROM NAMED` clauses did not always correctly use data from the default graph and sometimes threw an exception. See SPARQL DESCRIBE behavior with respect to the default graph.
- Fixed a SPARQL bug so that the correct exception message is returned when null characters are rejected.
- Fixed a SPARQL explain bug that affected plans containing a PipelinedHashIndexJoin operator.
- Fixed a bug that caused an internal error to be thrown when a query that returns a constant value was submitted.
- Fixed an issue with deadlock detector logic that occasionally made the engine unresponsive.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.1.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.6.2`
- *Gremlin latest version supported:* `3.6.2`

- *openCypher version:* `Neptune-9.0.20190305-1.1`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.1.0

You can manually upgrade to this release from any previous Neptune engine release greater than or equal to 1.1.0.0.

> **Note**
>
> Starting with engine release 1.2.0.0, all custom parameter groups and custom cluster parameter groups that you were using with engine versions earlier than `1.2.0.0` must now be re-created using parameter group family `neptune1.2`. Previous releases used parameter group family `neptune1`, and those parameter groups will not work with releases from `1.2.0.0` onwards. See Amazon Neptune parameter groups for more information.

You will not be automatically upgraded to this major version release.

Upgrading to This Release

Amazon Neptune 1.2.1.0 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
```

```
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.1.0.R7 (2023-10-06)

As of 2023-10-06, engine version 1.2.1.0.R7 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.

- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Defects Fixed in This Engine Release

- Fixed a bug where in some cases a failed transaction was not closed properly.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.1.0.R7, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.6.2`
- *Gremlin latest version supported:* `3.6.2`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrading to This Release

Amazon Neptune 1.2.1.0.R7 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations

on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.1.0.R6 (2023-09-12)

As of 2023-09-12, engine version 1.2.1.0.R6 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Note**
>
> **If upgrading from an engine version earlier than 1.2.0.0:**
>
> - Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group

family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.

- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

  If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.
- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

New Features in This Engine Release

- Released the Neptune data API.

  The Amazon Neptune data API provides SDK support for loading data, running queries, getting information about your data, and running machine-learning operations. It supports the Gremlin and openCypher query languages in Neptune and is available in all SDK languages. It automatically signs API requests and greatly simplifies integrating Neptune into applications.

Defects Fixed in This Engine Release

- Fixed a bug that could cause CPU spikes under high loads when Slow Query logs were enabled.

- Fixed a Gremlin bug where adding an edge and its properties followed by `inV()` or `outV()` raised an `InternalFailureException`.
- Fixed several issues with IAM role chaining that caused degraded bulk-loader performance in some cases.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.1.0.R6, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.6.2`
- *Gremlin latest version supported:* `3.6.2`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrading to This Release

Amazon Neptune 1.2.1.0.R6 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> ```

```
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.1.0.R5 (2023-09-02)

As of 2023-09-02, engine version 1.2.1.0.R5 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.

- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

New Features in This Engine Release

- Released the Neptune data API.

  The Amazon Neptune data API provides SDK support for loading data, running queries, getting information about your data, and running machine-learning operations. It supports the Gremlin and openCypher query languages in Neptune and is available in all SDK languages. It automatically signs API requests and greatly simplifies integrating Neptune into applications.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug where adding an edge and its properties followed by `inV()` or `outV()` raised an `InternalFailureException`.
- Fixed several issues with IAM role chaining that caused degraded bulk-loader performance in some cases.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.1.0.R5, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.6.2`
- *Gremlin latest version supported:* `3.6.2`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrading to This Release

Amazon Neptune 1.2.1.0.R5 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.1.0.R4 (2023-08-10)

As of 2023-08-10, engine version 1.2.1.0.R4 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Important

Changes introduced in this engine release may in some cases cause you to observe degraded bulk load performance. As a result, upgrades to this release have been temporarily suspended until the problem has been resolved.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

  If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.
- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/`

openCypher can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Improvements in This Engine Release

- Add GraphSON-1.0 support for Gremlin. To use GraphSON-1.0, pass `Accept header` with a value of:

```
application/vnd.gremlin-v1.0+json;types=false
```

Defects Fixed in This Engine Release

- Fixed a Gremlin bug where a transaction leak would occur when checking the Gremlin query status endpoint for queries with predicates in child traversals for steps that are not processed natively.
- Fixed an openCypher bug in Bolt transaction handling.
- Fixed a concurrency issue on the storage layer that could cause a crash.
- Fixed a bug in slow query logs to make sure that they're not active when they're disabled.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.1.0.R4, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.6.2`
- *Gremlin latest version supported:* `3.6.5`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.1.0.R4

Upgrading to This Release

Amazon Neptune 1.2.1.0.R4 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.1.0.R3 (2023-06-13)

As of 2023-06-13, engine version 1.2.1.0.R3 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Important**
>
> Changes introduced in this engine release may in some cases cause you to observe degraded bulk load performance. As a result, upgrades to this release have been temporarily suspended until the problem has been resolved.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

  If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.
- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

New Features in This Engine Release

- Added support for cross-account bulk loading using IAM role chaining.

Improvements in This Engine Release

- Improved Gremlin's `fail()` step to differentiate the exception it produced from a generic `InternalFailureException` and to ensure that any user-supplied message provided to it was propagated back to the caller.
- Improved Gremlin query engine optimizations for `store`, `aggregate`, `cap`, `limit`, and `hasLabel`.
- Added support for openCypher trignometric functions:
  - `acos()`
  - `asin()`
  - `atan()`
  - `atan2()`
  - `cos()`
  - `cot()`
  - `degrees()`
  - `pi()`
  - `radians()`
  - `sin()`
  - `tan()`
- Added support for several openCypher aggregating functions:
  - `percentileDisc()`
  - `stDev()`
- Added support for openCypher `epochmillis()` function that converts a `datetime` to `epochmillis`. For example:

```
MATCH (n) RETURN epochMillis(n.someDateTime)
1698972364782
```

- Added support for openCypher modulo (`%`) operator.
- Added support for the openCypher Static Debug Explain tool.
- Added support for the openCypher `randomUUID()` function.
- Improved openCypher performance:
  - Improved the parser and query planner.

- Improved CPU utilization in the DFE engine.
- Improved the performance of queries containing multiple update clauses that reuse the same variables. Examples are:

```
MERGE (n {name: 'John'})
  or
MERGE (m {name: 'Jim'})
  or
MERGE (n)-[:knows {since: 2023}]#(m)
```

- Optimized query plans for multi-hop query patterns such as:

```
MATCH (n)-->()-->()-->(m)
RETURN n m
```

- Improved the performance of list and map injection through parameterized queries. For example:

```
UNWIND $idList as id MATCH (n {`~id`: id})
RETURN n.name
```

- Improved query execution containing `WITH` by making it an appropriate barrier.
- Optimized to avoid redundant materialization of values in `Unfold` and aggregation functions.

- Improved the performance of SPARQL queries that contain a large number of static inputs in the `VALUES` clause, such as:

```
SELECT ?n WHERE { VALUES (?name) { ("John") ("Jim") ... many values ... } ?n a ?
n_type . ?n ?name . }
```

- Improved SPARQL CBD query performance.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug where long queries with deep nesting were causing high CPU usage and query timeouts during the query planning phase.
- Fixed a Gremlin bug where an invalid `NullPointerException` could be thrown when using `mergeV` or `mergeE`.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.1.0.R3, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.6.2`
- *Gremlin latest version supported:* `3.6.2`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.1.0.R3

Upgrading to This Release

Amazon Neptune 1.2.1.0.R3 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

Note

If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.1.0.R2 (2023-05-02)

As of 2023-05-02, engine version 1.2.1.0.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

  If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.

- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Improvements in This Engine Release

- Added an `enableInterContainerTrafficEncryption` parameter to all Neptune ML APIs, that you can use to enable and disable inter-container traffic encryption in training or hyper-parameter tuning jobs.
- Added multi-label support for Gremlin `mergeV()` and `mergeE()`.

Defects Fixed in This Engine Release

- Fixed an openCypher bug where update and return queries did not handle `orderBy`, `limit`, or `skip` properly.
- Fixed an openCypher bug that allowed parameters contained in one request to be overridden by parameters contained in another simultaneous request.
- Fixed an openCypher bug where slow query logs did not contain correct query times.
- Fixed a Gremlin bug where a transaction leak could occur when a query containing `GroupCountStep` was submitted as a string.
- Fixed a Gremlin bug where WebSocket queries were failing when slow query logs were enabled.
- Fixed a Gremlin bug where storage-counter debug logs were missing in the slow query logs for WebSocket requests.
- Fixed several Gremlin bugs involving `mergeV()` and `mergeE()`.
- Fixed a SPARQL bug where named graph query costs were mis-estimated, leading to sub-optimal query plans and out-of-memory errors.
- Fixed a bug that affected authorization for Gremlin and openCypher queries on an IAM-enabled cluster.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.1.0.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.6.2`
- *Gremlin latest version supported:* `3.6.2`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.1.0.R2

Upgrading to This Release

Amazon Neptune 1.2.1.0.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.2 (2022-11-20)

As of 2022-11-20, engine version 1.2.0.2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.

- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Subsequent Patch Releases for This Release

- Release: 1.2.0.2.R2 (2022-12-15)
- Release: 1.2.0.2.R3 (2023-03-27)
- Release: 1.2.0.2.R4 (2023-05-08)
- Release: 1.2.0.2.R5 (2023-08-16)
- Release: 1.2.0.2.R6 (2023-09-12)

New Features in This Engine Release

- Introduced real-time inductive inference for Gremlin in Neptune ML.
- Introduced an openCypher extension that supports specifying custom ID values for entities instead of the UUIDs that Neptune otherwise generates. The ability to assign custom IDs makes it easier to migrate to Neptune from Neo4j.

> **Warning**
>
> This extension to the openCypher specification is not backward compatible, because `~id` is now considered a reserved property name. If you are already using `~id` as a property in your data and queries, you must migrate the `~id` property to a new property key before upgrading to this release.

- Added several new SPARQL `DESCRIBE` modes along with query hints to configure them.

Improvements in This Engine Release

- Improved openCypher performance, particularly for VLP queries.
- Improved DFE performance for Gremlin queries with non-terminal limits, such as:

```
g.withSideEffect('Neptune#useDFE',true).V().hasLabel('Student').limit(5).out('takesCourse')
```

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.2, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.2

Upgrading to This Release

Amazon Neptune 1.2.0.2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.2 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.2.R6 (2023-09-12)

As of 2023-09-12, engine version 1.2.0.2.R6 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.

- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Defects Fixed in This Engine Release

- Fixed a SPARQL bug where the `REGEX` operator would never succeed when called on a language-tagged literal.
- Fixed an issue that caused bulk-load performance to be degraded.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.2.R6, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.5`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.2.R6

Your Neptune DB cluster will be upgraded to this maintenance patch release automatically during your next maintenance window if you are running engine version `1.2.0.2`.

Upgrading to This Release

Amazon Neptune 1.2.0.2.R6 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.2 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.2.R5 (2023-08-16)

As of 2023-08-16, engine version 1.2.0.2.R5 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Important

Changes introduced in this engine release may in some cases cause you to observe degraded bulk load performance. As a result, upgrades to this release have been temporarily suspended until the problem has been resolved.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

  If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.
- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/`

openCypher can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug where `order()` would not properly sort string outputs when some of them contained a space character.
- Fixed a Gremlin bug where a transaction leak would occur when checking the Gremlin query status endpoint for queries with predicates in child traversals for steps that are not processed natively.
- Fixed an openCypher bug in Bolt transaction handling.
- Fixed a concurrency issue on the storage layer that could cause a crash.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.2.R5, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.5`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.2.R5

Your Neptune DB cluster will be upgraded to this maintenance patch release automatically during your next maintenance window if you are running engine version `1.2.0.2`.

Upgrading to This Release

Amazon Neptune 1.2.0.2.R5 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.2 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.2.R4 (2023-05-08)

As of 2023-05-08, engine version 1.2.0.2.R4 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Note**
>
> **If upgrading from an engine version earlier than 1.2.0.0:**
>
> - Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group

family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.

- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

  If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.
- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Defects Fixed in This Engine Release

- Fixed a SPARQL bug where a large number of values injected through the `VALUES` clause could lead to performance degradation.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.2.R4, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.6`

- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.2.R4

Your Neptune DB cluster will be upgraded to this maintenance patch release automatically during your next maintenance window if you are running engine version `1.2.0.2`.

Upgrading to This Release

Amazon Neptune 1.2.0.2.R4 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.2 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.2.R3 (2023-03-27)

As of 2023-03-27, engine version 1.2.0.2.R3 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Note**
>
> **If upgrading from an engine version earlier than 1.2.0.0:**
>
> - Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
> - Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.
>
> You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.
>
> If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.
> - Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/`

openCypher can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Improvements in This Engine Release

- For serverless DB clusters, changed the minimum capacity setting to 1.0 NCU, and the lowest valid maximum setting to 2.5 NCUs. See Capacity scaling in a Neptune Serverless DB cluster
- Added an `enableInterContainerTrafficEncryption` parameter to all Neptune ML APIs, that you can use to enable and disable inter-container traffic encryption in training or hyper-parameter tuning jobs.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug where `option(Predicate)` was not being recognized as valid Gremlin syntax.
- Fixed a Gremlin bug that caused queries not to be properly cleaned up if they failed because they contained too many steps.
- Fixed a Gremlin correctness issue that affected DFE queries with `limit` as child traversal of non-union steps by falling back to Tinkerpop. Here is an example of such a query:

```
g.withSideEffect('Neptune#useDFE', true).V().as("a").select("a").by(out().limit(1))
```

- Fixed a potential Gremlin transaction leak when a query submitted as a String contains `GroupCountStep`.
- Fixed an openCypher bug where the type of parameter value was not correctly inferred in a list or list of maps.
- Fixed an openCypher bug where update and return queries did not handle `orderBy`, `limit`, or `skip` properly.
- Fixed an openCypher bug that allowed parameters contained in one request to be overridden by parameters contained in another simultaneous request.
- Fixed a SPARQL bug where a large number of values injected in a `VALUES` clause could lead to performance degradation.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.2.R3, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.6`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.2.R3

Your Neptune DB cluster will be upgraded to this maintenance patch release automatically during your next maintenance window if you are running engine version `1.2.0.2`.

Upgrading to This Release

Amazon Neptune 1.2.0.2.R3 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.2 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> ```

```
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.2.R2 (2022-12-15)

As of 2022-12-15, engine version 1.2.0.2.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.

- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Improvements in This Engine Release

- Improved performance of openCypher queries involving `MERGE` and `OPTIONAL MATCH`.
- Improved performance of openCypher queries involving `UNWIND` of a list of maps of literal values.
- Improved performance of openCypher queries that have an `IN` filter for `id`. For example:

```
MATCH (n) WHERE id(n) IN ['1', '2', '3'] RETURN n
```

- Performance improvements and correctness fixes for various Gremlin operators, including `repeat`, `coalesce`, `store`, and `aggregate`.

Defects Fixed in This Engine Release

- Fixed an openCypher bug where queries returned the string, `"null"`, instead of a null value in Bolt and SPARQL-JSON.
- Fixed a Gremlin bug that caused a step label attached to `UnionStep` not to be propagated to the last path element of its child traversals.
- Fixed a Gremlin bug that caused `valueMap()` not to be optimized under a `by()` traversal in the DFE engine.
- Fixed a Gremlin bug where read queries executed as part of a longer Gremlin transaction would not lock the rows.
- Fixed an audit log bug that caused unnecessary information to be logged and certain fields to be missing from the logs.

- Fixed an audit log bug where the IAM ARN of HTTP requests to an IAM-enabled DB cluster were not recorded.
- Fixed a lookup-cache bug so as to cap the incremental memory used for writes to the cache.
- Fixed a lookup-cache bug that involved setting read-only mode for the lookup cache when writes failed.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.2.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.2.R2

Your Neptune DB cluster will be upgraded to this maintenance patch release automatically during your next maintenance window if you are running engine version `1.2.0.2`.

Upgrading to This Release

Amazon Neptune 1.2.0.2.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.2 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.1 (2022-10-26)

As of 2022-10-26, engine version 1.2.0.1 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.

- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Subsequent Patch Releases for This Release

- Maintenance Release: 1.2.0.1.R2 (2022-12-13)
- Maintenance Release: 1.2.0.1.R3 (2023-09-27)

New Features in This Engine Release

- Introduced Amazon Neptune Serverless, an on-demand autoscaling configuration that scales your DB cluster up to meet increases in processing demand and then down again when the demand decreases.

Improvements in This Engine Release

- Improved performance of Gremlin `order-by` queries. Gremlin queries with an `order-by` at the end of a `NeptuneGraphQueryStep` now use a larger chunk size for better performance. This does not apply to `order-by` on an internal (non-root) node of the query plan.
- Improved performance of Gremlin update queries. Vertices and edges must now be locked against deletion while adding edges or properties. This change eliminates duplicate locks within a transaction, which improves performance.

- Improved performance of Gremlin queries that use `dedup()` inside of a `repeat()` subquery by pushing the `dedup` down to the native execution layer.
- Added user-friendly error messages for IAM authentication errors. These messages now show the your IAM user or role ARN, the resource ARN, and a list of unauthorized actions for the request. The list of unauthorized actions helps you see what might be missing or explicitly denied in the IAM policy that you're using.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug where using `PartitionStrategy` after upgrading to TinkerPop 3.5 incorrectly resulted an error with the message, "PartitionStrategy does not work with anonymous Traversals," which prevented the traversal from being executed.
- Fixed a Gremlin correctness bug involving `WherePredicateStep` translation, where Neptune's query engine was producing incorrect results for queries using `where(P.neq('x'))` and variations of that.
- Fixed an openCypher bug in the `MERGE` clause that in some cases caused duplicate node and edge creation.
- Fixed a SPARQL bug in the handling of queries that contain `(NOT) EXISTS` within an `OPTIONAL` clause, where in some cases query results were missing.
- Fixed a bulk loader bug that caused performance regressions under heavy insertion loads.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.1, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.1

Upgrading to This Release

Amazon Neptune 1.2.0.1 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.1.R3 (2023-09-27)

As of 2023-09-27, engine version 1.2.0.1.R3 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

  If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.
- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Improvements in This Engine Release

- Added an `enableInterContainerTrafficEncryption` parameter to all Neptune ML APIs, that you can use to enable and disable inter-container traffic encryption in training or hyper-parameter tuning jobs.
- For serverless DB clusters, changed the minimum capacity setting to 1.0 NCU, and the lowest valid maximum setting to 2.5 NCUs. See Capacity scaling in a Neptune Serverless DB cluster *(((before release, this change needs to be reflected in the serverless page too)))*.

Defects Fixed in This Engine Release

- Fixed an openCypher bug where update-and-return queries did not handle `orderBy`, `limit`, or `skip` properly.
- Fixed an openCypher bug that allowed parameters contained in one request to be overridden by parameters contained in another simultaneous request.
- Fixed an openCypher bug in Bolt transaction handling.
- Fixed Gremlin correctness issues for DFE queries with `limit` as a child traversal of non-union steps by falling back to Tinkerpop. For example, for queries like this:

```
g.withSideEffect('Neptune#useDFE', true)
 .V()
 .as("a")
 .select("a")
 .by(out()
 .limit(1))
```

- Fixed a Gremlin bug where a query would fail because it contained too many TinkerPop steps and then would not be cleaned up.
- Fixed a Gremlin bug where `order()` would not properly sort string outputs when some of them contained a space character.
- Fixed a Gremlin bug where a transaction leak could occur when a query was submitted as a String and contained `GroupCountStep`.
- Fixed a Gremlin bug where a transaction leak would occur when checking the Gremlin query status endpoint for queries with predicates in child traversals for steps that are not processed natively.

- Fixed a Gremlin bug where adding an Edge and its properties followed by `inV()` or `outV()` caused an `InternalFailureException`.
- Fixed a concurrency issue in the storage layer.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.1.R3, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.6`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.1.R3

Your Neptune DB cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version 1.2.0.1

Upgrading to This Release

Amazon Neptune 1.2.0.1.R3 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
```

```
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.1.R2 (2022-12-13)

As of 2022-12-13, engine version 1.2.0.1.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.

- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Improvements in This Engine Release

- Improved the performance of openCypher queries that involve `UNWIND` on a list of maps of literal values.
- Performance improvements and correctness fixes for various Gremlin operators, including `repeat`, `coalesce`, `store`, and `aggregate`.

Defects Fixed in This Engine Release

- Fixed an openCypher bug where queries returned the string, `"null"`, instead of a null value in Bolt and SPARQL-JSON.
- Fixed an audit log bug that caused unnecessary information to be logged and certain fields to be missing from the logs.
- Fixed a lookup-cache bug so as to cap the incremental memory used for writes to the cache.
- Fixed a lookup-cache bug that involved setting read-only mode for the lookup cache when writes failed.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.1.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.1.R2

Your Neptune DB cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version 1.2.0.1

Upgrading to This Release

Amazon Neptune 1.2.0.1.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> ```

```
proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.0 (2022-07-21)

As of 2022-07-21, engine version 1.2.0.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.

- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Subsequent Patch Releases for This Release

- Release: 1.2.0.0.R2 (2022-10-14)
- Release: 1.2.0.0.R3 (2022-12-15)
- Release: 1.2.0.0.R4 (2023-09-29)

New Features in This Engine Release

- Added support for more granular access control in Neptune IAM policies than has been available previously, based on data plane actions. This is a breaking change in that existing IAM policies that are based on the deprecated `connect` action must be adjusted to use the more granular data plane actions. See Types of IAM policies.
- Improved reader instance availability. Previously, when a writer instance restarted, all reader instances in the Neptune cluster automatically restarted too. Starting with engine release 1.2.0.0, reader instances remain active after a writer restart, which improves reader availability. Reader instances can be restarted separately to pick up parameter group changes. See Rebooting a DB instance in Amazon Neptune.
- Added a new neptune_streams_expiry_days DB cluster parameter which lets you set the number of days that stream records are kept on the server before being deleted. The range is 1 through 90, and the default is 7.

Improvements in This Engine Release

- Improved Gremlin serialization performance for ByteCode queries.

- Neptune now processes text predicates using the DFE engine, for improved performance.
- Neptune now processes Gremlin `limit()` steps using the DFE engine, including non-terminal and child traversal limits.
- Changed DFE handling of the Gremlin `union()` step to work with other new features, which means that reference nodes show up in query profiles as expected.
- Improved performance by up to a factor of 5 of some expensive join operations within DFE by parallelizing them.
- Added `by()` modulation support for `OrderGlobalStep order(global)` for the Gremlin DFE engine.
- Added display of injected static values in explain details for DFE.
- Improved performance when pruning duplicate patterns.
- Added order preservation support in the Gremlin DFE engine.
- Improved the performance of Gremlin queries having empty filters, such as these:

```
g.V().hasId(P.within([]))
```

```
g.V().hasId([])
```

- Improved error messaging when a SPARQL query uses a numeric value that is too large for Neptune to represent internally.
- Improved performance for dropping vertices with associated edges by reducing index searches when streams are disabled.
- Extended DFE support to more variants of the `has()` step, in particular to `hasKey()`, `hasLabel()`, and to range predicates for strings/URIs within `has()`. This affects queries such as the following:

```
// hasKey() on properties
g.V().properties().hasKey("name")
g.V().properties().has(T.key, TextP.startingWith("a"))
g.E().properties().hasKey("weight")
g.E().properties().hasKey(TextP.containing("t"))

// hasLabel() on vertex properties
g.V().properties().hasLabel("name")

// range predicates on ID and Label fields
```

```
g.V().has(T.label, gt("person"))
g.E().has(T.id, lte("(an ID value)"))
```

- Added a Neptune-specific openCypher `join()` function that concatenates strings in a list into a single string.
- Updated the Neptune managed policies to include data-access permissions and permissions for the new global database APIs.

Defects Fixed in This Engine Release

- Fixed a bug where an HTTP request with no content-type specified would automatically fail.
- Fixed a SPARQL bug in the query optimizer that prevented use of a service call inside a query.
- Fixed a SPARQL bug in the Turtle RDF parser where a particular combination of Unicode data caused failure.
- Fixed a SPARQL bug where a particular combination of `GRAPH` and`SELECT` clauses produced incorrect query results.
- Fixed a Gremlin bug that caused a correctness issue for queries that used any filter step within a union step, such as the following:

```
g.V("1").union(hasLabel("person"), out())
```

- Fixed a Gremlin bug where `count()` of `both().simplePath()` would result in double the actual number of results returned without `count()`.
- Fixed an openCypher bug where a faulty signature mismatch exception was generated by the server for Bolt requests to clusters with IAM authentication enabled.
- Fixed an openCypher bug where a query using HTTP keep-alive could be incorrectly closed if it was submitted after a failed request.
- Fixed an openCypher bug that could cause an internal error to be thrown when a query that returns a constant value is submitted.
- Fixed a bug in the explain details so that DFE subquery `Time(ms)` now correctly sums the CPU times of operators within the DFE subquery. Consider the following excerpt of explain output as an example:

```
subQuery1
##########################################################################################
```

```
# ID # Out #1 # Out #2 # Name                  # Arguments                        #
 Mode # Units In # Units Out # Ratio # Time (ms) #
##################################################################################
  ...
##################################################################################
# 1  # 2      # -      # DFEChunkLocalSubQuery # subQuery=...graph#336e.../graph_1 #
 -    # 1        # 1         # 1.00  # 0.38      #
#    #        #        #                       # coordinationTime(ms)=0.026        #
     #          #           #       #           #
##################################################################################
  ...
subQuery=...graph#336e.../graph_1
##################################################################################
# ID # Out #1 # Out #2 # Name                  # Arguments                        #
 Mode # Units In # Units Out # Ratio # Time (ms) #
##################################################################################
# 0  # 1      # -      # DFESolutionInjection  # solutions=[?100 -> [-10^^<LONG>]] #
 -    # 0        # 1         # 0.00  # 0.04      #
#    #        #        #                       # outSchema=[?100]                 #
     #          #           #       #           #
##################################################################################
# 1  # 3      # -      # DFERelationalJoin     # joinVars=[]                      #
 -    # 2        # 1         # 0.50  # 0.29      #
##################################################################################
# 2  # 1      # -      # DFESolutionInjection  # outSchema=[]                    #
 -    # 0        # 1         # 0.00  # 0.01      #
##################################################################################
# 3  # -      # -      # DFEDrain              # -                                #
 -    # 1        # 0         # 0.00  # 0.02      #
##################################################################################
```

The subQuery times in the last column of the lower table add up to 0.36 ms (`.04 + .29 + .01 + .02 = .36`). When you add in to the coordination time for that subquery (`.36 + .026 = .386`), you get a result that is close to the time for the subQuery recorded in the last column of the upper table, namely `0.38` ms.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.0

Because this is a major engine release, there is no automatic upgrade to it.

You can only upgrade to release `1.2.0.0` manually, from the latest patch release of engine release 1.1.1.0. Earlier engine releases must first be upgraded to the latest release of `1.1.1.0` before they can be upgraded to `1.2.0.0`.

Therefore, before you try to upgrade to this release, please confirm that you are currently running the latest patch release of release `1.1.1.0`. If you are not, start by upgrading to the latest patch release of `1.1.1.0`.

Before upgrading, you must also re-create any custom DB cluster parameter group that you have been using with your previous version, using parameter group family `neptune1.2`. See Amazon Neptune parameter groups for more information.

If you are upgrading first to release `1.1.1.0` and then immediately to `1.2.0.0`, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete (see Maintaining your Amazon Neptune DB Cluster).

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations

on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.0.R4 (2023-09-29)

As of 2023-09-29, engine version 1.2.0.0.R4 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Note**
>
> **If upgrading from an engine version earlier than 1.2.0.0:**
>
> - Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
> - Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.
>
> You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.
>
> If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.
> - Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Improvements in This Engine Release

- Added an `enableInterContainerTrafficEncryption` parameter to all Neptune ML APIs, that you can use to enable and disable inter-container traffic encryption in training or hyper-parameter tuning jobs.
- For serverless DB clusters, changed the minimum capacity setting to 1.0 NCU, and the lowest valid maximum setting to 2.5 NCUs. See Capacity scaling in a Neptune Serverless DB cluster *(((before release, this change needs to be reflected in the serverless page too)))*.

Defects Fixed in This Engine Release

- Fixed an openCypher bug where update-and-return queries did not handle `orderBy`, `limit`, or `skip` properly.
- Fixed an openCypher bug that allowed parameters contained in one request to be overridden by parameters contained in another simultaneous request.
- Fixed an openCypher bug in Bolt transaction handling.
- Fixed Gremlin correctness issues for DFE queries with `limit` as a child traversal of non-union steps by falling back to Tinkerpop. For example, for queries like this:

```
g.withSideEffect('Neptune#useDFE', true)
 .V()
 .as("a")
 .select("a")
 .by(out()
 .limit(1))
```

- Fixed a Gremlin bug where a query would fail because it contained too many TinkerPop steps and then would not be cleaned up.
- Fixed a Gremlin bug where `order()` would not properly sort string outputs when some of them contained a space character.
- Fixed a Gremlin bug where a transaction leak could occur when a query was submitted as a String and contained `GroupCountStep`.
- Fixed a Gremlin bug where a transaction leak would occur when checking the Gremlin query status endpoint for queries with predicates in child traversals for steps that are not processed natively.

- Fixed a Gremlin bug where adding an Edge and its properties followed by `inV()` or `outV()` caused an `InternalFailureException`.
- Fixed a concurrency issue in the storage layer.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.0.R4, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.6`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.0.R4

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.2.0.0`.

You can only upgrade to release `1.2.0.0` manually from the latest patch release of engine release 1.1.1.0. Earlier engine releases must first be upgraded to the latest release of `1.1.1.0` before they can be upgraded to `1.2.0.0`.

If you are upgrading first to release `1.1.1.0` and then immediately to `1.2.0.0`, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.0.R3 (2022-12-15)

As of 2022-12-15, engine version 1.2.0.0.R3 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

  If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.

- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Improvements in This Engine Release

- Improved performance of openCypher queries involving `MERGE` and `OPTIONAL MATCH`.
- Improved the performance of openCypher queries that involve `UNWIND` on a list of maps of literal values.
- Improved performance of openCypher queries that have an `IN` filter for `id`. For example:

```
MATCH (n) WHERE id(n) IN ['1', '2', '3'] RETURN n
```

- Performance improvements and correctness fixes for various Gremlin operators, including `repeat`, `coalesce`, `store`, and `aggregate`.

Defects Fixed in This Engine Release

- Fixed an openCypher bug where queries returned the string, `"null"`, instead of a null value in Bolt and SPARQL-JSON.
- Fixed an openCypher bug so as to be able to interpret the parameter type correctly when the value is a list or a list of maps.
- Fixed an audit log bug that caused unnecessary information to be logged and certain fields to be missing from the logs.
- Fixed an audit log bug where the IAM ARN of HTTP requests to an IAM-enabled DB cluster were not recorded.
- Fixed a lookup-cache bug so as to cap the incremental memory used for writes to the cache.
- Fixed a lookup-cache bug that involved setting read-only mode for the lookup cache when writes failed.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.0.R3, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.0.R3

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.2.0.0`.

You can only upgrade to release `1.2.0.0` manually from the latest patch release of engine release 1.1.1.0. Earlier engine releases must first be upgraded to the latest release of `1.1.1.0` before they can be upgraded to `1.2.0.0`.

If you are upgrading first to release `1.1.1.0` and then immediately to `1.2.0.0`, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.2.0.0.R2 (2022-10-14)

As of 2022-10-14, engine version 1.2.0.0.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Note

If upgrading from an engine version earlier than 1.2.0.0:

- Engine release 1.2.0.0 introduced a new format for custom parameter groups and custom cluster parameter groups. As a result, if you are upgrading from an engine version earlier than 1.2.0.0 to engine version 1.2.0.0 or above, you must re-create all your existing custom parameter groups and custom cluster parameter groups using parameter group family `neptune1.2`. Earlier releases used parameter group family `neptune1`, and those parameter groups won't work with release 1.2.0.0 and above. See Amazon Neptune parameter groups for more information.
- Engine release 1.2.0.0 also introduced a new format for undo logs. As a result, any undo logs created by an earlier engine version must be purged and the `UndoLogsListSize` CloudWatch metric must fall to zero before any upgrade from a version earlier than 1.2.0.0 can get started. If there are too many undo log records (200,000 or more) when you try to start an update, the upgrade attempt can time out while waiting for purging of the undo logs to complete.

  You can speed up the purge rate by upgrading the cluster's writer instance, which is where the purging occurs. Doing that before trying to upgrade can bring down the number of undo logs before you start. Increasing the size of the writer to a 24XL instance type can increase your purge rate to more than a million records per hour.

  If the `UndoLogsListSize` CloudWatch metric is extremely large, opening a support case may help you explore additional strategies for bringing it down.
- Finally, there was a breaking change in release 1.2.0.0 affecting earlier code that used the Bolt protocol with IAM authentication. Starting with release 1.2.0.0, Bolt needs a resource path for IAM signing. In Java, setting the resource path might look like this: `request.setResourcePath("/openCypher"));`. In other languages, the `/openCypher` can be appended to the endpoint URI. See Using the Bolt protocol for examples.

Improvements in This Engine Release

- Improved performance of Gremlin `order-by` queries. Gremlin queries with an `order-by` at the end of a `NeptuneGraphQueryStep` now use a larger chunk size for better performance. This does not apply to `order-by` on an internal (non-root) node of the query plan.
- Improved performance of Gremlin update queries. Vertices and edges must now be locked against deletion while adding edges or properties. This change eliminates duplicate locks within a transaction, which improves performance.
- Improved performance of Gremlin queries that use `dedup()` inside of a `repeat()` subquery by pushing the `dedup` down to the native execution layer.
- Added the Gremlin `Neptune#cardinalityEstimates` query hint. When set to `false`, this disables cardinality estimates.
- Added user-friendly error messages for IAM authentication errors. These messages now show the your IAM user or role ARN, the resource ARN, and a list of unauthorized actions for the request. The list of unauthorized actions helps you see what might be missing or explicitly denied in the IAM policy that you're using.

Defects Fixed in This Engine Release

- Fixed a Gremlin correctness bug involving `WherePredicateStep` translation, where Neptune's query engine was producing incorrect results for queries using `where(P.neq('x'))` and variations of that.
- Fixed a Gremlin bug where using `PartitionStrategy` after upgrading to TinkerPop 3.5 incorrectly resulted an error with the message, "PartitionStrategy does not work with anonymous Traversals," which prevented the traversal from being executed.
- Fixed various Gremlin bugs related to the `joinTime` of a final join and to statistics inside of `Project.ASK` subgroups.
- Fixed an openCypher bug in the `MERGE` clause that in some cases caused duplicate node and edge creation.
- Fixed a transaction bug where a session could insert graph data and commit even when the corresponding concurrent dictionary inserts got rolled back.
- Fixed a bulk loader bug that caused performance regressions under heavy insertion loads.
- Fixed a SPARQL bug in the handling of queries that contain `(NOT) EXISTS` within an `OPTIONAL` clause, where in some cases query results were missing.

- Fixed a bug where drivers could appear to hang in cases where requests were cancelled due to a timeout prior to their start of evaluation. It was possible to get into this state if all query processing threads on the server were consumed while timeouts occurred to items in the request queue. Because the timeouts from the request queue were not immediately sending messages, the responses appeared to the client to remain pending.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.2.0.0.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.2.0.0.R2

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.2.0.0`.

You can only upgrade to release `1.2.0.0` manually from the latest patch release of engine release 1.1.1.0. Earlier engine releases must first be upgraded to the latest release of `1.1.1.0` before they can be upgraded to `1.2.0.0`.

If you are upgrading first to release `1.1.1.0` and then immediately to `1.2.0.0`, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.2.0.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.2.0.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.1.1.0 (2022-04-19)

As of 2022-04-19, engine version 1.1.1.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Important

Upgrading to this engine release from a version earlier than `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.

In order to complete the upgrade successfully, each subnet in every availability zone (AZ) must have at least one IP address available per Neptune instance. For example, if there is one writer instance and two reader instances in subnet 1, and two reader instances in subnet 2, subnet 1 must have at least 3 IP addresses free and subnet 2 must have at least 2 IP addresses free before starting the upgrade.

At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.

When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for a number of minutes. You may resume write workloads after the upgrade completes.

This process generates the following events:

- Per-cluster event messages:

    - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-(autogenerated snapshot ID)]`
    - `Database cluster major version has been upgraded`
  - Per-instance event messages:
    - `Applying off-line patches to DB instance`
    - `DB instance shutdown`
    - `Finished applying off-line patches to DB instance`
    - `DB instance restarted`

Subsequent Patch Releases for This Release

- Maintenance release: 1.1.1.0.R2 (2022-05-16)
- Release: 1.1.1.0.R3 (2022-06-07)
- Release: 1.1.1.0.R4 (2022-06-23)
- Release: 1.1.1.0.R5 (2022-07-21)
- Release: 1.1.1.0.R6 (2022-09-23)
- Release: 1.1.1.0.R7 (2023-01-23)

New Features in This Engine Release

- The openCypher query language is now generally available for production use.

> **Warning**
>
> There is a breaking change in this release for code that uses openCypher with IAM authentication. In the Neptune preview for openCypher, the host string in the IAM signature included the protocol, such as `bolt://`, like this:
>
> ```
> "Host":"bolt://(host URL):(port)"
> ```
>
> Starting with this engine release, the protocol must be omitted:
>
> ```
> "Host":"(host URL):(port)"
> ```

See Using the Bolt protocol for examples.

- Added support for TinkerPop `3.5.2`. Among the changes in this release are support for remote transactions and bytecode support for sessions (using `g.tx`), and the addition of the `datetime()` function to the Gremlin language.

> **Warning**
>
> There are several breaking changes introduced in TinkerPop 3.5.0, 3.5.1, and 3.5.2 which may affect your Gremlin code. For example, using traversals spawned by a GraphTraversalSource as children like this will no longer work: `g.V().union(identity(), g.V())`.
> Now instead, use an anonymous traversal like this: `g.V().union(identity(), __.V())`.

- Added support for AWS global condition keys that you can use in IAM data-access policies that control access to data stored in Neptune a Neptune DB cluster.
- The Neptune DFE query engine is now generally available for production use with the openCypher query language, but not yet for Gremlin and SPARQL queries. You now enable it using its own neptune_dfe_query_engine instance parameter rather than the lab-mode parameter.

Improvements in This Engine Release

- Added new features to openCypher such as parameterized query support, abstract syntax tree (AST) caching for parameterized queries, variable length path (VLP) improvements, and new operators and clauses. See openCypher specification compliance in Amazon Neptune for the current level of language support.
- Made significant performance improvements to openCypher for simple read and write workloads, resulting in higher throughput when compared to Release 1.1.0.0.
- Removed openCypher bi-directional and depth limitations handling variable-length paths.
- Completed support in the DFE engine for Gremlin `within` and `without` predicates, including cases where they are combined with other predicate operators. For example:

```
g.V().has('age', within(12, 15, 18).or(gt(30)))
```

- Extended support in the DFE engine for the Gremlin `order` step when the scope is global (that is, not `order(local)`), and when `by()` modulators are not used. For example, this query would now have DFE support:

```
g.V().values("age").order()
```

- Added an `isLastOp` field to the Neptune streams change-log response format, to indicate that a record is the last operation in its transaction.
- Significantly improved the performance of audit logging and reduced latency when audit logging is enabled.
- Converted Gremlin WebSocket bytecode and HTTP queries into a user-readable format in audit logs. Queries can now be directly copied from the audit logs to be executed in Neptune notebooks and elsewhere. Note that this change to the current audit log format constitutes a breaking change.

Defects Fixed in This Engine Release

- Fixed a rare Gremlin bug where no results were returned when using nested `filter()` and `count()` steps in combination, such as in the following query:

```
g.V("1").filter(out("knows")
        .filter(in("knows")
        .hasId("notExists"))
        .count())
```

- Fixed a Gremlin bug where an error was returned when using a vertex stored by an aggregate step in either `to()` or `from()` traversals in conjunction with an `addE` step. An example of such a query is:

```
g.V("id").aggregate("v").out().addE().to(select("v").unfold()))
```

- Fixed a Gremlin bug where the `not` step was failing in edge cases when using the DFE engine. For example:

```
g.V().not(V())
```

- Fixed a Gremlin bug where `sideEffect` values were not available within `to()` and `from()` traversals.

- Fixed a bug that occasionally caused a fast reset to trigger an instance failover.
- Fixed a bulk loader bug where a failed transaction would not be closed before beginning the next load job.
- Fixed a bulk loader bug where a low memory condition could cause a crash of the system.
- Added a retry to fix a bulk loader bug where the loader did not wait long enough for IAM credentials to become availble after a failover.
- Fixed a bug where the internal credential cache was not being cleared properly for non-query endpoints such as the `status` endpoint.
- Fixed a streams bug to ensure that stream commit sequence numbers are properly ordered.
- Fixed a bug where long-running connections were terminated sooner than ten days on IAM-enabled clusters.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.1.1.0, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.1.1.0

You can manually upgrade any previous Neptune engine release to this release. Note that versions prior to the major version engine (1.1.0.0) will take longer to upgrade to this release.

You will not be automatically upgraded to this release.

Upgrading to This Release

> **Important**
>
> **Upgrading to this engine release from any version prior to `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write**

requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.

At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.

When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for around 6 minutes. You may resume write workloads after the upgrade completes.

This process generates the following events:

- Per-cluster event messages:
  - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-`*`(autogenerated snapshot ID)`*`]`
  - `Database cluster major version has been upgraded`
- Per-instance event messages:
  - `Applying off-line patches to DB instance`
  - `DB instance shutdown`
  - `Finished applying off-line patches to DB instance`
  - `DB instance restarted`

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine neptune \
    --engine-version 1.1.1.0 \
    --allow-major-version-upgrade \
```

```
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine neptune ^
    --engine-version 1.1.1.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let a previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.1.1.0.R7 (2023-01-23)

As of 2023-01-23, engine version 1.1.1.0.R7 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Important

Upgrading to this engine release from a version earlier than `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.

In order to complete the upgrade successfully, each subnet in every availability zone (AZ) must have at least one IP address available per Neptune instance. For example, if there is one writer instance and two reader instances in subnet 1, and two reader instances in subnet 2, subnet 1 must have at least 3 IP addresses free and subnet 2 must have at least 2 IP addresses free before starting the upgrade.

At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.

When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for a number of minutes. You may resume write workloads after the upgrade completes.

This process generates the following events:

- Per-cluster event messages:
  - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-`*`(autogenerated snapshot ID)`*`]`
  - `Database cluster major version has been upgraded`
- Per-instance event messages:
  - `Applying off-line patches to DB instance`
  - `DB instance shutdown`
  - `Finished applying off-line patches to DB instance`
  - `DB instance restarted`

Improvements in This Engine Release

- Improved performance of openCypher queries involving `MERGE` and `OPTIONAL MATCH`.
- Improved performance of openCypher queries involving `UNWIND` of a list of maps of literal values.
- Improved performance of openCypher queries that have an `IN` filter for `id`. For example:

```
MATCH (n) WHERE id(n) IN ['1', '2', '3'] RETURN n
```

- Performance improvements and correctness fixes for various Gremlin operators, including `repeat`, `coalesce`, `store`, and `aggregate`.

Defects Fixed in This Engine Release

- Fixed an openCypher bug where a request using HTTP keep-alive could be incorrectly closed if it was submitted after a failed request.
- Fixed an openCypher bug where the parameter type was not always correctly interpreted for a list or a list of maps.
- Fixed an openCypher bug where queries returned the string, `"null"`, instead of a null value in Bolt and SPARQL-JSON.
- Fixed openCypher error codes and error messages for query timeout failures and out-of-memory errors.
- Fixed a Gremlin bug that caused `valueMap()` not to be optimized under a `by()` traversal in the DFE engine.
- Fixed an issue with deadlock detector logic that occasionally made the engine unresponsive.
- Fixed an audit log bug that caused unnecessary information to be logged and certain fields to be missing from the logs.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.1.1.0.R7, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.3`

- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.1.1.0.R7

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.1.1.0`.

Upgrading to This Release

> **Important**
>
> **Upgrading to this engine release from any version prior to `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.**
>
> At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.
>
> When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for around 6 minutes. You may resume write workloads after the upgrade completes.
>
> This process generates the following events:
>
> - Per-cluster event messages:
> - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-`*`(autogenerated snapshot ID)`*`]`
> - `Database cluster major version has been upgraded`
> - Per-instance event messages:
> - `Applying off-line patches to DB instance`
> - `DB instance shutdown`
> - `Finished applying off-line patches to DB instance`

- DB instance restarted

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.1.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.1.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.1.1.0.R6 (2022-09-23)

As of 2022-09-23, engine version 1.1.1.0.R6 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Important

Upgrading to this engine release from a version earlier than `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.

In order to complete the upgrade successfully, each subnet in every availability zone (AZ) must have at least one IP address available per Neptune instance. For example, if there is one writer instance and two reader instances in subnet 1, and two reader instances in subnet 2, subnet 1 must have at least 3 IP addresses free and subnet 2 must have at least 2 IP addresses free before starting the upgrade.

At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.

When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for a number of minutes. You may resume write workloads after the upgrade completes.

This process generates the following events:

- Per-cluster event messages:
  - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-`*`(autogenerated snapshot ID)`*`]`
  - `Database cluster major version has been upgraded`
- Per-instance event messages:
  - `Applying off-line patches to DB instance`
  - `DB instance shutdown`
  - `Finished applying off-line patches to DB instance`
  - `DB instance restarted`

> **Note**
>
> There is a breaking change in this release for code that uses openCypher with IAM authentication. Up to now, the host string in the IAM signature included the protocol, such as `bolt://`, like this:
>
> ```
> "Host":"bolt://(host URL):(port)"
> ```
>
> Starting with engine release `1.1.1.0`, the protocol must be omitted:
>
> ```
> "Host":"(host URL):(port)"
> ```
>
> See Using the Bolt protocol for examples.

Improvements in This Engine Release

- Improved performance of Gremlin `order-by` queries. Gremlin queries with an `order-by` at the end of a `NeptuneGraphQueryStep` now use a larger chunk size for better performance. This does not apply to `order-by` on an internal (non-root) node of the query plan.
- Improved performance of Gremlin update queries. Vertices and edges must now be locked against deletion while adding edges or properties. This change eliminates duplicate locks within a transaction, which improves performance.

Defects Fixed in This Engine Release

- Fixed an openCypher bug in the `MERGE` clause that in some cases caused duplicate node and edge creation.
- Fixed a bug in the handling of SPARQL queries that contain `(NOT) EXISTS` within an `OPTIONAL` clause, where in some cases query results would be missing.
- Fixed a bug that delayed server restart when a bulk load was in progress.
- Fixed a bug where an openCypher variable-length pattern bi-directional traversal with a filter on the relationship property would result in an error. An example of such a variable-length pattern is `(n)-[r*1..2]->(m)`.
- Fixed a bug related to how cached data is sent back to the client, that in some cases resulted in unexpectedly long latency.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.1.1.0.R6, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.1.1.0.R6

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.1.1.0`.

Upgrading to This Release

Important

Upgrading to this engine release from any version prior to `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.

At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.

When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for around 6 minutes. You may resume write workloads after the upgrade completes.

This process generates the following events:

- Per-cluster event messages:

- `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-(autogenerated snapshot ID)]`
- `Database cluster major version has been upgraded`

- Per-instance event messages:
  - `Applying off-line patches to DB instance`
  - `DB instance shutdown`
  - `Finished applying off-line patches to DB instance`
  - `DB instance restarted`

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.1.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.1.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.1.1.0.R5 (2022-07-21)

As of 2022-07-21, engine version 1.1.1.0.R5 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Important**
>
> **Upgrading to this engine release from a version earlier than `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.**
>
> In order to complete the upgrade successfully, each subnet in every availability zone (AZ) must have at least one IP address available per Neptune instance. For example, if there is one writer instance and two reader instances in subnet 1, and two reader instances in subnet 2, subnet 1 must have at least 3 IP addresses free and subnet 2 must have at least 2 IP addresses free before starting the upgrade.
>
> At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.
>
> When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for a number of minutes. You may resume write workloads after the upgrade completes.
>
> This process generates the following events:
>
> - Per-cluster event messages:
> - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-`*`(autogenerated snapshot ID)`*`]`
> - `Database cluster major version has been upgraded`
> - Per-instance event messages:

- Applying off-line patches to DB instance
- DB instance shutdown
- Finished applying off-line patches to DB instance
- DB instance restarted

Note

There is a breaking change in this release for code that uses openCypher with IAM authentication. Up to now, the host string in the IAM signature included the protocol, such as `bolt://`, like this:

```
"Host":"bolt://(host URL):(port)"
```

Starting with engine release `1.1.1.0`, the protocol must be omitted:

```
"Host":"(host URL):(port)"
```

See Using the Bolt protocol for examples.

Improvements in This Engine Release

- Made improvements to support deadlock detection.

Defects Fixed in This Engine Release

- Fixed a bug that prevented a clean shutdown of DB clusters under certain conditions.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.1.1.0.R5, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`

- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.1.1.0.R5

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.1.1.0`.

Upgrading to This Release

> **Important**
>
> **Upgrading to this engine release from any version prior to `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.**
>
> At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.
>
> When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for around 6 minutes. You may resume write workloads after the upgrade completes.
>
> This process generates the following events:
>
> - Per-cluster event messages:
> - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-`*`(autogenerated snapshot ID)`*`]`
> - `Database cluster major version has been upgraded`
> - Per-instance event messages:
> - `Applying off-line patches to DB instance`
> - `DB instance shutdown`
> - `Finished applying off-line patches to DB instance`

- DB instance restarted

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.1.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.1.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.1.1.0.R4 (2022-06-23)

As of 2022-06-23, engine version 1.1.1.0.R4 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Important

Upgrading to this engine release from a version earlier than `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.

In order to complete the upgrade successfully, each subnet in every availability zone (AZ) must have at least one IP address available per Neptune instance. For example, if there is one writer instance and two reader instances in subnet 1, and two reader instances in subnet 2, subnet 1 must have at least 3 IP addresses free and subnet 2 must have at least 2 IP addresses free before starting the upgrade.

At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.

When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for a number of minutes. You may resume write workloads after the upgrade completes.

This process generates the following events:

- Per-cluster event messages:
  - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-`*`(autogenerated snapshot ID)`*`]`
  - `Database cluster major version has been upgraded`
- Per-instance event messages:
  - `Applying off-line patches to DB instance`
  - `DB instance shutdown`
  - `Finished applying off-line patches to DB instance`
  - `DB instance restarted`

Note

There is a breaking change in this release for code that uses openCypher with IAM authentication. Up to now, the host string in the IAM signature included the protocol, such as `bolt://`, like this:

```
"Host":"bolt://(host URL):(port)"
```

Starting with engine release `1.1.1.0`, the protocol must be omitted:

```
"Host":"(host URL):(port)"
```

See Using the Bolt protocol for examples.

Improvements in This Engine Release

- Updated instance configuration for `x2g` instance types.
- Improved performance of vertex drops.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug where solutions were not maintaining a stable order for a query called multiple times or across multiple readers for certain kinds of ASK joins.
- Also, narrowed the scope of a change in the previous release that was causing performance regressions for certain kinds of ASK joins in Gremlin.
- Fixed a Gremlin bug in the `union()` step that occurred when there was an edge input and a traversal to a vertex within child traversals.
- Fixed a Gremlin profile bug where some steps were reported as not optimized when they actually were.
- Fixed a SPARQL bug where variables used inside `FILTER` expressions nested into `UNION` clauses were getting assigned invalid scoping information.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.1.1.0.R4, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.1.1.0.R4

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.1.1.0`.

Upgrading to This Release

> **Important**
>
> **Upgrading to this engine release from any version prior to `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.**
>
> At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.
>
> When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for around 6 minutes. You may resume write workloads after the upgrade completes.
>
> This process generates the following events:
>
> - Per-cluster event messages:

    - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-(autogenerated snapshot ID)]`
    - `Database cluster major version has been upgraded`
  - Per-instance event messages:
    - `Applying off-line patches to DB instance`
    - `DB instance shutdown`
    - `Finished applying off-line patches to DB instance`
    - `DB instance restarted`

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.1.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.1.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.1.1.0.R3 (2022-06-07)

As of 2022-06-07, engine version 1.1.1.0.R3 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Important**
>
> **Upgrading to this engine release from a version earlier than `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.**
>
> At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.
>
> When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for a number of minutes. You may resume write workloads after the upgrade completes.
>
> This process generates the following events:
>
> - Per-cluster event messages:
> - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-`*`(autogenerated snapshot ID)`*`]`
> - `Database cluster major version has been upgraded`
> - Per-instance event messages:
> - `Applying off-line patches to DB instance`
> - `DB instance shutdown`
> - `Finished applying off-line patches to DB instance`
> - `DB instance restarted`

Note

There is a breaking change in this release for code that uses openCypher with IAM authentication. Up to now, the host string in the IAM signature included the protocol, such as `bolt://`, like this:

```
"Host":"bolt://(host URL):(port)"
```

Starting with engine release `1.1.1.0`, the protocol must be omitted:

```
"Host":"(host URL):(port)"
```

See Using the Bolt protocol for examples.

Improvements in This Engine Release

- Added support for the Graviton2-powered `x2g` instance types, optimized for memory-intensive workloads. These are initially available only in four AWS Regions:
  - US East (N. Virginia) (`us-east-1`)
  - US East (Ohio) (`us-east-2`)
  - US West (Oregon) (`us-west-2`)
  - Europe (Ireland) (`eu-west-1`)

  See the Neptune pricing page for more information.
- Improved performance of Gremlin steps where multiple edge or vertex traversals, property lookups, or label lookups are involved.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug in the processing of the `otherV()` step inside a child traversal.
- Fixed a Gremlin bug in queries with `union` having only filter steps as children. For example:

```
g.V().union(has("name"), out("knows")).out()
```

- Fixed a SPARQL bug where variables used inside `FILTER` expressions nested into `UNION` clauses were getting assigned invalid scoping information.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.1.1.0.R3, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.1.1.0.R3

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.1.1.0`.

Upgrading to This Release

> **Important**
>
> **Upgrading to this engine release from any version prior to `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.**
>
> At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.
>
> When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for around 6 minutes. You may resume write workloads after the upgrade completes.

This process generates the following events:

- Per-cluster event messages:
  - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-(autogenerated snapshot ID)]`
  - `Database cluster major version has been upgraded`
- Per-instance event messages:
  - `Applying off-line patches to DB instance`
  - `DB instance shutdown`
  - `Finished applying off-line patches to DB instance`
  - `DB instance restarted`

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.1.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.1.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune maintenance release, version 1.1.1.0.R2 (2022-05-16)

As of 2022-05-16, engine version 1.1.1.0.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Important

Upgrading to this engine release from a version earlier than `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.

In order to complete the upgrade successfully, each subnet in every availability zone (AZ) must have at least one IP address available per Neptune instance. For example, if there is one writer instance and two reader instances in subnet 1, and two reader instances in subnet 2, subnet 1 must have at least 3 IP addresses free and subnet 2 must have at least 2 IP addresses free before starting the upgrade.

At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.

When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for a number of minutes. You may resume write workloads after the upgrade completes.

This process generates the following events:

- Per-cluster event messages:

    - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-(autogenerated snapshot ID)]`
    - `Database cluster major version has been upgraded`
  - Per-instance event messages:
    - `Applying off-line patches to DB instance`
    - `DB instance shutdown`
    - `Finished applying off-line patches to DB instance`
    - `DB instance restarted`

> **Note**
>
> There is a breaking change in this release for code that uses openCypher with IAM authentication. Up to now, the host string in the IAM signature included the protocol, such as `bolt://`, like this:
>
> ```
> "Host":"bolt://(host URL):(port)"
> ```
>
> Starting with engine release `1.1.1.0`, the protocol must be omitted:
>
> ```
> "Host":"(host URL):(port)"
> ```
>
> See Using the Bolt protocol for examples.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.1.1.0.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin earliest version supported:* `3.5.2`
- *Gremlin latest version supported:* `3.5.4`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade paths to engine release 1.1.1.0.R2

Your cluster will be upgraded to this maintenance patch release automatically during your next maintenance window if you are running engine version `1.1.1.0`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Important

Upgrading to this engine release from any version prior to `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.

At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.

When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for around 6 minutes. You may resume write workloads after the upgrade completes.

This process generates the following events:

- Per-cluster event messages:
  - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-`*`(autogenerated snapshot ID)`*`]`
  - `Database cluster major version has been upgraded`
- Per-instance event messages:
  - `Applying off-line patches to DB instance`
  - `DB instance shutdown`
  - `Finished applying off-line patches to DB instance`
  - `DB instance restarted`

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.1.1.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.1.1.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.1.0.0 (2021-11-19)

As of 2021-11-19, engine version 1.1.0.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Important**
>
> **Upgrading to this engine release from a version earlier than `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed,**

you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.

In order to complete the upgrade successfully, each subnet in every availability zone (AZ) must have at least one IP address available per Neptune instance. For example, if there is one writer instance and two reader instances in subnet 1, and two reader instances in subnet 2, subnet 1 must have at least 3 IP addresses free and subnet 2 must have at least 2 IP addresses free before starting the upgrade.

At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.

When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for a number of minutes. You may resume write workloads after the upgrade completes.

This process generates the following events:

- Per-cluster event messages:
  - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-`*`(autogenerated snapshot ID)`*`]`
  - `Database cluster major version has been upgraded`
- Per-instance event messages:
  - `Applying off-line patches to DB instance`
  - `DB instance shutdown`
  - `Finished applying off-line patches to DB instance`
  - `DB instance restarted`

Note

Starting with this engine release, Neptune no longer supports R4 instance types. If you are using an R4 instance in your DB cluster, you must manually replace it with a different instance type before upgrading to this release. If your writer instance is an R4, follow these instructions to move it.

Subsequent patch releases for this release

- Maintenance release: 1.1.0.0.R2 (2022-05-16)
- Maintenance release: 1.1.0.0.R3 (2022-12-23)

New Features in This Engine Release

- Introduced general-purpose `T4g` and memory-optimized `R6g` database instances powered by the AWS Graviton2 processor. Graviton2-based instances deliver significantly better price/performance than comparable current-generation x86-based instances for a variety of workloads. Applications work as normal on these new instance types, and there is no need to port application code when you upgrade to them.

  For more information on pricing and regional availability, see the Amazon Neptune pricing page.
- Introduced custom models in Neptune ML.
- Added support for SPARQL inference queries in Neptune ML.
- Added a new streams endpoint for property-graph data, namely:

  ```
  https://Neptune-DNS:8182/propertygraph/stream
  ```

  The output format of this endpoint, named `PG_JSON`, is exactly the same as as the `GREMLIN_JSON` format output by the old `gremlin/stream`.

  The new `propertygraph/stream` endpoint extends Neptune stream support to openCypher and replaces the `gremlin/stream` endpoint with its associated `GREMLIN_JSON` output format.

Improvements in This Engine Release

- Made improvements to Neptune streams:
  - Added a `commitTimestamp` field to the `records` object the Neptune streams change-log response format, to provide a timestamp for each record in a change-log stream.
  - Added a `LATEST` value to the `iteratorType` parameter, allowing you to retrieve the last valid eventId from the streams. See Calling the Streams API.
- Added support for getting the inference confidence score in Gremlin node classification and regression queries.

- Added support for the `OPTIONAL MATCH` clause in openCypher.
- Added support for the `MERGE` clause in openCypher.
- Added support for using `ORDER BY` in `WITH` clauses in openCypher.
- Added support for pattern comprehension in openCypher, and extended support for pattern expression beyond existence checking.
- Extended support for the `DELETE` and `DELETE DETACH` clauses in openCypher, so that they can now be used with other update clauses.
- Extended support for `CREATE` and `UPDATE` clauses used with `RETURN` in openCypher.
- Added support in the DFE engine for the Gremlin `limit`, `range`, and `skip` steps.
- Improved query execution in the DFE engine when neither `explain` nor `profile` is requested.
- Improved query execution in the DFE engine for the `value` expression.
- Improved a number of chained Gremlin conditional insert patterns so as to avoid concurrent-modification exceptions and allow chaining of query patterns like these:
  - Conditional vertex insertion by ID, such as:

    ```
    g.V(ID).fold().coalesce(unfold(), g.addV("L1").property(id,ID))
    ```

  - Conditional vertex insertion with multiple labels, such as:

    ```
    g.V(ID).fold().coalesce(unfold(), g.addV("L1::L2").property(id,ID))
    ```

  - Conditional edge insertion by ID, such as:

    ```
    g.E(ID).fold().coalesce(unfold(), V(from).addE(label).to(V(to)).property(id, ID))
    ```

  - Conditional edge insertion with multiple labels, such as:

    ```
    g.E(ID).fold().coalesce(unfold(),
     g.addE(label).from(V(from)).to(V(to)).property(id, ID))
    ```

  - Conditional insertion followed by a query, such as:

    ```
    g.V(ID).fold().coalesce(unfold(),
     g.addV("L1").property(id,ID)).project("myvalues").by(valueMap())
    ```

  - Conditional insertion with added properties, such as:

```
g.V(ID).fold().coalesce(unfold(),
 g.addV("L1").property(id,ID).property("name","pumba"))
```

Defects Fixed in This Engine Release

- Disabled the statistics feature on `T3.medium` instance types, which were not able to support it.
- Fixed a SPARQL bug in `explain` with an `IN` function that took non-constant values.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.1.0.0, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.11`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.1.0.0

You can manually upgrade any previous Neptune engine release to this release.

You will not automatically be upgraded to this release.

Upgrading to This Release

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.1.0.0 \
    --allow-major-version-upgrade \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.1.0.0 ^
    --allow-major-version-upgrade ^
    --apply-immediately
```

Instead of `--apply-immediately`, you can specify `--no-apply-immediately`. To perform a major version upgrade, the allow-major-version-upgrade parameter is required. Also, be sure to include the engine version or your engine may be upgraded to a different version.

If your cluster uses a custom cluster parameter group, be sure to include this paramater to specify it:

```
    --db-cluster-parameter-group-name (name of the custom DB cluster parameter group)
```

Similarly, if any instances in the cluster use a custom DB parameter group, be sure to include this parameter to specify it:

```
    --db-instance-parameter-group-name (name of the custom instance parameter group)
```

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune maintenance release, version 1.1.0.0.R3 (2022-12-23)

As of 2022-12-23, engine version 1.1.0.0.R3 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

> **Important**
>
> **Upgrading to this engine release from a version earlier than `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed,**

you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.

In order to complete the upgrade successfully, each subnet in every availability zone (AZ) must have at least one IP address available per Neptune instance. For example, if there is one writer instance and two reader instances in subnet 1, and two reader instances in subnet 2, subnet 1 must have at least 3 IP addresses free and subnet 2 must have at least 2 IP addresses free before starting the upgrade.

At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.

When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for a number of minutes. You may resume write workloads after the upgrade completes.

This process generates the following events:

- Per-cluster event messages:
  - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-`*`(autogenerated snapshot ID)`*`]`
  - `Database cluster major version has been upgraded`
- Per-instance event messages:
  - `Applying off-line patches to DB instance`
  - `DB instance shutdown`
  - `Finished applying off-line patches to DB instance`
  - `DB instance restarted`

Improvements in This Engine Release

- Performance improvements and correctness fixes for various Gremlin operators, including `repeat`, `coalesce`, `store`, and `aggregate`.

Defects Fixed in This Engine Release

- Fixed a CPU spike issue.
- Fixed an openCypher bug where queries returned the string, `"null"`, instead of a null value in Bolt and SPARQL-JSON.
- Fixed an audit log bug that caused unnecessary information to be logged and certain fields to be missing from the logs.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.1.0.0.R3, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.11`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.1.0.0.R3

Your cluster will be upgraded to this maintenance patch release automatically during your next maintenance window if you are running engine version `1.1.0.0`.

> **Important**
>
> **Upgrading to this engine release from any version prior to `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.**
>
> At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.
>
> When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be

unavailable at this point for around 6 minutes. You may resume write workloads after the upgrade completes.
This process generates the following events:

- Per-cluster event messages:
  - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-(autogenerated snapshot ID)]`
  - `Database cluster major version has been upgraded`
- Per-instance event messages:
  - `Applying off-line patches to DB instance`
  - `DB instance shutdown`
  - `Finished applying off-line patches to DB instance`
  - `DB instance restarted`

> **Note**
>
> Starting with this engine release, Neptune no longer supports R4 instance types. If you are using an R4 instance in your DB cluster, you must manually replace it with a different instance type before upgrading to this release. If your writer instance is an R4, follow these instructions to move it.

Upgrading to This Release

Amazon Neptune 1.1.0.0.R3 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.1.0.0 \
```

```
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.1.0.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

Note

If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune maintenance release, version 1.1.0.0.R2 (2022-05-16)

As of 2022-05-16, engine version 1.1.0.0.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Important

Upgrading to this engine release from a version earlier than `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.

At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.

When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be

unavailable at this point for a number of minutes. You may resume write workloads after the upgrade completes.
This process generates the following events:

- Per-cluster event messages:
  - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-(autogenerated snapshot ID)]`
  - `Database cluster major version has been upgraded`
- Per-instance event messages:
  - `Applying off-line patches to DB instance`
  - `DB instance shutdown`
  - `Finished applying off-line patches to DB instance`
  - `DB instance restarted`

Defects Fixed in This Engine Release

- Fixed a bug where the internal credential cache was not being cleared properly for non-query endpoints such as the status endpoint.
- Fixed a bug that caused replication lag to increase after an engine upgrade.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.1.0.0.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.11`
- *openCypher version:* `Neptune-9.0.20190305-1.0`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.1.0.0.R2

Your cluster will be upgraded to this maintenance patch release automatically during your next maintenance window if you are running engine version `1.1.0.0`.

Important

Upgrading to this engine release from any version prior to `1.1.0.0` also triggers an operating-system upgrade on all the instances in your DB cluster. Because active write requests that occur during the operating-system upgrade will not be processed, you must pause all write workloads to the cluster being upgraded, including bulk data loads, before starting the upgrade.

At the start of the upgrade, Neptune generates a snapshot with a name composed of `preupgrade` followed by an autogenerated identifier based on your DB cluster information. You will not be charged for this snapshot, and you can use it to restore your DB cluster if anything goes wrong during the upgrade process.

When the engine upgrade itself has completed, the new engine version will be available briefly on the old operating system, but in less than 5 minutes all the the instances in your cluster will simultaneously begin an operating-system upgrade. Your DB cluster will be unavailable at this point for around 6 minutes. You may resume write workloads after the upgrade completes.

This process generates the following events:

- Per-cluster event messages:
  - `Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-`*`(autogenerated snapshot ID)`*`]`
  - `Database cluster major version has been upgraded`
- Per-instance event messages:
  - `Applying off-line patches to DB instance`
  - `DB instance shutdown`
  - `Finished applying off-line patches to DB instance`
  - `DB instance restarted`

Note

Starting with this engine release, Neptune no longer supports R4 instance types. If you are using an R4 instance in your DB cluster, you must manually replace it with a different instance type before upgrading to this release. If your writer instance is an R4, follow these instructions to move it.

Upgrading to This Release

Amazon Neptune 1.1.0.0.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.1.0.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.1.0.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.5.1 (2021-10-01)

As of 2021-10-01, engine version 1.0.5.1 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Subsequent Patch Releases for This Release

- Release: 1.0.5.1.R2 (2021-10-26)
- Release: 1.0.5.1.R3 (2022-01-13)
- Maintenance release: 1.0.5.1.R4 (2022-05-16)

New Features in This Engine Release

- Added a results cache for caching the results of specified queries.
- Added Date/time support in Neptune openCypher.
- Added support for `List` and `Map` access to elements in Neptune openCypher.

Improvements in This Engine Release

- Made Neptune openCypher endpoint names case-insensitive.
- Improved openCypher explain.
- Improved Gremlin single upsert query patterns terminating with `iterate()` and `profile()` steps.
- Improved performance in Gremlin `keys()` and `property()` functions.
- The Gremlin `dedup()` step is run in the DFE when it is used with global scope.
- The following Gremlin HAS predicates are run in the DFE engine when the DFE engine is enabled:
  - `EQ`
  - `NEQ`
  - `LT`
  - `LTE`
  - `GT`
  - `GTE`
  - `BETWEEN`
  - `INSIDE`
  - `OUTSIDE`
  - `WITHIN`
  - `AND (connectives)`

- `OR (connectives)`
- Improved LIMIT query performance.
- Improved performance of openCypher general aggregation queries.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug that allowed an edge to be connected to another edge.
- Fixed a Gremlin bug that caused a sub-optimal join strategy to be chosen.
- Fixed a Gremlin bug that caused serialization of nodes and relationships to stall when more than 100 properties were present.
- Fixed a bug that slowed down query execution planning for queries with large graph patterns.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.5.1, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.11`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.5.1

You can manually upgrade any previous Neptune engine release to this release.

You will not automatically upgrade to this release.

Upgrading to This Release

Amazon Neptune 1.0.5.1 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.5.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.5.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot

that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune maintenance release, version 1.0.5.1.R4 (2022-05-16)

As of 2022-05-16, engine version 1.0.5.1.R4 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.5.1.R4, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.11`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.5.1.R4

Your cluster will be upgraded to this maintenance patch release automatically during your next maintenance window if you are running engine version `1.0.5.1`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.5.1.R4 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.5.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.5.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.5.1.R3 (2022-01-13)

As of 2022-01-13, engine version 1.0.5.1.R3 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects Fixed in This Engine Release

- Fixed a bug that can cause a resource leak when a query fails to acquire all the resources it needs.
- Fixed a small memory leak during query execution caused by an unclaimed memory allocation.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.5.1.R3, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.11`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.5.1.R3

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.5.1`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.5.1.R3 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.5.1 \
```

```
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.5.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.5.1.R2 (2021-10-26)

As of 2021-10-26, engine version 1.0.5.1.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects Fixed in This Engine Release

- Fixed a bug that caused a server restart when a transient error occurred while creating an older version of a graph element, under repeatable read isolation. Neptune now returns an error instead, so that the client can retry.
- Fixed a bug that caused a server restart when a transient error occurrred during a single cardinality update. Neptune now returns an error instead, so that the client can retry.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.5.1.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.11`

- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.5.1.R2

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.5.1`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.5.1.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.5.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.5.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.5.0 (2021-07-27)

As of 2021-07-27, engine version 1.0.5.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Subsequent Patch Releases for This Release

- Release: 1.0.5.0.R2 (2021-08-16)
- Release: 1.0.5.0.R3 (2021-09-15)
- Maintenance release: 1.0.5.0.R5 (2022-05-16)

New Features in This Engine Release

- Neptune ML was released for production use with many new features, and is no longer in lab mode.
- Added initial support for the openCypher query language, in Lab Mode. **openCypher** is the open-source standard for the Cypher query language. Its syntax is specified in the Cypher Query Language Reference (Version 9), and is maintained by the openCypher project.

  See Accessing the Neptune Graph with openCypher for information about the Neptune implementation of the language.

  Support for the Bolt protocol, which Neptune clients use for openCypher queries, is also supported. See Using the Bolt protocol to make openCypher queries to Neptune.

  Support for openCypher is now automatically enabled, but it depends on the Neptune DFE engine, which is currently only available in lab mode. The default `DFEQueryEngine` setting in the `neptune_lab_mode` DB cluster parameter is now `DFEQueryEngine=viaQueryHint`, which means that the engine is enabled but only used for queries that have the `useDFE` query hint present and set to `true`. If you disable the DFE engine by setting `DFEQueryEngine=disabled`, you will not be able to use openCypher.
- Added support for the SPARQL 1.1 Graph Store HTTP Protocol. See Using the SPARQL 1.1 Graph Store HTTP Protocol (GSP) in Amazon Neptune.

- Changed the default lab-mode setting for the Neptune DFE engine to `viaQueryHint`, which means that the DFE engine is now enabled by default, but only used for queries that have the `useDFE` query hint present and set to `true`.
- Added a new Amazon CloudWatch metric, `StatsNumStatementsScanned`, for monitoring the computation of statistics for the Neptune DFE engine. See Using the `StatsNumStatementsScanned` CloudWatch metric to monitor statistics computation.

Improvements in This Engine Release

- Added support for Apache TinkerPop 3.4.11.

> **Important**
>
> A change was made in TinkerPop version 3.4.11 that improves correctness of how queries are processed, but for the moment can sometimes seriously impact query performance. For example, a query of this sort may run significantly slower:
>
> ```
> g.V().hasLabel('airport').
>  order().
>  by(out().count(),desc).
>  limit(10).
>  out()
> ```
>
> The vertices after the limit step are now fetched in a non-optimal way beause of the TinkerPop 3.4.11 change. To avoid this, you can modify the query by adding the barrier() step at any point after the `order().by()`. For example:
>
> ```
> g.V().hasLabel('airport').
>  order().
>  by(out().count(),desc).
>  limit(10).
>  barrier().
>  out()
> ```

- The SPARQL `joinOrder` query hint is now supported by the Neptune DFE alternative query engine.

- The output of the Neptune status API has been expanded and reorganized to provide more clarity about your DB cluster's settings and features.

  The new output has a top-level `features` object that contains status information about your DB cluster's features, and a top-level `settings` object that contain settings information. To review the new format, see Example output from the instance status command.
- Handling of streaming change logs has been improved when `AFTER_SEQUENCE_NUMBER` streams are requested with the last event ID on the server, when that event ID has already expired. The server no longer throws an expired event ID error if the requested event ID is the most recently purged event ID on the server.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug related to the ordering of numeric values.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.5.0, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.11`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.5.0

You can manually upgrade any previous Neptune engine release to this release.

You will not automatically upgrade to this release.

Upgrading to This Release

Amazon Neptune 1.0.5.0 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.5.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.5.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune maintenance release, version 1.0.5.0.R5 (2022-05-16)

As of 2022-05-16, engine version 1.0.5.0.R5 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.5.0.R5, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.11`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.5.0.R5

Your cluster will be upgraded to this maintenance patch release automatically during your next maintenance window if you are running engine version 1.0.5.0.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.5.0.R5 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.5.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.5.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.5.0.R3 (2021-09-15)

As of 2021-09-15, engine version 1.0.5.0.R3 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects Fixed in This Engine Release

- Fixed a bug that causes the engine to become unresponsive in either of these situations:
  - A bulk load happens at the same time as automatic statistics computation is taking place.
  - A statistics computation was requested manually at the same time that one was already occurring.
- Fixed a bug in deadlock detection and in lock acquisition that could cause the engine to crash.
- Fixed a Gremlin bug where the engine threw an error when it encountered unknown data from a remote ML endpoint in a Gremlin inference query.
- Fixed several bugs in ML model management APIs related to model transform jobs and instance recommendations.
- Fixed a bug that could cause the engine to crash when generating node and edge IDs.
- Fixed a bug that slowed down the generation of query plans for queries with large graph patterns.
- Fixed an openCypher bug that could cause a query to stall when retrieving a node that had more than 100 properties.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.5.0.R3, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.11`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.5.0.R3

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version 1.0.5.0.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.5.0.R3 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.5.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.5.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.5.0.R2 (2021-08-16)

As of 2021-08-16, engine version 1.0.5.0.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects Fixed in This Engine Release

- Disabled an optimization made in engine release `1.0.5.0` that made the Neptune lookup cache survive engine restarts on replicas. Replica restarts now clear the lookup cache.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.5.0.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.11`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.5.0.R2

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.5.0`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.5.0.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.5.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
```

```
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.5.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.4.2 (2021-06-01)

Note

Engine release version 1.0.4.2.R2 was the first version of 1.0.4.2 actually to be released.

Topics

- Amazon Neptune Engine Version 1.0.4.2.R5 (2021-08-16)
- Amazon Neptune Engine Version 1.0.4.2.R4 (2021-07-23)
- Amazon Neptune Engine Version 1.0.4.2.R3 (2021-06-28)
- Amazon Neptune Engine Version 1.0.4.2.R2 (2021-06-01)
- Amazon Neptune Engine Version 1.0.4.2.R1 (2021-05-27)

Amazon Neptune Engine Version 1.0.4.2.R5 (2021-08-16)

As of 2021-08-16, engine version 1.0.4.2.R5 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects Fixed in This Engine Release

- Disabled an optimization made in engine release `1.0.4.2.R4` that made the Neptune lookup cache survive engine restarts on replicas. Replica restarts now clear the lookup cache.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.4.2.R5, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.10`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.4.2.R5

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.4.2`.

You can manually upgrade any previous Neptune engine release to this release.

Amazon Neptune Engine Version 1.0.4.2.R4 (2021-07-23)

As of 2021-07-23, engine version 1.0.4.2.R4 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Improvements in This Engine Release

- Improved the behavior of the lookup cache to avoid redundant cache clearing after running fast reset on a replica.
- Improved handling of streaming change logs when `AFTER_SEQUENCE_NUMBER` streams are requested with the last event ID on the server, when that event ID has already expired. The server no longer throws an expired event ID error if the requested event ID is the most recently purged event ID on the server.

Defects Fixed in This Engine Release

- Fixed a bug introduced in 1.0.4.0.R1 where queries would not return the entirety of string values larger than 760 characters. The terms affected by this bug were RDF literals and URIs, or Gremlin IDs, keys, and string values.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.4.2.R4, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.10`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.4.2.R4

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.4.2`.

You can manually upgrade any previous Neptune engine release to this release.

Amazon Neptune Engine Version 1.0.4.2.R3 (2021-06-28)

As of 2021-06-28, engine version 1.0.4.2.R3 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Known issues in this engine release

Issue:

A SPARQL bug that fails to honor media type in an `Accept` header if there are spaces present.

For example, a query with `-H "Accept: text/csv; q=1.0, */*; q=0.1"` returns JSON output rather than CSV output.

Workaround:

If you remove the spaces in the `Accept` clause in the header, the engine returns output in the correct requested format. In other words, instead of `-H "Accept: text/csv; q=1.0, */*; q=0.1"` , use:

```
-H "Accept: text/csv;q=1.0,*/*;q=0.1"
```

Defects Fixed in This Engine Release

- Fixed a bug in clearing the lookup cache on replicas after a fast reset.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.4.2.R3, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.10`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.4.2.R3

This patch release is optional unless your DB cluster is using one or more R5d instances. If your cluster has R5d instances, it will automatically be upgraded in the next maintenance window. Otherwise, it will not automatically be upgraded to this patch release.

You can upgrade release `1.0.4.2.R2` to this `1.0.4.2.R3` release manually using the AWS CLI apply-pending-maintenance-action command (the ApplyPendingMaintenanceAction API).

Amazon Neptune Engine Version 1.0.4.2.R2 (2021-06-01)

As of 2021-06-01, engine version 1.0.4.2.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Subsequent Patch Releases for This Release

- Release: 1.0.4.2.R3 (2021-06-28)

Known issues in this engine release

Issue:

A SPARQL bug that fails to honor media type in an `Accept` header if there are spaces present.

For example, a query with `-H "Accept: text/csv; q=1.0, */*; q=0.1"` returns JSON output rather than CSV output.

Workaround:

If you remove the spaces in the `Accept` clause in the header, the engine returns output in the correct requested format. In other words, instead of `-H "Accept: text/csv; q=1.0, */*; q=0.1"` , use:

```
-H "Accept: text/csv;q=1.0,*/*;q=0.1"
```

New Features in This Engine Release

- Added the new R5d instance type, which includes a lookup cache for speeding up reads in use cases involving a high volume of property value or RDF literal lookups. See The Neptune lookup cache can accelerate read queries.
- Added a new lab-mode parameter that lets the experimental DFE engine be invoked only on a per-query basis with the `useDFE` query hint.

Improvements in This Engine Release

- Added support for TinkerPop 3.4.10.
- Added support for using the `withStrategies( )` configuration step when sending Gremlin script requests. Specifically, the `SubgraphStrategy`, `PartitionStrategy`, `ReadOnlyStrategy`, `EdgeLabelVerificationStrategy`, and `ReservedKeysVerificationStrategy` are all supported.
- Added optimization for `V()` traversals in the middle of a query. Previously, such traversals were not optimized in Neptune.
- Added support for RFC 2141 URNs to be used as the `baseUri` and `namedGraphUri` parameters for a bulk load.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug in the parser where incorrect queries were treated as valid.
- Fixed a Gremlin bug where unfolding an `aggregate()` side-effect with `cap().unfold()` to a `valueMap()` would raise an exception.

- Fixed a Gremlin bug where some `property()` steps after an `addV()` step fail with a "cannot cast to String" error.
- Fixed a Gremlin bug to prevent some conditional insert patterns from raising concurrent-modification exceptions.
- Fixed a Gremlin bug so that the query request timeout now cannot exceed the session timeout.
- Fixed a SPARQL bug where updates using LOAD or UNLOAD could fail with an HTTP code 500 instead of HTTP code 400 when the remote server is unavailable.
- Fixed a bug where stream API calls were failing when `commitNum` or `opNum` values larger than the 32-bit signed integer limit (2,147,483,647) were used.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.4.2.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.10`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.4.2.R2

You can manually upgrade any previous Neptune engine release to this release.

You will not automatically upgrade to this release.

Upgrading to This Release

Amazon Neptune 1.0.4.2.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.4.2 \
```

```
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.4.2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.4.2.R1 (2021-05-27)

Engine release 1.0.4.2.R1 was never deployed.

Amazon Neptune Engine Version 1.0.4.1 (2020-12-08)

As of 2020-12-08, engine version 1.0.4.1 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Subsequent Patch Releases for This Release

- Release: 1.0.4.1.R1.1 (2021-03-22)
- Release: 1.0.4.1.R2 (2021-02-24)

> **Important**
>
> Release: 1.0.4.0 (2020-10-12) made TLS 1.2 and HTTPS mandatory for all connections to Amazon Neptune. However, a bug in that release has allowed HTTP connections and/

or outdated TLS connections to continue to work for customers who previously set a DB cluster parameter to prevent enforcement of HTTPS connections.
That bug was fixed in patch releases 1.0.4.0.R2 and 1.0.4.1.R2, but the fix has caused unexpected connection failures when the patches are automatically installed. For this reason, both patches have been reverted, and can only be installed manually, to give you a chance to update your setup for TLS 1.2.
Having to use SSL/TLS for all connections to Neptune affects your connections with the Gremlin console, the Gremlin driver, Gremlin Python, .NET, nodeJs, REST APIs, and also load-balancer connections. If you have been using HTTP or an older TLS version for any or all of these up until now, you must update the relevant client and drivers and change your code to use HTTPS exclusively before updating your system to the latest patches.

New Features in This Engine Release

- Introduced the Neptune ML feature, which brings powerful machine learning capabilities to Amazon Neptune. See Amazon Neptune ML for machine learning on graphs.
- Added a custom SPARQL `UNLOAD` operation for removing data retrieved from a remote source. See SPARQL UPDATE UNLOAD.

Improvements in This Engine Release

- Optimized some Gremlin conditional insert patterns to avoid concurrent-modification exceptions.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug that could cause missing results for a specific pattern of queries that used the `as()` step.
- Fixed a Gremlin bug that could cause errors when using the `project()` step nested inside another step such as `union()`.
- Fixed a Gremlin bug in the `project()` step.
- Fixed a Gremlin bug in string-based traversal where the `none()` step did not work.
- Fixed a Gremlin bug in string-based traversal where an empty map was not supported as an arguent to the `inject()` step.

- Fixed a Gremlin bug in string-based traversal execution in the DFE engine where a terminal method such as `toList()` did not work properly.
- Fixed a Gremlin bug that failed to close transactions which used the `iterate()` step in String queries.
- Fixed a Gremlin bug that could cause queries using the `is(P.gte(0))` pattern to throw an exception in some situations.
- Fixed a Gremlin bug that could cause queries using the `order().by(T.id)` pattern to throw an exception in some situations.
- Fixed a Gremlin bug that could cause queries using the `addV().aggregate()` pattern to give incorrect results in some situations.
- Fixed a Gremlin bug that could cause queries using the `path()` step followed by the `project()` step pattern to throw an exception in some situations.
- Fixed a SPARQL bug where the SUBSTR function signals an error instead of returning an empty string.
- Fixed a bug in the DFE engine that could cause join operations in non-blocking query plans to generate incorrect results in the presence of unbound variables.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.4.1, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.8`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.4.1

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.4.1`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.4.1 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.4.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.4.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.4.1.R1.1 (2021-03-22)

As of 2021-03-22, engine version 1.0.4.1.R1.1 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects Fixed in This Engine Release

- Disabled an optimization for Gremlin conditional insert patterns which can add or append to existing labels and properties.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.4.1.R1.1, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.8`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.4.1.R1.1

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.4.1`.

Upgrading to This Release

Amazon Neptune 1.0.4.1.R1.1 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.4.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.4.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.4.1.R2 (2021-02-24)

As of 2021-02-24, engine version 1.0.4.1.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Subsequent Patch Releases for This Release

- Release: 1.0.4.1.R2.1 (2021-03-11)

New Features in This Engine Release

- Neptune now supports compression of single files in `bzip2` format for bulk loads. See Load Data Formats.

Defects Fixed in This Engine Release

- Fixed a bug in Release: 1.0.4.0 (2020-10-12) that allowed connections to Neptune using HTTP or earlier versions of TLS, rather than HTTPS and TLS 1.2.

> **Important**
>
> **Having to use SSL/TLS for all connections to Neptune can be a breaking change.** It affects your connections with the Gremlin console, the Gremlin driver, Gremlin Python, .NET, nodeJs, REST APIs, and also load-balancer connections. If you have been using HTTP or an older TLS version for any or all of these up until now, you must update the relevant client and drivers before installing this patch, and change your code to use HTTPS exclusively.

- Fixed a Gremlin bug where `InternalFailureException` was set as the response code in certain circumstances when a `ConcurrentModificationException` occurred.
- Fixed a Gremlin bug where under certain conditions updating edges or vertices could cause a transient `InternalFailureException`.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.4.1.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.8`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.4.1.R2

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.4.1`.

Upgrading to This Release

Amazon Neptune 1.0.4.1.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.4.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
```

```
    --engine-version 1.0.4.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.4.1.R2.1 (2021-03-11)

As of 2021-03-11, engine version 1.0.4.1.R2.1 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects Fixed in This Engine Release

- Disabled an optimization for Gremlin conditional insert patterns which can add or append to existing labels and properties.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.4.1.R2.1, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.8`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.4.1.R2.1

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.4.1.R2`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.4.1.R2.1 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.4.1.R2 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.4.1.R2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.4.0 (2020-10-12)

As of 2020-10-12, engine version 1.0.4.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Subsequent Patch Releases for This Release

- Release: 1.0.4.0.R2 (2021-02-24)

New Features in This Engine Release

- Added frame-level compression for Gremlin.

Improvements in This Engine Release

- Amazon Neptune now requires the use of the Secure Sockets Layer (SSL) with the TLSv1.2 protocol for all connections to Neptune in all regions, using these strong cipher suites:
  - `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`
  - `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`
  - `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384`
  - `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256`
  - `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA`
  - `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA`

  This is true for both REST and WebSocket connections to Neptune, and means that you must use HTTPS rather than HTTP when connecting to Neptune in all regions.

  Because client connections using HTTP or TLS 1.1 will no longer be supported anywhere, please make sure that your clients and code have been updated to use TLS 1.2 and HTTPS before upgrading to this engine release.

> **Important**
>
> **Having to use SSL/TLS for all connections to Neptune can be a breaking change.** It affects your connections with the Gremlin console, the Gremlin driver, Gremlin Python, .NET, nodeJs, REST APIs, and also load-balancer connections. If you have been using HTTP for any or all of these, you must now update the relevant client and drivers and change your code to use HTTPS or your connections will fail.
> A bug in this release has allowed HTTP connections and/or outdated TLS connections to continue to work for customers who previously set a DB cluster parameter to prevent

enforcement of HTTPS connections. That bug was fixed in patch releases 1.0.4.0.R2 and 1.0.4.1.R2, but the fix has caused unexpected connection failures when the patches are automatically installed.
For this reason, both patches have been reverted, and can only be installed manually, to give you a chance to update your setup for TLS 1.2.

- Upgraded TinkerPop to version 3.4.8. This is a backwards compatible upgrade. See the TinkerPop change log for what's new.
- Improved performance for the Gremlin `properties()` step.
- Added details about `BindOp` and `MultiplexerOp` in explain and profile reports.
- Added data prefetch to improve performance when there are cache misses.
- Added a new `allowEmptyStrings` setting in the bulk loader's `parserConfiguration` parameter that allows empty strings to be treated as valid property values in CSV loads (see Neptune Loader Request Parameters).
- The loader now allows an escaped semicolon in multivalue CSV columns.

Defects Fixed in This Engine Release

- Fixed a potential Gremlin memory leak related to the `both()` step.
- Fixed a bug where request metrics were missing because an endpoint ending in '/' was not being handled correctly.
- Fix a bug that caused replicas to fall behind and restart under heavy load when the DFE engine is enabled in lab mode.
- Fixed a bug that prevented the correct error message from being reported when a bulk load failed because of an out-of-memory condition.
- Fixed a SPARQL bug where the character encoding was placed in the Content-Encoding header in SPARQL query responses. Now `charset` is placed in the Content-Type header instead, enabling HTTP clients to recognize the character set being used automatically.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.4.0, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.8`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.4.0

You can manually upgrade any previous Neptune engine release to this release.

You will not automatically upgrade to this release.

Upgrading to This Release

Amazon Neptune 1.0.4.0 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.4.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.4.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.4.0.R2 (2021-02-24)

As of 2021-02-24, engine version 1.0.4.0.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects Fixed in This Engine Release

- Fixed a bug in Release: 1.0.4.0 (2020-10-12) that allowed connections to Neptune using HTTP or earlier versions of TLS, rather than HTTPS and TLS 1.2.

  Important

  Having to use SSL/TLS for all connections to Neptune can be a breaking change. It affects your connections with the Gremlin console, the Gremlin driver, Gremlin Python, .NET, nodeJs, REST APIs, and also load-balancer connections. If you have been using HTTP or an older TLS version for any or all of these up until now, you must update the relevant client and drivers before installing this patch, and change your code to use HTTPS exclusively.

- Fixed a bug in the CSV bulk load involving labels that end in #.
- Fixed a Gremlin bug where `InternalFailureException` was set as the response code in certain circumstances when a `ConcurrentModificationException` occurred.
- Fixed a Gremlin bug where under certain conditions updating edges or vertices could cause a transient `InternalFailureException`.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.4.0.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.8`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.4.0.R2

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.4.0`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.4.0.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.4.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.4.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

Note

If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.3.0 (2020-08-03)

As of 2020-08-03, engine version 1.0.3.0 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Subsequent Patch Releases for This Release

- Release: 1.0.3.0.R2 (2020-10-12)
- Release: 1.0.3.0.R3 (2021-02-19)

New Features in This Engine Release

- Neptune has introduced a new, alternative query engine (DFE) which can significantly speed up query execution. See The Amazon Neptune alternative query engine (DFE).
- The DFE relies on pre-generated statistics about your Neptune graph data that are managed through new statistics endpoints. See DFE statistics.
- You can now exclude queued load jobs from the list of load IDs returned by the Loader Get-Status API by setting the new `includeQueuedLoads` parameter to FALSE. See Neptune Loader Get-Status request parameters.
- Neptune now supports trailing headers for SPARQL query responses that can contain an error code and message if a request fails after it begins to return response chunks. See Optional HTTP trailing headers for multi-part SPARQL responses.
- Neptune now also lets you enable chunked response encoding for Gremlin queries. As in the SPARQL case, the response chunks have trailing headers that can contain an error code and message if a failure occurs after the query has begun to return response chunks. See Use optional HTTP trailing headers to enable multi-part Gremlin responses.

Improvements in This Engine Release

- You can now provide the size of batch requests to ElasticSearch for full- text searches in Gremlin.
- Improved memory usage for SPARQL GROUP BY queries.
- Added a new Gremlin query optimizer to prune certain unbound filters.
- Increased the maximum time a WebSocket connection authenticated using IAM can stay open, from 36 hours to 10 days.

Defects Fixed in This Engine Release

- Fixed a bug where if you sent an un-encoded URL parameter in a POST request, Neptune returned an HTTP status code of 500 and an `InternalServerErrorException`. Now Neptune returns an HTTP status code of 400 and a `BadRequestException`, with the message: `Failure to process the POST request parameters`.
- Fixed a Gremlin bug where a WebSocket connection failure was not correctly reported.
- Fixed a Gremlin bug involving disappearing sideEffects.
- Fixed a Gremlin bug where the full-text search `batchsize` parameter was not properly supported.
- Fixed a Gremlin bug to handle `toV` and `fromV` individually for each direction on `bothE`.
- Fixed a Gremlin bug involving `Edge pathType` in the `hasLabel` step.
- Fixed a SPARQL bug where join re-ordering with static bindings was not working correctly.
- Fixed a SPARQL UPDATE LOAD bug where an unavailable Amazon S3 bucket was not correctly reported.
- Fixed a SPARQL bug where an issue with a SERVICE node in a subquery was not correctly reported.
- Fixed a SPARQL bug in which queries containing nested FILTER EXISTS or FILTER NOT EXISTS conditions were not being properly evaluated.
- Fixed a SPARQL bug to correctly handle duplicate generated bindings when calling SPARQL Service endpoints through generate queries.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.3.0, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.3`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.3.0

You can manually upgrade any previous Neptune engine release to this release.

If your cluster has its `AutoMinorVersionUpgrade` parameter set to `True`, your cluster will be upgraded to this engine release automatically two to three weeks after the date of this release, during a maintenance window.

Upgrading to This Release

Amazon Neptune 1.0.3.0 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.3.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.3.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> ```

```
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.3.0.R3 (2021-02-19)

As of 2021-02-19, engine version 1.0.3.0.R3 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects Fixed in This Engine Release

- Fixed a bug in the CSV bulk load involving labels that end in #.
- Fixed a Gremlin bug that could cause missing results for a specific pattern of queries that use the `as()` step.
- Fixed a Gremlin bug that could cause errors when using the `project()` step nested inside another step such as `union()`.
- Fixed a Gremlin bug in string traversal execution in the experimental DFE engine when a terminal method like `toList()` is used.
- Fixed a Gremlin bug that fails to close a transaction when using the `iterate()` step in a string query.
- Fixed a Gremlin bug that could cause queries using the `is(P.gte(0))` pattern to throw an exception under certain conditions.
- Fixed a Gremlin bug where `InternalFailureException` was set as the response code in certain circumstances when a `ConcurrentModificationException` occurred.
- Fixed a Gremlin bug where under certain conditions updating edges or vertices could cause a transient `InternalFailureException`.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.3.0.R3, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.8`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.3.0.R3

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.3.0`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.3.0.R3 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.3.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.3.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

Note

If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.3.0.R2 (2020-10-12)

As of 2020-10-12, engine version 1.0.3.0.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Improvements in This Engine Release

- Improved performance for the Gremlin `properties()` step.
- Added details about `BindOp` and `MultiplexerOp` in explain and profile reports.
- For SPARQL query responses, added `charset` to the Content-Type header, enabling HTTP clients to recognize the charset being used automatically.

Defects Fixed in This Engine Release

- Fixed a SPARQL bug where `CancellationException` was not handled.
- Fixed a SPARQL bug where queries containing nested optionals did not work correctly.
- Fixed a SPARQL bug in LOAD where a `ConcurrentModificationException` could cause a query to hang.
- Fixed a SPARQL bug that prevented query responses from being gzip-compressed.
- Fixed a Gremlin bug in the `groupBy()` step.
- Fixed a Gremlin bug related to the use of an `aggregate()` step inside a `local()` step.
- Fixed a Gremlin bug related to using `bothE()` followed by a predicate that uses aggregate values.
- Fixed a Gremlin bug related to using the `bothE()` step with the `repeat()` step.
- Fixed a potential Gremlin memory leak related to the `both()` step.
- Fixed a bug where request metrics were missing because an endpoint ending in '/' was not being handled correctly.

- Fixed a bug that could raise a `ThrottlingException` even when the request queue is not full.
- Fixed a bug in fetching load status when a load fails for a reason such as `LOAD_DATA_FAILED_DUE_TO_FEED_MODIFIED_OR_DELETE`.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.3.0.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.3`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.3.0.R2

You can manually upgrade any previous Neptune engine release to this release.

If your cluster has its `AutoMinorVersionUpgrade` parameter set to `True`, your cluster will be upgraded to this engine release automatically two to three weeks after the date of this release, during a maintenance window.

Upgrading to This Release

Amazon Neptune 1.0.3.0.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.3.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
```

```
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.3.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.2 (2020-03-09)

As of 2020-03-09, engine version 1.0.2.2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Subsequent Patch Releases for This Release

- Release: 1.0.2.2.R2 (2020-04-02)
- Release: 1.0.2.2.R3 (2020-07-22)
- Release: 1.0.2.2.R4 (2020-07-23)
- Release: 1.0.2.2.R5 (2020-10-12)
- Release: 1.0.2.2.R6 (2021-02-19)

Improvements in This Engine Release

- Added information to the status API about transactions that are being rolled back. See Instance Status.
- Upgraded the version of Apache TinkerPop to 3.4.3.

Version 3.4.3 is backwards compatible with the previous version supported by Neptune (3.4.1). It does introduce one minor change in behavior: Gremlin no longer returns an error when you try to close a session that does not exist (see Prevent error when closing sessions that don't exist).

- Removed performance bottlenecks in execution of Gremlin full-text search steps.

Defects Fixed in This Engine Release

- Fixed a SPARQL bug in the handling of empty graph patterns in queries.
- Fixed a SPARQL bug in the handling of unencoded semicolons in URL-encoded queries.
- Fixed a Gremlin bug in the handling of repeated vertices in the `Union` step.
- Fixed a Gremlin bug that caused some queries with a `.simplePath()` or `.cyclicPath()` inside a `.repeat()` to return incorrect results.
- Fixed a Gremlin bug that caused `.project()` to return incorrect results if its child traversal returned no solutions.
- Fixed a Gremlin bug where errors from read-write conflicts raised an `InternalFailureException` rather than a `ConcurrentModificationException`.
- Fixed a Gremlin bug that caused `.group().by(...).by(values("property"))` failures.
- Fixed Gremlin bugs in the profile output for full-text-search steps.
- Fixed a resource leak in Gremlin sessions.
- Fixed a bug that prevented the status API from reporting the correct orderable version in some cases.
- Fixed a bulk loader bug that allowed a URL to a location other than Amazon S3 to be used as the source in a bulk load request.
- Fixed a bulk loader bug in the detailed load status.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.2, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.3`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.2

You can manually upgrade any previous Neptune engine release to this release.

If your cluster has its `AutoMinorVersionUpgrade` parameter set to `True`, your cluster will be upgraded to this engine release automatically two to three weeks after the date of this release, during a maintenance window.

Upgrading to This Release

Amazon Neptune 1.0.2.2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.2.2 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.2.R6 (2021-02-19)

As of 2021-02-19, engine version 1.0.2.2.R6 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug where `InternalFailureException` was set as the response code in certain circumstances when a `ConcurrentModificationException` occurred.
- Fixed a Gremlin bug where under certain conditions updating edges or vertices could cause a transient `InternalFailureException`.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.2.R6, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.8`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.2.R6

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.2.2`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.2.2.R6 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.2.2 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot

that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.2.R5 (2020-10-12)

As of 2020-10-12, engine version 1.0.2.2.R5 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Improvements in This Engine Release

- Improved performance for the Gremlin `properties()` step.
- Added details about `BindOp` and `MultiplexerOp` in explain and profile reports.
- For SPARQL query responses, added `charset` to the Content-Type header, enabling HTTP clients to recognize the charset being used automatically.

Defects Fixed in This Engine Release

- Fixed a SPARQL bug where `CancellationException` was not handled.

- Fixed a SPARQL bug where queries containing nested optionals did not work correctly.
- Fixed a SPARQL bug in LOAD where a `ConcurrentModificationException` could cause a query to hang.
- Fixed a SPARQL bug that prevented query responses from being gzip-compressed.
- Fixed a Gremlin bug in the `groupBy()` step.
- Fixed a Gremlin bug related to the use of an `aggregate()` step inside a `local()` step.
- Fixed a Gremlin bug related to using `bothE()` followed by a predicate that uses aggregate values.
- Fixed a Gremlin bug related to using the `bothE()` step with the `repeat()` step.
- Fixed a potential Gremlin memory leak related to the `both()` step.
- Fixed a bug where request metrics were missing because an endpoint ending in '/' was not being handled correctly.
- Fixed a bug that could raise a `ThrottlingException` even when the request queue is not full.
- Fixed a bug in fetching load status when a load fails for a reason such as `LOAD_DATA_FAILED_DUE_TO_FEED_MODIFIED_OR_DELETE`.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.2.R5, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.3`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.2.R5

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.2.2`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.2.2.R5 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.2.2 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.2.R4 (2020-07-23)

As of 2020-07-23, engine version 1.0.2.2.R4 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Improvements in This Engine Release

- Improved memory usage by releasing unused memory back to the operating system more frequently.
- Also improved memory usage for SPARQL GROUP BY queries.

- Increased the maximum time a WebSocket connection can stay open that is authenticated using IAM, from 36 hours to 10 days.
- Added the `BufferCacheHitRatio` CloudWatch metric, which can be useful in diagnosing query latency and tuning instance types. See Neptune metrics.

Defects Fixed in This Engine Release

- Fixed a bug in closing idle or expired IAM WebSocket connections. Neptune now sends a close frame before closing the connection.
- Fixed a SPARQL bug in the evaluation of queries containing nested FILTER EXISTS and/or FILTER NOT EXISTS conditions.
- Fixed a SPARQL query termination bug that caused blocked threads on the server under certain extreme conditions.
- Fixed a Gremlin bug involving Edge pathType in the `hasLabel` step.
- Fixed a Gremlin bug to handle `toV` and `fromV` individually for each direction on `bothE`.
- Fixed a Gremlin bug involving disappearing sideEffects.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.2.R4, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.3`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.2.R4

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.2.2`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.2.2.R4 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.2.2 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.2.R3 (2020-07-22)

Engine release 1.0.2.2.R3 was incorporated into engine release 1.0.2.2.R4.

Amazon Neptune Engine Version 1.0.2.2.R2 (2020-04-02)

As of 2020-04-02, engine version 1.0.2.2.R2 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Improvements in This Engine Release

- You can now queue up to 64 bulk-load jobs, rather than having to wait for one to finish before initiating the next one. You can also make execution of a queued load request contingent on the successful completion of one or more previously queued load jobs using the `dependencies` parameter of the `load` command. See Neptune Loader Command.
- Full-text-search output can now be sorted (see Full-text search parameters).
- There is now a DB cluster parameter for invoking Neptune streams, and the feature has been moved out of Lab Mode. See Enabling Neptune Streams.

Defects Fixed in This Engine Release

- Fixed a stochastic failure in server startup which delayed instance creation.
- Fixed an optimizer issue where `BIND` statements in the query made the optimizer start out with unselective patterns in join-order planning.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.2.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.3`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.2.R2

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.2.2`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.2.2.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations

on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.2.2 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.2 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.1 (2019-11-22)

Subsequent Patch Releases for This Release

- Release: 1.0.2.1.R6 (2020-04-22)
- Release: 1.0.2.1.R5 (2020-04-22) *This patch release was not deployed.*
- Release: 1.0.2.1.R4 (2019-12-20)
- Release: 1.0.2.1.R3 (2019-12-12)

- Release: 1.0.2.1.R2 (2019-11-25)

New Features in This Engine Release

- Added full-text search capabilities through integration with the Amazon OpenSearch Service. See Neptune full text search
- Added the option using lab mode to create a fourth index (an OSGP index) for large numbers of predicates. See OSGP index.
- Added a *details* mode to SPARQL Explain. See Using SPARQL `explain` and Details mode output for details.
- Added lab mode information to the engine status report. See Instance Status for details.
- DB Cluster snapshots can now be copied across AWS Regions. See Copying a Snapshot.

Improvements in This Engine Release

- Improved performance when handling a large number of predicates.
- Enhanced query optimization. While this should be entirely transparent to customers, we encourage you to test your applications before upgrading to ensure that they behave as expected.
- Minor enhancements to error reporting.
- Added optimizations for Gremlin `.project()` and `.identity()` steps.
- Added optimizations for non-terminal Gremlin `.union()` cases.
- Added native support for Gremlin `.path().by()` traversals.
- Added native support for Gremlin `.coalesce()`.
- Further optimization of bulk write.
- We now require that HTTPS connections use at least TLS version 1.2 or higher, to prevent outdated/insecure ciphers being used.

Defects Fixed in This Engine Release

- Fixed a Gremlin `addE()` inner traversal handling bug.
- Fixed a Gremlin bug caused by AST annotations leaking from child traversals to the parent.
- Fixed a bug that occurred in Gremlin when `.otherV()` was called after `select()`.

- Fixed a Gremlin bug that caused some `.hasLabel()` steps to fail if they appeared after a `bothE()` step.
- Made minor fixes for Gremlin .sum() and .project().
- Fixed a bug in processing SPARQL queries that lack a closing brace.
- Fixed some minor bugs in SPARQL Explain.
- Fixed a bug in the handling of concurrent get load status requests.
- Reduced memory used for executing some Gremlin traversals with `.project()` steps.
- Fixed numeric comparisons of special values in SPARQL. See Standards Compliance.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.1, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.1`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.1

You can manually upgrade any previous Neptune engine release to this release.

You will not automatically upgrade to this release.

Upgrading to This Release

Amazon Neptune 1.0.2.1 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
```

```
    --engine-version 1.0.2.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that

begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.1.R6 (2020-04-22)

As of 2020-04-22, engine version 1.0.2.1.R6 is being generally deployed. Please note that it takes several days for a new release to become available in every region.

Defects Fixed in This Engine Release

- Fixed a bug where `ConcurrentModificationConflictException` and `TransactionException` were not converted into a `NeptuneGremlinException`, causing `InternalFailureException` to be returned to customers.
- Fixed a bug where Neptune reported its status as healthy before the server was completely ready.
- Fixed a bug where dictionary and user transaction commits were out of order when two `value->id` mappings were being inserted concurrently.
- Fixed a bug in load-status serialization.

- Fixed a Gremlin sessions bug.
- Fixed a bug where Neptune failed to throw an exception when the server failed to start.
- Fixed a bug where Neptune failed to send a Web socket close frame before closing the channel.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.1.R6, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.1`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.1.R6

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.2.1`.

You can manually upgrade any previous Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.2.1.R6 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.2.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
```

```
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.1.R5 (2020-04-22)

Engine release 1.0.2.1.R5 was never deployed.

Amazon Neptune Engine Version 1.0.2.1.R4 (2019-12-20)

Improvements in This Engine Release

- Neptune now tries always to place any full-text-search call first in the execution pipeline. This reduces the volume of calls to OpenSearch, which can significantly improve performance. See Full-text-search query execution.
- Neptune now raises an `IllegalArgumentException` if you try to access a non-existent property, vertex, or edge. Previously, Neptune raised an `UnsupportedOperationException` in that situation.

  For example, if you try to add an edge referencing a nonexistent vertex, you will now raise an `IllegalArgumentException`.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug where a `union` traversal inside a `project-by` does not return results or returns incorrect results.
- Fixed a Gremlin bug that caused nested `.project().by()` steps to return incorrect results.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.1.R4, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.1`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.1.R4

You can manually upgrade any previous Neptune engine release to this release.

However, **automatic updating to this release is not supported**.

Upgrading to This Release

Amazon Neptune 1.0.2.1.R4 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.2.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.1.R3 (2019-12-12)

Defects Fixed in This Engine Release

- Fixed a bug where the `OSGP` index was disabled even though the feature was correctly enabled using inLab Mode using the `ObjectIndex` value in the `neptune_lab_mode` parameter.
- Fixed a bug that affected Gremlin queries with a `.fold()` inside a `.project().by()` step. For example, it caused the following query to return incomplete results:

```
g.V().project("a").by(valueMap().fold())
```

- Fixed a performance bottleneck in bulk loads of RDF data.
- Fixed a bug that caused a crash on replicas when streams were enabled and the replica was restarted before the primary.
- Fixed a bug where rotated SSL certificates on instances were not picked up without an instance restart.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.1.R3, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.1`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.1.R3

You can manually upgrade any previous Neptune engine release to this release.

However, **automatic updating to this release is not supported**.

Upgrading to This Release

Amazon Neptune 1.0.2.1.R3 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.2.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```
>
> If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.1.R2 (2019-11-25)

Defects Fixed in This Engine Release

- Fixed a bug affecting all `project().by()` queries with non round-robin by-traversals and non `path()` by-traversals.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.1.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.1`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.1.R2

You can manually upgrade any previous Neptune engine release to this release.

However, **automatic updating to this release is not supported**.

Upgrading to This Release

Amazon Neptune 1.0.2.1.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.2.1 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
```

```
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.1 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.0 (2019-11-08)

IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED

Starting from 2020-05-19, no new instances using this engine version will be created.

This engine version is now superseded by version 1.0.2.1, which contains all the bug fixes in this version as well as additional features such as full-text search integration, OSGP index support, and database snapshot cluster copy across AWS Regions.

Starting June 1, 2020, Neptune will automatically upgrade any cluster running this engine version to the latest patch of version 1.0.2.1 during the next maintenance window. You can upgrade manually before then, as described here.

If you have any issues with the upgrade, please contact us through AWS Support or the AWS Developer Forums.

Subsequent Patch Releases for This Release

- Release: 1.0.2.0.R3 (2020-05-05)
- Release: 1.0.2.0.R2 (2019-11-21)

New Features in This Engine Release

In addition to maintenance updates, this release adds new functionality to support more than one engine version at a time (see Maintaining your Amazon Neptune DB Cluster).

As a result, the numbering of engine releases has changed (see Version numbering before engine release 1.3.0.0).

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.0, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.1`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.0

You can manually upgrade any previous Neptune engine release to this release.

You will not automatically upgrade to this release.

Upgrading to This Release

Amazon Neptune 1.0.2.0 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.2.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.0.R3 (2020-05-05)

IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED

Starting from 2020-05-19, no new instances using this engine version will be created.

This engine version is now superseded by version 1.0.2.1, which contains all the bug fixes in this version as well as additional features such as full-text search integration, OSGP index support, and database snapshot cluster copy across AWS Regions.

Starting June 1, 2020, Neptune will automatically upgrade any cluster running this engine version to the latest patch of version 1.0.2.1 during the next maintenance window. You can upgrade manually before then, as described here.

If you have any issues with the upgrade, please contact us through AWS Support or the AWS Developer Forums.

Defects Fixed in This Engine Release

- Fixed a bug where `ConcurrentModificationConflictException` and `TransactionException` were reported as generic `InternalFailureExceptions`.
- Fixed bugs in health checks that caused frequent restarts of the server during start up.
- Fixed a bug where data was not visible on replicas because commits were out of order under certain conditions.

- Fixed a bug in load-status serialization where a load failed from a lack of Amazon S3 access permissions.
- Fixed a resource leak in Gremlin sessions.
- Fixed a bug in health check that hid the unhealthy status on start-up of components managing IAM authentication.
- Fixed a bug where Neptune failed to send a WebSocket close frame before closing the channel.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.0.R3, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.1`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.0.R3

Your cluster will be upgraded to this patch release automatically during your next maintenance window if you are running engine version `1.0.2.0`.

You can manually upgrade any earlier Neptune engine release to this release.

Upgrading to This Release

Amazon Neptune 1.0.2.0.R3 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.2.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:

```
  We're sorry, your request to modify DB cluster (cluster identifier) has
failed.
  Cannot modify engine version because instance (instance identifier) is
  running on an old configuration. Apply any pending maintenance actions on the
instance before
  proceeding with the upgrade.
```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.2.0.R2 (2019-11-21)

IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED

Starting from 2020-05-19, no new instances using this engine version will be created.

This engine version is now superseded by version 1.0.2.1, which contains all the bug fixes in this version as well as additional features such as full-text search integration, OSGP index support, and database snapshot cluster copy across AWS Regions.

Starting June 1, 2020, Neptune will automatically upgrade any cluster running this engine version to the latest patch of version 1.0.2.1 during the next maintenance window. You can upgrade manually before then, as described here.

If you have any issues with the upgrade, please contact us through AWS Support or the AWS Developer Forums.

Defects Fixed in This Engine Release

- Improved the caching strategy for dirty pages on the server so that `FreeableMemory` recovers faster when the server enters a low-memory state.
- Fixed a bug that could cause a race condition and crash when many concurrent load status and/or start load requests are processed on the server.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.2.0.R2, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.1`
- *SPARQL version:* `1.1`

Upgrade Paths to Engine Release 1.0.2.0.R2

You can manually upgrade any previous Neptune engine release to this release.

However, **automatic updating to this release is not supported**.

Upgrading to This Release

Amazon Neptune 1.0.2.0.R2 is now generally available.

If a DB cluster is running an engine version from which there is an upgrade path to this release, it is eligible to be upgraded now. You can upgrade any eligible cluster using the DB cluster operations on the console or by using the SDK. The following CLI command will upgrade an eligible cluster immediately:

For Linux, OS X, or Unix:

```
aws neptune modify-db-cluster \
    --db-cluster-identifier (your-neptune-cluster) \
    --engine-version 1.0.2.0 \
    --apply-immediately
```

For Windows:

```
aws neptune modify-db-cluster ^
    --db-cluster-identifier (your-neptune-cluster) ^
    --engine-version 1.0.2.0 ^
    --apply-immediately
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on those instances, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using the DB cluster.

Always test before you upgrade

When a new major or minor Neptune engine version is released, always test your Neptune applications on it first before upgrading to it. Even a minor upgrade could introduce new features or behavior that would affect your code.

Start by comparing the release notes pages from your current version to those of the targeted version to see if there will be changes in query language versions or other breaking changes.

The best way to test a new version before upgrading your production DB cluster is to clone your production cluster so that the clone is running the new engine version. You can then run queries on the clone without affecting the production DB cluster.

Always create a manual snapshot before you upgrade

Before performing an upgrade, we strongly recommend that you always create a manual snapshot of your DB cluster. Having an automatic snapshot only offers short-term protection, whereas a manual snapshot remains available until you explicitly delete it.

In certain cases Neptune creates a manual snapshot for you as a part of the upgrade process, but you should not rely on this, and should create your own manual snapshot in any case.

When you are certain that you won't need to revert your DB cluster to its pre-upgrade state, you can explicitly delete the manual snapshot that you created yourself, as well as the manual snapshot that Neptune might have created. If Neptune creates a manual snapshot, it will have a name that begins with `preupgrade`, followed by the name of your DB cluster, the source engine version, the target engine version, and the date.

> **Note**
>
> If you are trying to upgrade while a pending action is in process, you may encounter an error such as the following:
>
> ```
> We're sorry, your request to modify DB cluster (cluster identifier) has failed.
> Cannot modify engine version because instance (instance identifier) is
> running on an old configuration. Apply any pending maintenance actions on the instance before
> proceeding with the upgrade.
> ```

If you encounter this error, wait for the pending action to finish, or trigger a maintenance window immediately to let the previous upgrade complete.

For more information about upgrading your engine version, see Maintaining your Amazon Neptune DB Cluster. If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Amazon Neptune Engine Version 1.0.1.2 (2020-06-10)

IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED

Starting from 2021-04-27, no new instances using this engine version will be created.

Improvements in This Engine Release

- Neptune now raises an `IllegalArgumentException` if you try to access a non-existent property, vertex, or edge. Previously, Neptune raised an `UnsupportedOperationException` in that situation.

  For example, if you try to add an edge referencing a nonexistent vertex, you will now raise an `IllegalArgumentException`.

Defects Fixed in This Engine Release

- Fixed a bug where dictionary and user transaction commits were out of order when two `value->id` mappings were being inserted concurrently.
- Fixed a bug in load-status serialization.
- Fixed a stochastic failure in server startup which delayed instance creation.
- Fixed a cursor leak.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.1.2, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.4.1`
- *SPARQL version:* `1.1`

Amazon Neptune Engine Version 1.0.1.1 (2020-06-26)

IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED

Starting from 2021-04-27, no new instances using this engine version will be created.

Defects Fixed in This Engine Release

- Fixed a bug where commits were out of order when inserted concurrently.
- Fixed a bug in load-status serialization.
- Fixed a stochastic failure in server startup which delayed instance creation.
- Fixed a memory leak.

Query-Language Versions Supported in This Release

Before upgrading a DB cluster to version 1.0.1.1, make sure that your project is compatible with these query-language versions:

- *Gremlin version:* `3.3.2`
- *SPARQL version:* `1.1`

Amazon Neptune Engine Version 1.0.1.0 (2019-07-02)

IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED

Starting from 2021-04-27, no new instances using this engine version will be created.

Amazon Neptune Engine Updates 2019-10-31

Version: 1.0.1.0.200502.0

IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED

No new instances using this engine version will be created, beginning 2021-04-27.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug in the serialization of the `tree()` step's response when clients connect to Neptune using `traversal().withRemote(...)` (in other words, using GLV bytecode).

  This release addresses an issue in which clients who connected to Neptune using `traversal().withRemote(...)` received an invalid response to Gremlin queries that contained a `tree()` step.
- Fixed a SPARQL bug in `DELETE WHERE LIMIT` queries, in which the query termination process would hang because of a race condition, causing the query to time out.

Amazon Neptune Engine Updates 2019-10-15

Version: 1.0.1.0.200463.0

IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED

No new instances using this engine version will be created, beginning 2021-04-27.

New Features in This Engine Release

- Added a Gremlin Explain/Profile feature (see Analyzing Neptune query execution using Gremlin `explain`).
- Added Support for Gremlin script-based sessions to enable executing multiple Gremlin traversals in a single transaction.
- Added support for the SPARQL Federated Query extension in Neptune (see SPARQL 1.1 Federated Query and SPARQL federated queries in Neptune using the `SERVICE` extension).
- Added a feature letting you inject your own `queryId` into a Gremlin or SPARQL query, either through an HTTP URL parameter or through a SPARQL `queryId` query hint (see Inject a Custom ID Into a Neptune Gremlin or SPARQL Query).
- Added a Lab Mode feature to Neptune that can allow you to try out upcoming features which are not yet ready to be used in production.
- Added an upcoming Neptune streams feature that reliably logs every change made to your database into a stream that persists for a week. This feature is available only in Lab Mode.
- Updated the formal semantics for concurrent transactions (see Transaction Semantics in Neptune). This feature provides industry-standard guarantees around concurrency.

By default, these transaction semantics are enabled. In some scenarios, this feature may change current load behavior and reduce load performance. You can use the DB Cluster `neptune_lab_mode` parameter to revert to the previous semantics by including `ReadWriteConflictDetection=disabled` in the parameter value.

Improvements in This Engine Release

- Improved the Instance Status API by reporting what version of TinkerPop and what version of SPARQL the engine is using.
- Improved Gremlin subgraph operator performance.
- Improved the performance of Gremlin response serialization.
- Improved the performance in the Gremlin Union step.
- Improved the latency of simple SPARQL queries.

Defects Fixed in This Engine Release

- Fixed a Gremlin bug where timeout was incorrectly being returned as an internal failure.
- Fixed a SPARQL bug in which ORDER BY over a partial set of variables caused an Internal Server Error.

Amazon Neptune Engine Updates 2019-09-19

IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED

No new instances using this engine version will be created, beginning 2021-04-27.

Version: 1.0.1.0.200457.0

Amazon Neptune 1.0.1.0.200457.0 is generally available. All new Neptune DB clusters, including those restored from snapshots, will be created in Neptune 1.0.1.0.200457.0 after the engine update is complete for that Region.

Existing clusters can be upgraded to this release immediately using the DB cluster operations on the console or by using the SDK. You can use the following CLI command to upgrade a DB cluster:

```
aws neptune apply-pending-maintenance-action \
```

```
    --apply-action system-update \
    --opt-in-type immediate \
    --resource-identifier arn:aws-eusc:rds:<region>:<account
number>:<resourcetype>:<name>
```

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on all instances in a DB cluster, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using your DB cluster or clusters. You can view or change your maintenance window settings on the Neptune console.

If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Defects Fixed in This Engine Release

- Fixed a Gremlin correctness issue introduced in the previous engine release (1.0.1.0.200369.0) by removing the performance improvement to conjunctive predicate handling that caused it.
- Fixed a SPARQL bug that caused queries with `DISTINCT` and a single pattern wrapped into `OPTIONAL` to generate an `InternalServerError`.

Amazon Neptune Engine Updates 2019-08-13

IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED

Starting from 2021-04-27, no new instances using this engine version will be created.

New Features in This Engine Release

- Added an `OVERSUBSCRIBE` option to the `parallelism` parameter of the Neptune Loader Command, which causes the Neptune bulk loader to use all available threads and resources.

Improvements in This Engine Release

- Improved performance of SPARQL filters containing simple logical OR expressions.
- Improved Gremlin performance in handling conjunctive predicates.

Defects Fixed in This Engine Release

- Fixed a SPARQL bug preventing subtraction of an `xsd:duration` from an `xsd:date`.
- Fixed a SPARQL bug causing incomplete results from static inlining in the presence of a `UNION`.
- Fixed a SPARQL bug in query cancellation.
- Fixed a Gremlin bug causing overflow during type promotion.
- Fixed a Gremlin bug in the handling of vertex elements in `addE().from().to()` steps.
- Fixed a Gremlin bug (released 2019-07-26 in Engine version 1.0.1.0.200366.0) involving the handling of NaN doubles and floats in single-cardinality inserts.
- Fixed a bug in generating query plans involving property based searches.

Amazon Neptune Engine Updates 2019-07-26

Version: 1.0.1.0.200366.0

IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED

No new instances using this engine version will be created, beginning 2021-04-27.

New Features in This Engine Release

- Upgraded to TinkerPop 3.4.1 (see TinkerPop Upgrade Information, and TinkerPop 3.4.1 Change Log).

  For Neptune customers, these changes provide new functionality and improvements, such as:

  - `GraphBinary` is now available as a serialization format.
  - A keep-alive bug that caused memory leaks in the TinkerPop Java driver has been fixed, so a work-around is no longer necessary.

  However, in a few cases, they may affect existing Gremlin code in Neptune. For example:

  - `valueMap()` now returns a `Map<Object,Object>` instead of a `Map<String,Object>`.
  - Inconsistent behavior of the `within()` step was fixed so it would work consistently with other steps. Previously, types had to match for comparisons to work. Now, numbers of different types can be accurately compared. For example, `33` now compares as equal to `33L`, which it did not before.

- A bug in `ReducingBarrierStep` was fixed, so it now returns no value if no elements are available for output.
- The order of `select()` scopes changed (the order is now `maps`, `side-effects`, `paths`). This changes the results of the rare queries that combine `side-effects` and `select` with the same key name for `side-effects` as for `select`.
- `bulkSet()` is now part of the GraphSON protocol. Queries that end with `toBulkSet()` won't work with older clients.
- One parameterization of the `Submit()` step was removed from the 3.4 client.

Many other changes introduced in TinkerPop 3.4 do not affect current Neptune behavior. For example, Gremlin `io()` was added as a step to `Traversal` and is now deprecated in `Graph`, but was never enabled in Neptune.

- Added support for single cardinality vertex properties to the bulk loader for Gremlin, for loading property graph data.
- Added an option to overwrite the existing values for a single-cardinality property in the bulk loader.
- Added the ability to retrieve the status of a Gremlin query, and to cancel a Gremlin query.
- Added a query hint for SPARQL query timeouts.
- Added the ability to see the instance role in the status API (see Instance Status).
- Added support for database cloning (see Database Cloning in Neptune).

Improvements in This Engine Release

- Improved the SPARQL Query Explanation to show graph variables from FROM clauses.
- Improved performance for SPARQL in filters, equal filters, VALUES clauses, and range counts.
- Improved performance for Gremlin step ordering.
- Improved performance for Gremlin `.repeat.dedup` traversals.
- Improved the performance of Gremlin `valueMap()` and `path().by()` traversals.

Defects Fixed in This Engine Release

- Fixed multiple issues with SPARQL property paths including operation with named graphs.
- Fixed an issue with SPARQL CONSTRUCT queries causing memory issues.

- Fixed an issue with the RDF Turtle parser and local names.
- Fixed an issue to correct error messages displayed to users.
- Fixed an issue with Gremlin `repeat()...drop()` traversals.
- Fixed an issue with the Gremlin `drop()` step.
- Fixed an issue with Gremlin label filters.
- Fixed an issue with Gremlin query timeouts.

Amazon Neptune Engine Updates 2019-07-02

IMPORTANT: THIS ENGINE VERSION IS NOW DEPRECATED

No new instances using this engine version will be created, beginning 2021-04-27.

Defects Fixed in This Engine Release

- Fixed a bug that caused certain patterns with a property name and value bound not to be optimized.

Earlier Neptune Engine Releases

Topics

- Amazon Neptune Engine Updates 2019-06-12
- Amazon Neptune Engine Updates 2019-05-01
- Amazon Neptune Engine Updates 2019-01-21
- Amazon Neptune Engine Updates 2018-11-19
- Amazon Neptune Engine Updates 2018-11-08
- Amazon Neptune Engine Updates 2018-10-29
- Amazon Neptune Engine Updates 2018-09-06
- Amazon Neptune Engine Updates 2018-07-24
- Amazon Neptune Engine Updates 2018-06-22

Amazon Neptune Engine Updates 2019-06-12

Version: 1.0.1.0.200310.0

Amazon Neptune 1.0.1.0.200310.0 is generally available. All new Neptune DB clusters, including those restored from snapshots, will be created in Neptune 1.0.1.0.200310.0 after the engine update is complete for that Region.

Existing clusters can be upgraded to this release immediately using the DB cluster operations on the console or by using the SDK. You can use the following CLI command to upgrade a DB cluster to this release immediately:

```
aws neptune apply-pending-maintenance-action \
    --apply-action system-update \
    --opt-in-type immediate \
    --resource-identifier arn:aws-eusc:rds:<region>:<account
 number>:<resourcetype>:<name>
```

Neptune DB clusters will automatically be upgraded to engine release 1.0.1.0.200310.0 during system maintenance windows. The timing of when updates are applied depends on the Region and maintenance window setting for the DB cluster, as well as on the type of update.

> **Note**
>
> The instance maintenance window does not apply to engine updates.

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on all instances in a DB cluster, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using your DB cluster or clusters. You can view or change your maintenance window settings on the Neptune console.

If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Improvements

- Fixes a bug where concurrent insertion and dropping of an edge can result in multiple edges with the same id.
- Other minor fixes and improvements.

Amazon Neptune Engine Updates 2019-05-01

Version: 1.0.1.0.200296.0

Amazon Neptune 1.0.1.0.200296.0 is generally available. All new Neptune DB clusters, including those restored from snapshots, will be created in Neptune 1.0.1.0.200296.0 after the engine update is complete for that Region.

Existing clusters can be upgraded to this release immediately using the DB cluster operations on the console or by using the SDK. You can use the following CLI command to upgrade a DB cluster to this release immediately:

```
aws neptune apply-pending-maintenance-action \
    --apply-action system-update \
    --opt-in-type immediate \
    --resource-identifier arn:aws-eusc:rds:<region>:<account
 number>:<resourcetype>:<name>
```

Neptune DB clusters will automatically be upgraded to engine release 1.0.1.0.200296.0 during system maintenance windows. The timing of when updates are applied depends on the Region and maintenance window setting for the DB cluster, as well as on the type of update.

> **Note**
>
> The instance maintenance window does not apply to engine updates.

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on all instances in a DB cluster, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using your DB cluster or clusters. You can view or change your maintenance window settings on the Neptune console.

If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Improvements

- Added the new `explain` feature to Neptune SPARQL queries to help you visualize the query plan and take steps to optimize it if necessary. For information, see SPARQL `explain`.
- Improved SPARQL performance and reporting in various ways.
- Improved Gremlin performance and behavior in various ways.
- Improved the timing-out of long-running `drop( )` queries.

- Improved the performance of `otherV( )` queries.
- Added two fields to the information returned when you query the Neptune health status of a DB cluster or instance, namely the engine version number and the cluster or instance start time. See Instance Status.
- The Neptune loader `Get-Status` API now returns a `startTime` field that records when a load job started.
- The loader command now takes an optional `parallelism` parameter that lets you restrict the number of threads the loader uses.

Amazon Neptune Engine Updates 2019-01-21

Version: 1.0.1.0.200267.0

Amazon Neptune 1.0.1.0.200267.0 is generally available. All new Neptune DB clusters, including those restored from snapshots, will be created in Neptune 1.0.1.0.200267.0 after the engine update is complete for that Region.

Existing clusters can be upgraded to this release immediately using the DB cluster operations on the console or by using the SDK. You can use the following CLI command to upgrade a DB cluster to this release immediately:

```
aws neptune apply-pending-maintenance-action \
    --apply-action system-update \
    --opt-in-type immediate \
    --resource-identifier arn:aws-eusc:rds:<region>:<account
 number>:<resourcetype>:<name>
```

Neptune DB clusters will automatically be upgraded to engine release 1.0.1.0.200267.0 during system maintenance windows. The timing of when updates are applied depends on the Region and maintenance window setting for the DB cluster, as well as on the type of update.

> **Note**
>
> The instance maintenance window does not apply to engine updates.

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on all instances in a DB cluster, so you will experience downtime ranging from 20–30

seconds to several minutes, after which you can resume using your DB cluster or clusters. You can view or change your maintenance window settings on the Neptune console.

If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Improvements

- Neptune waits longer (within the query timeout specified) for any conflicts to get resolved. This reduces the number of concurrent modification exceptions that need to be handled by the client (see Query Errors).
- Fixed an issue where Gremlin cardinality enforcement sometimes caused the engine to restart.
- Improved Gremlin performance for `emit.times` repeat queries.
- Fixed a Gremlin issue where `repeat.until` was allowing `.emit` solutions through that should have been filtered.
- Improved error handling in Gremlin.

Amazon Neptune Engine Updates 2018-11-19

Version: 1.0.1.0.200264.0

Amazon Neptune 1.0.1.0.200264.0 is generally available. All new Neptune DB clusters, including those restored from snapshots, will be created in Neptune 1.0.1.0.200264.0 after the engine update is complete for that Region.

Existing clusters can be upgraded to this release immediately using the DB cluster operations on the console or by using the SDK. You can use the following CLI command to upgrade a DB cluster to this release immediately:

```
aws neptune apply-pending-maintenance-action \
    --apply-action system-update \
    --opt-in-type immediate \
    --resource-identifier arn:aws-eusc:rds:<region>:<account
 number>:<resourcetype>:<name>
```

Neptune DB clusters will automatically be upgraded to engine release 1.0.1.0.200264.0 during system maintenance windows. The timing of when updates are applied depends on the Region and maintenance window setting for the DB cluster, as well as on the type of update.

> **Note**
>
> The instance maintenance window does not apply to engine updates.

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on all instances in a DB cluster, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using your DB cluster or clusters. You can view or change your maintenance window settings on the Neptune console.

If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Improvements

- Added support for the section called "Query hints".
- Improved error messages for IAM authentication. For more information, see the section called "IAM Errors".
- Improved SPARQL query performance with a high number of predicates.
- Improved SPARQL property path performance.
- Improved Gremlin performance for conditional mutations, such as the `fold().coalesce(unfold(), …)` pattern, when used with `addV()`, `addE()`, and `property()` steps.
- Improved Gremlin performance for `by()` and `sack()` modulations.
- Improved Gremlin performance for `group()` and `groupCount()` steps.
- Improved Gremlin performance for `store()`, `sideEffect()`, and `cap().unfold()` steps.
- Improved support for Gremlin single cardinality properties constraints.
  - Improved enforcement of single cardinality for edge properties and vertex properties marked as single cardinality properties.
  - Introduced an error if additional property values are specified for an existing edge property during Neptune Load jobs.

Amazon Neptune Engine Updates 2018-11-08

Version: 1.0.1.0.200258.0

Amazon Neptune 1.0.1.0.200258.0 is generally available. All new Neptune DB clusters, including those restored from snapshots, will be created in Neptune 1.0.1.0.200258.0 after the engine update is complete for that Region.

Existing clusters can be upgraded to this release immediately using the DB cluster operations on the console or by using the SDK. You can use the following CLI command to upgrade a DB cluster to this release immediately:

```
aws neptune apply-pending-maintenance-action \
    --apply-action system-update \
    --opt-in-type immediate \
    --resource-identifier arn:aws-eusc:rds:<region>:<account
 number>:<resourcetype>:<name>
```

Neptune DB clusters will automatically be upgraded to engine release 1.0.1.0.200258.0 during system maintenance windows. The timing of when updates are applied depends on the Region and maintenance window setting for the DB cluster, as well as on the type of update.

> **Note**
>
> The instance maintenance window does not apply to engine updates.

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on all instances in a DB cluster, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using your DB cluster or clusters. You can view or change your maintenance window settings on the Neptune console.

If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Improvements

- Added support for SPARQL query hints.
- Improved performance for SPARQL FILTER (NOT) Exists queries.
- Improved performance for SPARQL DESCRIBE queries.
- Improved performance for the repeat until pattern in Gremlin.
- Improved performance for adding edges in Gremlin.
- Fixed an issue where SPARQL Update DELETE queries could fail in some cases.

- Fixed an issue for handling timeouts with the Gremlin WebSocket server.

Amazon Neptune Engine Updates 2018-10-29

Version: 1.0.1.0.200255.0

Amazon Neptune 1.0.1.0.200255.0 is generally available. All new Neptune DB clusters, including those restored from snapshots, will be created in Neptune 1.0.1.0.200255.0 after the engine update is complete for that Region.

Existing clusters can be upgraded to this release immediately using the DB cluster operations on the console or by using the SDK. You can use the following CLI command to upgrade a DB cluster to this release immediately:

```
aws neptune apply-pending-maintenance-action \
    --apply-action system-update \
    --opt-in-type immediate \
    --resource-identifier arn:aws-eusc:rds:<region>:<account
 number>:<resourcetype>:<name>
```

Neptune DB clusters will automatically be upgraded to engine release 1.0.1.0.200255.0 during system maintenance windows. The timing of when updates are applied depends on the Region and maintenance window setting for the DB cluster, as well as on the type of update.

> **Note**
>
> The instance maintenance window does not apply to engine updates.

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on all instances in a DB cluster, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using your DB cluster or clusters. You can view or change your maintenance window settings on the Neptune console.

If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Improvements

- Added IAM authentication information to Audit logs.

- Added Support for temporary credentials using IAM Roles and Instance Profiles.
- Added WebSocket connection termination for IAM authentication when permission is revoked or if the IAM user or role is deleted.
- Limited the maximum number of WebSocket connections to 60,000 per instance.
- Improved Bulk Load performance for smaller instance types.
- Improved performance for queries that include the `and()`, `or()`, `not()`, `drop()` operators in Gremlin.
- The NTriples parser now rejects invalid URIs, such as URIs containing whitespace.

Amazon Neptune Engine Updates 2018-09-06

Version: 1.0.1.0.200237.0

Amazon Neptune 1.0.1.0.200237.0 is generally available. All new Neptune DB clusters, including those restored from snapshots, will be created in Neptune 1.0.1.0.200237.0 after the engine update is complete for that Region.

Existing clusters can be upgraded to this release immediately using the DB cluster operations on the console or by using the SDK. You can use the following CLI command to upgrade a DB cluster to this release immediately:

```
aws neptune apply-pending-maintenance-action \
    --apply-action system-update \
    --opt-in-type immediate \
    --resource-identifier arn:aws-eusc:rds:<region>:<account
 number>:<resourcetype>:<name>
```

Neptune DB clusters will automatically be upgraded to engine release 1.0.1.0.200237.0 during system maintenance windows. The timing of when updates are applied depends on the Region and maintenance window setting for the DB cluster, as well as on the type of update.

> **Note**
>
> The instance maintenance window does not apply to engine updates.

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on all instances in a DB cluster, so you will experience downtime ranging from 20–30

seconds to several minutes, after which you can resume using your DB cluster or clusters. You can view or change your maintenance window settings on the Neptune console.

If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Improvements

- Fixed an issue where some `SPARQL COUNT(DISTINCT)` queries failed.
- Fixed an issue where `COUNT`, `SUM`, `MIN` queries with a `DISTINCT` clause would run out of memory.
- Fixed an issue where `BLOB` type data would cause a Neptune Loader job to fail.
- Fixed an issue where duplicate inserts would cause transaction failures.
- Fixed an issue where `DROP ALL` queries could not be cancelled.
- Fixed an issue where Gremlin clients could hang intermittently.
- Updated all error codes for payloads bigger than 150M to be `HTTP 400`.
- Improved performance and accuracy of single-triple-pattern `COUNT()` queries.
- Improved performance of `SPARQL UNION` queries with `BIND` clauses.

Amazon Neptune Engine Updates 2018-07-24

Version: 1.0.1.0.200236.0

Amazon Neptune 1.0.1.0.200236.0 is generally available. All new Neptune DB clusters, including those restored from snapshots, will be created in Neptune 1.0.1.0.200236.0 after the engine update is complete for that Region.

Existing clusters can be upgraded to this release immediately using the DB cluster operations on the console or by using the SDK. You can use the following CLI command to upgrade a DB cluster to this release immediately:

```
aws neptune apply-pending-maintenance-action \
    --apply-action system-update \
    --opt-in-type immediate \
    --resource-identifier arn:aws-eusc:rds:<region>:<account
 number>:<resourcetype>:<name>
```

Neptune DB clusters will automatically be upgraded to engine release 1.0.1.0.200236.0 during system maintenance windows. The timing of when updates are applied depends on the Region and maintenance window setting for the DB cluster, as well as on the type of update.

> **Note**
>
> The instance maintenance window does not apply to engine updates.

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on all instances in a DB cluster, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using your DB cluster or clusters. You can view or change your maintenance window settings on the Neptune console.

If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Improvements

- Updated the SPARQL serialization for the `xsd:string` datatype. xsd:string is no longer included in JSON serialization, which is now consistent with other output formats.
- Fixed handling of `xsd:double/xsd:float` infinity. `-INF`, `NaN`, and `INF` values are now properly recognized and handled in all SPARQL data loader formats, SPARQL 1.1 UPDATE, and SPARQL 1.1 Query.
- Fixed an issue where a Gremlin query with empty string values fail unexpectedly.
- Fixed an issue where Gremlin `aggregate()` and `cap()` on an empty graph fails unexpectedly.
- Fixed an issue where incorrect error responses are returned for Gremlin when the cardinality specification is invalid, e.g. `.property(set,id,'10')` and `.property(single,id,'10')`.
- Fixed an issue where invalid Gremlin syntax was returned as an InternalFailureException.
- Fixed the spelling in `TimeLimitExceeededException` to `TimeLimitExceededException`, in error messages.
- Changed the SPARQL and GREMLIN endpoints respond in a consistent way when no script is supplied.
- Clarified error messages for too many concurrent requests.

Amazon Neptune Engine Updates 2018-06-22

Version: 1.0.1.0.200233.0

Amazon Neptune 1.0.1.0.200233.0 is generally available. All new Neptune DB clusters, including those restored from snapshots, will be created in Neptune 1.0.1.0.200233.0 after the engine update is complete for that Region.

Existing clusters can be upgraded to this release immediately using the DB cluster operations on the console or by using the SDK. You can use the following CLI command to upgrade a DB cluster to this release immediately:

```
aws neptune apply-pending-maintenance-action \
    --apply-action system-update \
    --opt-in-type immediate \
    --resource-identifier arn:aws-eusc:rds:<region>:<account
 number>:<resourcetype>:<name>
```

Neptune DB clusters will automatically be upgraded to engine release 1.0.1.0.200233.0 during system maintenance windows. The timing of when updates are applied depends on the Region and maintenance window setting for the DB cluster, as well as on the type of update.

> **Note**
>
> The instance maintenance window does not apply to engine updates.

Updates are applied to all instances in a DB cluster simultaneously. An update requires a database restart on all instances in a DB cluster, so you will experience downtime ranging from 20–30 seconds to several minutes, after which you can resume using your DB cluster or clusters. You can view or change your maintenance window settings on the Neptune console.

If you have any questions or concerns, the AWS Support team is available on the community forums and through AWS Premium Support.

Improvements

- Fixed an issue where a large number of bulk load requests are issued in quick succession results in an error.

- Fixed a data-dependent issue where a query could fail with an InternalServerError. The following example shows the type of query affected.

```
g.V("my-id123").as("start").outE("knows").has("edgePropertyKey1",
 P.gt(0)).as("myedge").inV()
               .as("end").select("start", "end", "myedge").by("vertexPropertyKey1")
               .by("vertexPropertyKey1").by("edgePropertyKey1")
```

- Fixed an issue where a Gremlin Java client cannot connect to the server using the same WebSocket connection after the timeout of a long-running query.
- Fixed an issue where the escaped sequences contained as part of the Gremlin query over HTTP or string-based queries over the WebSocket connection were not handled correctly.

Introduction to using Amazon Neptune APIs

The Amazon Neptune management APIs provide SDK support for creating, managing and deleting Neptune DB clusters and instances, while the Neptune data APIs provide SDK support for loading data into your graph, running queries, getting information about the data in your graph, and running machine-learning operations. These APIs are available in all SDK languages. By automatically signing API requests, they greatly simply integrating Neptune into applications.

This page provides information about how to use these APIs.

IAM Actions with different names than their Neptune data API SDK counterparts

When you're calling a Neptune API methods on a cluster that has IAM authentication enabled, you have to have an IAM policy attached to the user or role making the calls that provides permissions for the actions you want to make. You set those permissions in the policy using corresponding IAM Actions. You can also restrict the actions that can be taken using IAM Condition keys.

Most IAM actions have the same name as the API methods that they correspond to, but some methods in the data API have different names, because some are shared by more than one method. The table below lists data methods and their corresponding IAM actions:

Data API operation name	IAM correspondences
CancelGremlinQuery (cancel_gremlin_query)	*Action:* `neptune-db:` **`CancelQuery`**
CancelLoaderJob (cancel_loader_job)	*Action:* `neptune-db:CancelLoaderJob`
CancelMLDataProcessingJob (cancel_ml_data_processing_job)	*Action:* `neptune-db:CancelMLDataProcessingJob`
CancelMLModelTrainingJob (cancel_ml_model_training_job)	*Action:* `neptune-db:CancelMLModelTrainingJob`

Data API operation name	IAM correspondences
CancelOpenCypherQuery (cancel_open_cypher_query)	*Action:* `neptune-db:` **`CancelQuery`**
CreateMLEndpoint (create_ml_endpoint)	*Action:* `neptune-db:CreateMLEndpoint`
DeleteMLEndpoint (delete_ml_endpoint)	*Action:* `neptune-db:DeleteMLEndpoint`
DeletePropertygraphStatistics (delete_propertygraph_statistics)	*Action:* `neptune-db:` **`DeleteStatistics`**
DeleteSparqlStatistics (delete_sparql_statistics)	*Action:* `neptune-db:` **`DeleteStatistics`**
ExecuteFastReset execute_fast_reset()	*Action:* `neptune-db:` **`ResetDatabase`**
ExecuteGremlinExplainQuery (execute_gremlin_explain_query)	*Actions:* • `neptune-db:` **`ReadDataViaQuery`** • `neptune-db:` **`WriteDataViaQuery`** • `neptune-db:` **`DeleteDataViaQuery`** *Condition key:* `neptune-db:QueryLanguage:Gremlin`

Data API operation name	IAM correspondences	
ExecuteGremlinProfileQuery (execute_gremlin_profile_query)	*Action:* `neptune-db:` **`ReadDataViaQuery`** *Condition key:* `neptune-db:QueryLanguage:Gremlin`	
ExecuteGremlinQuery (execute_gremlin_query)	*Actions:* • `neptune-db:` **`ReadDataViaQuery`** • `neptune-db:` **`WriteDataViaQuery`** • `neptune-db:` **`DeleteDataViaQuery`** *Condition key:* `neptune-db:QueryLanguage:Gremlin`	
ExecuteOpenCypherExplainQuery (execute_open_cypher_explain_query)	*Action:* `neptune-db:` **`ReadDataViaQuery`** *Condition key:* `neptune-db:QueryLanguage:OpenCypher`	

Data API operation name	IAM correspondences	
ExecuteOpenCypherQuery (execute_open_cypher_query)	*Actions:* • `neptune-db:` **`ReadDataViaQuery`** • `neptune-db:` **`WriteDataViaQuery`** • `neptune-db:` **`DeleteDataViaQuery`** *Condition key:* `neptune-db:QueryLanguage:OpenCypher`	
GetEngineStatus (get_engine_status)	*Action:* `neptune-db:GetEngineStatus`	
GetGremlinQueryStatus (get_gremlin_query_status)	*Action:* `neptune-db:` **`:GetQueryStatus`** *Condition key:* `neptune-db:QueryLanguage:Gremlin`	
GetLoaderJobStatus (get_loader_job_status)	*Action:* `neptune-db:GetLoaderJobStatus`	
GetMLDataProcessingJob (get_ml_data_processing_job)	*Action:* `neptune-db:` **`GetMLDataProcessingJobStatus`**	
GetMLEndpoint (get_ml_endpoint)	*Action:* `neptune-db:` **`GetMLEndpointStatus`**	

Data API operation name	IAM correspondences	
GetMLModelTrainingJob (get_ml_model_training_job)	*Action:* neptune-db: **GetMLModelTrainingJobStatus**	
GetMLModelTransformJob (get_ml_model_transform_job)	*Action:* neptune-db: **GetMLModelTransformJobStatus**	
GetOpenCypherQueryStatus (get_open_cypher_query_status)	*Action:* neptune-db: **:GetQueryStatus** *Condition key:* neptune-db:QueryLanguage:OpenCypher	
GetPropertygraphStatistics (get_propertygraph_statistics)	*Action:* neptune-db: **GetStatisticsStatus**	
GetPropertygraphStream (get_propertygraph_stream)	*Action:* neptune-db: **GetStreamRecords** *Condition keys:* • neptune-db:QueryLanguage:Gremlin • neptune-db:QueryLanguage:OpenCypher	
GetPropertygraphSummary (get_propertygraph_summary)	*Action:* neptune-db: **GetGraphSummary**	
GetRDFGraphSummary (get_rdf_graph_summary)	*Action:* neptune-db: **GetGraphSummary**	

Data API operation name	IAM correspondences	
GetSparqlStatistics (get_sparql_statistics)	*Action:* `neptune-db: GetStatisticsStatus`	
GetSparqlStream (get_sparql_stream)	*Action:* `neptune-db: :GetStreamRecords` *Condition key:* `neptune-db:QueryLanguage:Sparql`	
ListGremlinQueries (list_gremlin_queries)	*Action:* `neptune-db: :GetQueryStatus` *Condition key:* `neptune-db:QueryLanguage:Gremlin`	
ListMLEndpoints (list_ml_endpoints)	*Action:* `neptune-db:ListMLEndpoints`	
ListMLModelTrainingJobs (list_ml_model_training_jobs)	*Action:* `neptune-db:ListMLModelTrainingJobs`	
ListMLModelTransformJobs (list_ml_model_transform_jobs)	*Action:* `neptune-db:ListMLModelTransformJobs`	
ListOpenCypherQueries (list_open_cypher_queries)	*Action:* `neptune-db: :GetQueryStatus` *Condition key:* `neptune-db:QueryLanguage:OpenCypher`	

Data API operation name	IAM correspondences	
ManagePropertygraphStatistics (manage_propertygraph_statistics)	*Action:* `neptune-db:` **`ManageStatistics`**	
ManageSparqlStatistics (manage_sparql_statistics)	*Action:* `neptune-db:` **`ManageStatistics`**	
StartLoaderJob (start_loader_job)	*Action:* `neptune-db:StartLoaderJob`	
StartMLModelDataProcessingJob (start_ml_data_processing_job)	*Action:* `neptune-db:StartMLModelDataProcessingJob`	
StartMLModelTrainingJob (start_ml_model_training_job)	*Action:* `neptune-db:StartMLModelTrainingJob`	
StartMLModelTransformJob (start_ml_model_transform_job)	*Action:* `neptune-db:StartMLModelTransformJob`	

Amazon Neptune Management API Reference

This chapter documents the Neptune API operations that you can use to manage and maintain your Neptune DB cluster.

Neptune operates on clusters of database servers that are connected in a replication topology. Thus, managing Neptune often involves deploying changes to multiple servers and making sure that all Neptune replicas are keeping up with the primary server.

Because Neptune transparently scales the underlying storage as your data grows, managing Neptune requires relatively little management of disk storage. Likewise, because Neptune automatically performs continuous backups, a Neptune cluster does not require extensive planning or downtime for performing backups.

Contents

- Neptune DB Clusters API
  - CreateDBCluster (action)
  - DeleteDBCluster (action)
  - ModifyDBCluster (action)
  - StartDBCluster (action)
  - StopDBCluster (action)
  - AddRoleToDBCluster (action)
  - RemoveRoleFromDBCluster (action)
  - FailoverDBCluster (action)
  - PromoteReadReplicaDBCluster (action)
  - DescribeDBClusters (action)
  - Structures:
  - DBCluster (structure)
  - DBClusterMember (structure)
  - DBClusterRole (structure)
  - CloudwatchLogsExportConfiguration (structure)
  - PendingCloudwatchLogsExports (structure)
  - ClusterPendingModifiedValues (structure)

- Neptune Global Database API
  - CreateGlobalCluster (action)
  - DeleteGlobalCluster (action)
  - ModifyGlobalCluster (action)
  - DescribeGlobalClusters (action)
  - FailoverGlobalCluster (action)
  - RemoveFromGlobalCluster (action)
  - Structures:
  - GlobalCluster (structure)
  - GlobalClusterMember (structure)
- Neptune Instances API
  - CreateDBInstance (action)
  - DeleteDBInstance (action)
  - ModifyDBInstance (action)
  - RebootDBInstance (action)
  - DescribeDBInstances (action)
  - DescribeOrderableDBInstanceOptions (action)
  - DescribeValidDBInstanceModifications (action)
  - Structures:
  - DBInstance (structure)
  - DBInstanceStatusInfo (structure)
  - OrderableDBInstanceOption (structure)
  - PendingModifiedValues (structure)
  - ValidStorageOptions (structure)
  - ValidDBInstanceModificationsMessage (structure)
- Neptune Parameters API
  - CopyDBParameterGroup (action)
  - CopyDBClusterParameterGroup (action)
  - CreateDBParameterGroup (action)
  - CreateDBClusterParameterGroup (action)

- DeleteDBParameterGroup (action)
- DeleteDBClusterParameterGroup (action)
- ModifyDBParameterGroup (action)
- ModifyDBClusterParameterGroup (action)
- ResetDBParameterGroup (action)
- ResetDBClusterParameterGroup (action)
- DescribeDBParameters (action)
- DescribeDBParameterGroups (action)
- DescribeDBClusterParameters (action)
- DescribeDBClusterParameterGroups (action)
- DescribeEngineDefaultParameters (action)
- DescribeEngineDefaultClusterParameters (action)
- Structures:
- Parameter (structure)
- DBParameterGroup (structure)
- DBClusterParameterGroup (structure)
- DBParameterGroupStatus (structure)
- Neptune Subnet API
  - CreateDBSubnetGroup (action)
  - DeleteDBSubnetGroup (action)
  - ModifyDBSubnetGroup (action)
  - DescribeDBSubnetGroups (action)
  - Structures:
  - Subnet (structure)
  - DBSubnetGroup (structure)
- Neptune Snapshots API
  - CreateDBClusterSnapshot (action)
  - DeleteDBClusterSnapshot (action)
  - CopyDBClusterSnapshot (action)
  - ModifyDBClusterSnapshotAttribute (action)

- RestoreDBClusterFromSnapshot (action)
- RestoreDBClusterToPointInTime (action)
- DescribeDBClusterSnapshots (action)
- DescribeDBClusterSnapshotAttributes (action)
- Structures:
- DBClusterSnapshot (structure)
- DBClusterSnapshotAttribute (structure)
- DBClusterSnapshotAttributesResult (structure)
- Neptune Events API
  - CreateEventSubscription (action)
  - DeleteEventSubscription (action)
  - ModifyEventSubscription (action)
  - DescribeEventSubscriptions (action)
  - AddSourceIdentifierToSubscription (action)
  - RemoveSourceIdentifierFromSubscription (action)
  - DescribeEvents (action)
  - DescribeEventCategories (action)
  - Structures:
  - Event (structure)
  - EventCategoriesMap (structure)
  - EventSubscription (structure)
- Other Neptune APIs
  - AddTagsToResource (action)
  - ListTagsForResource (action)
  - RemoveTagsFromResource (action)
  - ApplyPendingMaintenanceAction (action)
  - DescribePendingMaintenanceActions (action)
  - DescribeDBEngineVersions (action)
  - Structures:

  - DBEngineVersion (structure)

- EngineDefaults (structure)
- PendingMaintenanceAction (structure)
- ResourcePendingMaintenanceActions (structure)
- UpgradeTarget (structure)
- Tag (structure)
- Common Neptune Datatypes
  - AvailabilityZone (structure)
  - DBSecurityGroupMembership (structure)
  - DomainMembership (structure)
  - DoubleRange (structure)
  - Endpoint (structure)
  - Filter (structure)
  - Range (structure)
  - ServerlessV2ScalingConfiguration (structure)
  - ServerlessV2ScalingConfigurationInfo (structure)
  - Timezone (structure)
  - VpcSecurityGroupMembership (structure)
- Neptune Exceptions Specific to Individual APIs
  - AuthorizationAlreadyExistsFault (structure)
  - AuthorizationNotFoundFault (structure)
  - AuthorizationQuotaExceededFault (structure)
  - CertificateNotFoundFault (structure)
  - DBClusterAlreadyExistsFault (structure)
  - DBClusterNotFoundFault (structure)
  - DBClusterParameterGroupNotFoundFault (structure)
  - DBClusterQuotaExceededFault (structure)
  - DBClusterRoleAlreadyExistsFault (structure)
  - DBClusterRoleNotFoundFault (structure)
  - DBClusterRoleQuotaExceededFault (structure)
  - DBClusterSnapshotAlreadyExistsFault (structure)

- DBClusterSnapshotNotFoundFault (structure)
- DBInstanceAlreadyExistsFault (structure)
- DBInstanceNotFoundFault (structure)
- DBLogFileNotFoundFault (structure)
- DBParameterGroupAlreadyExistsFault (structure)
- DBParameterGroupNotFoundFault (structure)
- DBParameterGroupQuotaExceededFault (structure)
- DBSecurityGroupAlreadyExistsFault (structure)
- DBSecurityGroupNotFoundFault (structure)
- DBSecurityGroupNotSupportedFault (structure)
- DBSecurityGroupQuotaExceededFault (structure)
- DBSnapshotAlreadyExistsFault (structure)
- DBSnapshotNotFoundFault (structure)
- DBSubnetGroupAlreadyExistsFault (structure)
- DBSubnetGroupDoesNotCoverEnoughAZs (structure)
- DBSubnetGroupNotAllowedFault (structure)
- DBSubnetGroupNotFoundFault (structure)
- DBSubnetGroupQuotaExceededFault (structure)
- DBSubnetQuotaExceededFault (structure)
- DBUpgradeDependencyFailureFault (structure)
- DomainNotFoundFault (structure)
- EventSubscriptionQuotaExceededFault (structure)
- GlobalClusterAlreadyExistsFault (structure)
- GlobalClusterNotFoundFault (structure)
- GlobalClusterQuotaExceededFault (structure)
- InstanceQuotaExceededFault (structure)
- InsufficientDBClusterCapacityFault (structure)
- InsufficientDBInstanceCapacityFault (structure)
- InsufficientStorageClusterCapacityFault (structure)
- InvalidDBClusterEndpointStateFault (structure)

- InvalidDBClusterSnapshotStateFault (structure)
- InvalidDBClusterStateFault (structure)
- InvalidDBInstanceStateFault (structure)
- InvalidDBParameterGroupStateFault (structure)
- InvalidDBSecurityGroupStateFault (structure)
- InvalidDBSnapshotStateFault (structure)
- InvalidDBSubnetGroupFault (structure)
- InvalidDBSubnetGroupStateFault (structure)
- InvalidDBSubnetStateFault (structure)
- InvalidEventSubscriptionStateFault (structure)
- InvalidGlobalClusterStateFault (structure)
- InvalidOptionGroupStateFault (structure)
- InvalidRestoreFault (structure)
- InvalidSubnet (structure)
- InvalidVPCNetworkStateFault (structure)
- KMSKeyNotAccessibleFault (structure)
- OptionGroupNotFoundFault (structure)
- PointInTimeRestoreNotEnabledFault (structure)
- ProvisionedIopsNotAvailableInAZFault (structure)
- ResourceNotFoundFault (structure)
- SNSInvalidTopicFault (structure)
- SNSNoAuthorizationFault (structure)
- SNSTopicArnNotFoundFault (structure)
- SharedSnapshotQuotaExceededFault (structure)
- SnapshotQuotaExceededFault (structure)
- SourceNotFoundFault (structure)
- StorageQuotaExceededFault (structure)
- StorageTypeNotSupportedFault (structure)
- SubnetAlreadyInUse (structure)
- SubscriptionAlreadyExistFault (structure)

- SubscriptionCategoryNotFoundFault (structure)
- SubscriptionNotFoundFault (structure)

Neptune DB Clusters API

Actions:

- CreateDBCluster (action)
- DeleteDBCluster (action)
- ModifyDBCluster (action)
- StartDBCluster (action)
- StopDBCluster (action)
- AddRoleToDBCluster (action)
- RemoveRoleFromDBCluster (action)
- FailoverDBCluster (action)
- PromoteReadReplicaDBCluster (action)
- DescribeDBClusters (action)

Structures:

- DBCluster (structure)
- DBClusterMember (structure)
- DBClusterRole (structure)
- CloudwatchLogsExportConfiguration (structure)
- PendingCloudwatchLogsExports (structure)
- ClusterPendingModifiedValues (structure)

CreateDBCluster (action)

The AWS CLI name for this API is: `create-db-cluster`.

Creates a new Amazon Neptune DB cluster.

You can use the `ReplicationSourceIdentifier` parameter to create the DB cluster as a Read Replica of another DB cluster or Amazon Neptune DB instance.

Note that when you create a new cluster using `CreateDBCluster` directly, deletion protection is disabled by default (when you create a new production cluster in the console, deletion protection is enabled by default). You can only delete a DB cluster if its `DeletionProtection` field is set to `false`.

Request

- **AvailabilityZones** (in the CLI: `--availability-zones`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of EC2 Availability Zones that instances in the DB cluster can be created in.
- **BackupRetentionPeriod** (in the CLI: `--backup-retention-period`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The number of days for which automated backups are retained. You must specify a minimum value of 1.

  Default: 1

  Constraints:

  - Must be a value from 1 to 35
- **CopyTagsToSnapshot** (in the CLI: `--copy-tags-to-snapshot`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, tags are copied to any snapshot of the DB cluster that is created.
- **DatabaseName** (in the CLI: `--database-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name for your database of up to 64 alpha-numeric characters. If you do not provide a name, Amazon Neptune will not create a database in the DB cluster you are creating.
- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The DB cluster identifier. This parameter is stored as a lowercase string.

  Constraints:

- Must contain from 1 to 63 letters, numbers, or hyphens.
- First character must be a letter.
- Cannot end with a hyphen or contain two consecutive hyphens.

Example: `my-cluster1`

- **DBClusterParameterGroupName** (in the CLI: `--db-cluster-parameter-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster parameter group to associate with this DB cluster. If this argument is omitted, the default is used.

  Constraints:

  - If supplied, must match the name of an existing DBClusterParameterGroup.
- **DBSubnetGroupName** (in the CLI: `--db-subnet-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  A DB subnet group to associate with this DB cluster.

  Constraints: Must match the name of an existing DBSubnetGroup. Must not be default.

  Example: `mySubnetgroup`
- **DeletionProtection** (in the CLI: `--deletion-protection`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled. By default, deletion protection is enabled.
- **EnableCloudwatchLogsExports** (in the CLI: `--enable-cloudwatch-logs-exports`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of the log types that this DB cluster should export to CloudWatch Logs. Valid log types are: `audit` (to publish audit logs) and `slowquery` (to publish slow-query logs). See Publishing Neptune logs to Amazon CloudWatch logs.
- **EnableIAMDatabaseAuthentication** (in the CLI: `--enable-iam-database-authentication`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, enables Amazon Identity and Access Management (IAM) authentication for the entire DB cluster (this cannot be set at an instance level).

Default: `false`.

- **Engine** (in the CLI: `--engine`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the database engine to be used for this DB cluster.

  Valid Values: `neptune`

- **EngineVersion** (in the CLI: `--engine-version`) – a String, of type: `string` (a UTF-8 encoded string).

  The version number of the database engine to use for the new DB cluster.

  Example: `1.2.1.0`

- **GlobalClusterIdentifier** (in the CLI: `--global-cluster-identifier`) – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  The ID of the Neptune global database to which this new DB cluster should be added.

- **KmsKeyId** (in the CLI: `--kms-key-id`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon KMS key identifier for an encrypted DB cluster.

  The KMS key identifier is the Amazon Resource Name (ARN) for the KMS encryption key. If you are creating a DB cluster with the same Amazon account that owns the KMS encryption key used to encrypt the new DB cluster, then you can use the KMS key alias instead of the ARN for the KMS encryption key.

  If an encryption key is not specified in `KmsKeyId`:

  - If `ReplicationSourceIdentifier` identifies an encrypted source, then Amazon Neptune will use the encryption key used to encrypt the source. Otherwise, Amazon Neptune will use your default encryption key.
  - If the `StorageEncrypted` parameter is true and `ReplicationSourceIdentifier` is not specified, then Amazon Neptune will use your default encryption key.

  Amazon KMS creates the default encryption key for your Amazon account. Your Amazon account has a different default encryption key for each Amazon Region.

If you create a Read Replica of an encrypted DB cluster in another Amazon Region, you must set `KmsKeyId` to a KMS key ID that is valid in the destination Amazon Region. This key is used to encrypt the Read Replica in that Amazon Region.

- **Port** (in the CLI: `--port`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The port number on which the instances in the DB cluster accept connections.

  Default: `8182`

- **PreferredBackupWindow** (in the CLI: `--preferred-backup-window`) – a String, of type: `string` (a UTF-8 encoded string).

  The daily time range during which automated backups are created if automated backups are enabled using the `BackupRetentionPeriod` parameter.

  The default is a 30-minute window selected at random from an 8-hour block of time for each Amazon Region. To see the time blocks available, see Neptune Maintenance Window in the *Amazon Neptune User Guide.*

  Constraints:

  - Must be in the format `hh24:mi-hh24:mi`.
  - Must be in Universal Coordinated Time (UTC).
  - Must not conflict with the preferred maintenance window.
  - Must be at least 30 minutes.

- **PreferredMaintenanceWindow** (in the CLI: `--preferred-maintenance-window`) – a String, of type: `string` (a UTF-8 encoded string).

  The weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

  Format: `ddd:hh24:mi-ddd:hh24:mi`

  The default is a 30-minute window selected at random from an 8-hour block of time for each Amazon Region, occurring on a random day of the week. To see the time blocks available, see Neptune Maintenance Window in the *Amazon Neptune User Guide.*

  Valid Days: Mon, Tue, Wed, Thu, Fri, Sat, Sun.

  Constraints: Minimum 30-minute window.

- **PreSignedUrl** (in the CLI: `--pre-signed-url`) – a String, of type: `string` (a UTF-8 encoded string).

  This parameter is not currently supported.
- **ReplicationSourceIdentifier** (in the CLI: `--replication-source-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of the source DB instance or DB cluster if this DB cluster is created as a Read Replica.
- **ServerlessV2ScalingConfiguration** (in the CLI: `--serverless-v2-scaling-configuration`) – A ServerlessV2ScalingConfiguration object.

  Contains the scaling configuration of a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.
- **StorageEncrypted** (in the CLI: `--storage-encrypted`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster is encrypted.
- **StorageType** (in the CLI: `--storage-type`) – a String, of type: `string` (a UTF-8 encoded string).

  The storage type for the new DB cluster.

  Valid Values:

  - **`standard`** – (*the default*) Configures cost-effective database storage for applications with moderate to small I/O usage. When set to `standard`, the storage type is not returned in the response.
  - **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

    Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.
- **Tags** (in the CLI: `--tags`) – An array of Tag objects.

  The tags to assign to the new DB cluster.
- **VpcSecurityGroupIds** (in the CLI: `--vpc-security-group-ids`) – a String, of type: `string` (a UTF-8 encoded string).

A list of EC2 VPC security groups to associate with this DB cluster.

Response

Contains the details of an Amazon Neptune DB cluster.

This data type is used as a response element in the the section called "DescribeDBClusters".

- **AllocatedStorage** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  `AllocatedStorage` always returns 1, because Neptune DB cluster storage size is not fixed, but instead automatically adjusts as needed.
- **AssociatedRoles** – An array of DBClusterRole objects.

  Provides a list of the Amazon Identity and Access Management (IAM) roles that are associated with the DB cluster. IAM roles that are associated with a DB cluster grant permission for the DB cluster to access other Amazon services on your behalf.
- **AutomaticRestartTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Time at which the DB cluster will be automatically restarted.
- **AvailabilityZones** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster can be created in.
- **BacktrackConsumedChangeRecords** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.
- **BacktrackWindow** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.
- **BackupRetentionPeriod** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.
- **Capacity** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Not supported by Neptune.
- **CloneGroupId** – a String, of type: `string` (a UTF-8 encoded string).

Identifies the clone group to which the DB cluster is associated.

- **ClusterCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).

- **CopyTagsToSnapshot** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, tags are copied to any snapshot of the DB cluster that is created.

- **CrossAccountClone** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, the DB cluster can be cloned across accounts.

- **DatabaseName** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the initial database of this DB cluster that was provided at create time, if one was specified when the DB cluster was created. This same name is returned for the life of the DB cluster.

- **DBClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster.

- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied DB cluster identifier. This identifier is the unique key that identifies a DB cluster.

- **DBClusterMembers** – An array of DBClusterMember objects.

  Provides the list of instances that make up the DB cluster.

- **DBClusterParameterGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB cluster parameter group for the DB cluster.

- **DbClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Region-unique, immutable identifier for the DB cluster. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB cluster is accessed.

- **DBSubnetGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies information on the subnet group associated with the DB cluster, including the name, description, and subnets in the subnet group.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled.
- **EarliestBacktrackTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Not supported by Neptune.
- **EarliestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the earliest time to which a database can be restored with point-in-time restore.
- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

  A list of the log types that this DB cluster is configured to export to CloudWatch Logs. Valid log types are: `audit` (to publish audit logs to CloudWatch) and slowquery (to publish slow-query logs to CloudWatch). See Publishing Neptune logs to Amazon CloudWatch logs.
- **Endpoint** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the connection endpoint for the primary instance of the DB cluster.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB cluster.
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.
- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.
- **HostedZoneId** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the ID that Amazon Route 53 assigns when you create a hosted zone.
- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.

- **IOOptimizedNextAllowedModificationTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The next time you can modify the DB cluster to use the `iopt1` storage type.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster.

- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.

- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster has instances in multiple Availability Zones.

- **PendingModifiedValues** – A ClusterPendingModifiedValues object.

  This data type is used as a response element in the `ModifyDBCluster` operation and contains changes that will be applied during the next maintenance window.

- **PercentProgress** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the progress of the operation as a percentage.

- **Port** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the database engine is listening on.

- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.

- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

- **ReaderEndpoint** – a String, of type: `string` (a UTF-8 encoded string).

The reader endpoint for the DB cluster. The reader endpoint for a DB cluster load-balances connections across the Read Replicas that are available in a DB cluster. As clients request new connections to the reader endpoint, Neptune distributes the connection requests among the Read Replicas in the DB cluster. This functionality can help balance your read workload across multiple Read Replicas in your DB cluster.

If a failover occurs, and the Read Replica that you are connected to is promoted to be the primary instance, your connection is dropped. To continue sending your read workload to other Read Replicas in the cluster, you can then reconnect to the reader endpoint.

- **ReadReplicaIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB cluster.

- **ReplicationSourceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ReplicationType** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ServerlessV2ScalingConfiguration** – A ServerlessV2ScalingConfigurationInfo object.

  Shows the scaling configuration for a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this DB cluster.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster is encrypted.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  The storage type used by the DB cluster.

  Valid Values:

  - **`standard`** – (*the default*) Provides cost-effective database storage for applications with moderate to small I/O usage.

- **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

  Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.
- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

  Provides a list of VPC security groups that the DB cluster belongs to.

Errors

- DBClusterAlreadyExistsFault
- InsufficientStorageClusterCapacityFault
- DBClusterQuotaExceededFault
- StorageQuotaExceededFault
- DBSubnetGroupNotFoundFault
- InvalidVPCNetworkStateFault
- InvalidDBClusterStateFault
- InvalidDBSubnetGroupStateFault
- InvalidSubnet
- InvalidDBInstanceStateFault
- DBClusterParameterGroupNotFoundFault
- KMSKeyNotAccessibleFault
- DBClusterNotFoundFault
- DBInstanceNotFoundFault
- DBSubnetGroupDoesNotCoverEnoughAZs
- GlobalClusterNotFoundFault
- InvalidGlobalClusterStateFault

DeleteDBCluster (action)

The AWS CLI name for this API is: `delete-db-cluster`.

The DeleteDBCluster action deletes a previously provisioned DB cluster. When you delete a DB cluster, all automated backups for that DB cluster are deleted and can't be recovered. Manual DB cluster snapshots of the specified DB cluster are not deleted.

Note that the DB Cluster cannot be deleted if deletion protection is enabled. To delete it, you must first set its `DeletionProtection` field to `False`.

Request

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The DB cluster identifier for the DB cluster to be deleted. This parameter isn't case-sensitive.

  Constraints:

  - Must match an existing DBClusterIdentifier.
- **FinalDBSnapshotIdentifier** (in the CLI: `--final-db-snapshot-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  The DB cluster snapshot identifier of the new DB cluster snapshot created when `SkipFinalSnapshot` is set to `false`.

  > **Note**
  >
  > Specifying this parameter and also setting the `SkipFinalShapshot` parameter to true results in an error.

  Constraints:

  - Must be 1 to 255 letters, numbers, or hyphens.
  - First character must be a letter
  - Cannot end with a hyphen or contain two consecutive hyphens
- **SkipFinalSnapshot** (in the CLI: `--skip-final-snapshot`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Determines whether a final DB cluster snapshot is created before the DB cluster is deleted. If `true` is specified, no DB cluster snapshot is created. If `false` is specified, a DB cluster snapshot is created before the DB cluster is deleted.

> **Note**
>
> You must specify a `FinalDBSnapshotIdentifier` parameter if `SkipFinalSnapshot` is `false`.

Default: `false`

Response

Contains the details of an Amazon Neptune DB cluster.

This data type is used as a response element in the the section called "DescribeDBClusters".

- **AllocatedStorage** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  `AllocatedStorage` always returns 1, because Neptune DB cluster storage size is not fixed, but instead automatically adjusts as needed.
- **AssociatedRoles** – An array of DBClusterRole objects.

  Provides a list of the Amazon Identity and Access Management (IAM) roles that are associated with the DB cluster. IAM roles that are associated with a DB cluster grant permission for the DB cluster to access other Amazon services on your behalf.
- **AutomaticRestartTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Time at which the DB cluster will be automatically restarted.
- **AvailabilityZones** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster can be created in.
- **BacktrackConsumedChangeRecords** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.
- **BacktrackWindow** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.
- **BackupRetentionPeriod** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

Specifies the number of days for which automatic DB snapshots are retained.

- **Capacity** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Not supported by Neptune.

- **CloneGroupId** – a String, of type: `string` (a UTF-8 encoded string).

  Identifies the clone group to which the DB cluster is associated.

- **ClusterCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).

- **CopyTagsToSnapshot** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, tags are copied to any snapshot of the DB cluster that is created.

- **CrossAccountClone** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, the DB cluster can be cloned across accounts.

- **DatabaseName** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the initial database of this DB cluster that was provided at create time, if one was specified when the DB cluster was created. This same name is returned for the life of the DB cluster.

- **DBClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster.

- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied DB cluster identifier. This identifier is the unique key that identifies a DB cluster.

- **DBClusterMembers** – An array of DBClusterMember objects.

  Provides the list of instances that make up the DB cluster.

- **DBClusterParameterGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB cluster parameter group for the DB cluster.

- **DbClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

The Amazon Region-unique, immutable identifier for the DB cluster. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB cluster is accessed.

- **DBSubnetGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies information on the subnet group associated with the DB cluster, including the name, description, and subnets in the subnet group.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled.

- **EarliestBacktrackTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Not supported by Neptune.

- **EarliestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the earliest time to which a database can be restored with point-in-time restore.

- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

  A list of the log types that this DB cluster is configured to export to CloudWatch Logs. Valid log types are: `audit` (to publish audit logs to CloudWatch) and slowquery (to publish slow-query logs to CloudWatch). See Publishing Neptune logs to Amazon CloudWatch logs.

- **Endpoint** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the connection endpoint for the primary instance of the DB cluster.

- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB cluster.

- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.

- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.

- **HostedZoneId** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the ID that Amazon Route 53 assigns when you create a hosted zone.

- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.

- **IOOptimizedNextAllowedModificationTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The next time you can modify the DB cluster to use the `iopt1` storage type.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster.

- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.

- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster has instances in multiple Availability Zones.

- **PendingModifiedValues** – A ClusterPendingModifiedValues object.

  This data type is used as a response element in the `ModifyDBCluster` operation and contains changes that will be applied during the next maintenance window.

- **PercentProgress** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the progress of the operation as a percentage.

- **Port** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the database engine is listening on.

- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.

- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

- **ReaderEndpoint** – a String, of type: `string` (a UTF-8 encoded string).

  The reader endpoint for the DB cluster. The reader endpoint for a DB cluster load-balances connections across the Read Replicas that are available in a DB cluster. As clients request new connections to the reader endpoint, Neptune distributes the connection requests among the Read Replicas in the DB cluster. This functionality can help balance your read workload across multiple Read Replicas in your DB cluster.

  If a failover occurs, and the Read Replica that you are connected to is promoted to be the primary instance, your connection is dropped. To continue sending your read workload to other Read Replicas in the cluster, you can then reconnect to the reader endpoint.

- **ReadReplicaIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB cluster.

- **ReplicationSourceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ReplicationType** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ServerlessV2ScalingConfiguration** – A ServerlessV2ScalingConfigurationInfo object.

  Shows the scaling configuration for a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this DB cluster.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster is encrypted.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  The storage type used by the DB cluster.

  Valid Values:

  - **`standard`** – (*the default*) Provides cost-effective database storage for applications with moderate to small I/O usage.
  - **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

    Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.
- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

  Provides a list of VPC security groups that the DB cluster belongs to.

Errors

- DBClusterNotFoundFault
- InvalidDBClusterStateFault
- DBClusterSnapshotAlreadyExistsFault
- SnapshotQuotaExceededFault
- InvalidDBClusterSnapshotStateFault

ModifyDBCluster (action)

The AWS CLI name for this API is: `modify-db-cluster`.

Modify a setting for a DB cluster. You can change one or more database configuration parameters by specifying these parameters and the new values in the request.

Request

- **AllowMajorVersionUpgrade** (in the CLI: `--allow-major-version-upgrade`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether upgrades between different major versions are allowed.

Constraints: You must set the allow-major-version-upgrade flag when providing an `EngineVersion` parameter that uses a different major version than the DB cluster's current version.

- **ApplyImmediately** (in the CLI: `--apply-immediately`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A value that specifies whether the modifications in this request and any pending modifications are asynchronously applied as soon as possible, regardless of the `PreferredMaintenanceWindow` setting for the DB cluster. If this parameter is set to `false`, changes to the DB cluster are applied during the next maintenance window.

  The `ApplyImmediately` parameter only affects `NewDBClusterIdentifier` values. If you set the `ApplyImmediately` parameter value to false, then changes to `NewDBClusterIdentifier` values are applied during the next maintenance window. All other changes are applied immediately, regardless of the value of the `ApplyImmediately` parameter.

  Default: `false`

- **BackupRetentionPeriod** (in the CLI: `--backup-retention-period`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The number of days for which automated backups are retained. You must specify a minimum value of 1.

  Default: 1

  Constraints:

  - Must be a value from 1 to 35

- **CloudwatchLogsExportConfiguration** (in the CLI: `--cloudwatch-logs-export-configuration`) – A CloudwatchLogsExportConfiguration object.

  The configuration setting for the log types to be enabled for export to CloudWatch Logs for a specific DB cluster. See Using the CLI to publish Neptune audit logs to CloudWatch Logs.

- **CopyTagsToSnapshot** (in the CLI: `--copy-tags-to-snapshot`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, tags are copied to any snapshot of the DB cluster that is created.

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The DB cluster identifier for the cluster being modified. This parameter is not case-sensitive.

  Constraints:

  - Must match the identifier of an existing DBCluster.
- **DBClusterParameterGroupName** (in the CLI: `--db-cluster-parameter-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster parameter group to use for the DB cluster.
- **DBInstanceParameterGroupName** (in the CLI: `--db-instance-parameter-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB parameter group to apply to all instances of the DB cluster.

  > **Note**
  >
  > When you apply a parameter group using `DBInstanceParameterGroupName`, parameter changes aren't applied during the next maintenance window but instead are applied immediately.

  Default: The existing name setting

  Constraints:

  - The DB parameter group must be in the same DB parameter group family as the target DB cluster version.
  - The `DBInstanceParameterGroupName` parameter is only valid in combination with the `AllowMajorVersionUpgrade` parameter.
- **DeletionProtection** (in the CLI: `--deletion-protection`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled. By default, deletion protection is disabled.
- **EnableIAMDatabaseAuthentication** (in the CLI: `--enable-iam-database-authentication`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

True to enable mapping of Amazon Identity and Access Management (IAM) accounts to database accounts, and otherwise false.

Default: `false`

- **EngineVersion** (in the CLI: `--engine-version`) – a String, of type: `string` (a UTF-8 encoded string).

  The version number of the database engine to which you want to upgrade. Changing this parameter results in an outage. The change is applied during the next maintenance window unless the `ApplyImmediately` parameter is set to true.

  For a list of valid engine versions, see Engine Releases for Amazon Neptune, or call the section called "DescribeDBEngineVersions".

- **NewDBClusterIdentifier** (in the CLI: `--new-db-cluster-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  The new DB cluster identifier for the DB cluster when renaming a DB cluster. This value is stored as a lowercase string.

  Constraints:

  - Must contain from 1 to 63 letters, numbers, or hyphens
  - The first character must be a letter
  - Cannot end with a hyphen or contain two consecutive hyphens

  Example: `my-cluster2`

- **Port** (in the CLI: `--port`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The port number on which the DB cluster accepts connections.

  Constraints: Value must be `1150-65535`

  Default: The same port as the original DB cluster.

- **PreferredBackupWindow** (in the CLI: `--preferred-backup-window`) – a String, of type: `string` (a UTF-8 encoded string).

  The daily time range during which automated backups are created if automated backups are enabled, using the `BackupRetentionPeriod` parameter.

The default is a 30-minute window selected at random from an 8-hour block of time for each Amazon Region.

Constraints:

- Must be in the format `hh24:mi-hh24:mi`.
- Must be in Universal Coordinated Time (UTC).
- Must not conflict with the preferred maintenance window.
- Must be at least 30 minutes.

- **PreferredMaintenanceWindow** (in the CLI: `--preferred-maintenance-window`) – a String, of type: `string` (a UTF-8 encoded string).

  The weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

  Format: `ddd:hh24:mi-ddd:hh24:mi`

  The default is a 30-minute window selected at random from an 8-hour block of time for each Amazon Region, occurring on a random day of the week.

  Valid Days: Mon, Tue, Wed, Thu, Fri, Sat, Sun.

  Constraints: Minimum 30-minute window.

- **ServerlessV2ScalingConfiguration** (in the CLI: `--serverless-v2-scaling-configuration`) – A ServerlessV2ScalingConfiguration object.

  Contains the scaling configuration of a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

- **StorageType** (in the CLI: `--storage-type`) – a String, of type: `string` (a UTF-8 encoded string).

  The storage type to associate with the DB cluster.

  Valid Values:

  - **`standard`** – (*the default*) Configures cost-effective database storage for applications with moderate to small I/O usage.

- **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

  Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.
- **VpcSecurityGroupIds** (in the CLI: `--vpc-security-group-ids`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of VPC security groups that the DB cluster will belong to.

Response

Contains the details of an Amazon Neptune DB cluster.

This data type is used as a response element in the the section called "DescribeDBClusters".

- **AllocatedStorage** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  `AllocatedStorage` always returns 1, because Neptune DB cluster storage size is not fixed, but instead automatically adjusts as needed.
- **AssociatedRoles** – An array of DBClusterRole objects.

  Provides a list of the Amazon Identity and Access Management (IAM) roles that are associated with the DB cluster. IAM roles that are associated with a DB cluster grant permission for the DB cluster to access other Amazon services on your behalf.
- **AutomaticRestartTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Time at which the DB cluster will be automatically restarted.
- **AvailabilityZones** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster can be created in.
- **BacktrackConsumedChangeRecords** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.
- **BacktrackWindow** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.

- **BackupRetentionPeriod** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.
- **Capacity** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Not supported by Neptune.
- **CloneGroupId** – a String, of type: `string` (a UTF-8 encoded string).

  Identifies the clone group to which the DB cluster is associated.
- **ClusterCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).
- **CopyTagsToSnapshot** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, tags are copied to any snapshot of the DB cluster that is created.
- **CrossAccountClone** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, the DB cluster can be cloned across accounts.
- **DatabaseName** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the initial database of this DB cluster that was provided at create time, if one was specified when the DB cluster was created. This same name is returned for the life of the DB cluster.
- **DBClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster.
- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied DB cluster identifier. This identifier is the unique key that identifies a DB cluster.
- **DBClusterMembers** – An array of DBClusterMember objects.

  Provides the list of instances that make up the DB cluster.
- **DBClusterParameterGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB cluster parameter group for the DB cluster.
- **DbClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

The Amazon Region-unique, immutable identifier for the DB cluster. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB cluster is accessed.

- **DBSubnetGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies information on the subnet group associated with the DB cluster, including the name, description, and subnets in the subnet group.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled.

- **EarliestBacktrackTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Not supported by Neptune.

- **EarliestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the earliest time to which a database can be restored with point-in-time restore.

- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

  A list of the log types that this DB cluster is configured to export to CloudWatch Logs. Valid log types are: `audit` (to publish audit logs to CloudWatch) and slowquery (to publish slow-query logs to CloudWatch). See Publishing Neptune logs to Amazon CloudWatch logs.

- **Endpoint** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the connection endpoint for the primary instance of the DB cluster.

- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB cluster.

- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.

- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.

- **HostedZoneId** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the ID that Amazon Route 53 assigns when you create a hosted zone.

- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.

- **IOOptimizedNextAllowedModificationTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The next time you can modify the DB cluster to use the `iopt1` storage type.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster.

- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.

- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster has instances in multiple Availability Zones.

- **PendingModifiedValues** – A ClusterPendingModifiedValues object.

  This data type is used as a response element in the `ModifyDBCluster` operation and contains changes that will be applied during the next maintenance window.

- **PercentProgress** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the progress of the operation as a percentage.

- **Port** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the database engine is listening on.

- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.

- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

- **ReaderEndpoint** – a String, of type: `string` (a UTF-8 encoded string).

  The reader endpoint for the DB cluster. The reader endpoint for a DB cluster load-balances connections across the Read Replicas that are available in a DB cluster. As clients request new connections to the reader endpoint, Neptune distributes the connection requests among the Read Replicas in the DB cluster. This functionality can help balance your read workload across multiple Read Replicas in your DB cluster.

  If a failover occurs, and the Read Replica that you are connected to is promoted to be the primary instance, your connection is dropped. To continue sending your read workload to other Read Replicas in the cluster, you can then reconnect to the reader endpoint.

- **ReadReplicaIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB cluster.

- **ReplicationSourceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ReplicationType** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ServerlessV2ScalingConfiguration** – A ServerlessV2ScalingConfigurationInfo object.

  Shows the scaling configuration for a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this DB cluster.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster is encrypted.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  The storage type used by the DB cluster.

  Valid Values:

  - **`standard`** – (*the default*) Provides cost-effective database storage for applications with moderate to small I/O usage.
  - **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

    Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.
- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

  Provides a list of VPC security groups that the DB cluster belongs to.

Errors

- DBClusterNotFoundFault
- InvalidDBClusterStateFault
- StorageQuotaExceededFault
- DBSubnetGroupNotFoundFault
- InvalidVPCNetworkStateFault
- InvalidDBSubnetGroupStateFault
- InvalidSubnet
- DBClusterParameterGroupNotFoundFault
- InvalidDBSecurityGroupStateFault
- InvalidDBInstanceStateFault
- DBClusterAlreadyExistsFault
- StorageTypeNotSupportedFault

StartDBCluster (action)

The AWS CLI name for this API is: `start-db-cluster`.

Starts an Amazon Neptune DB cluster that was stopped using the Amazon console, the Amazon CLI stop-db-cluster command, or the StopDBCluster API.

Request

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The DB cluster identifier of the Neptune DB cluster to be started. This parameter is stored as a lowercase string.

Response

Contains the details of an Amazon Neptune DB cluster.

This data type is used as a response element in the the section called "DescribeDBClusters".

- **AllocatedStorage** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  `AllocatedStorage` always returns 1, because Neptune DB cluster storage size is not fixed, but instead automatically adjusts as needed.
- **AssociatedRoles** – An array of DBClusterRole objects.

  Provides a list of the Amazon Identity and Access Management (IAM) roles that are associated with the DB cluster. IAM roles that are associated with a DB cluster grant permission for the DB cluster to access other Amazon services on your behalf.
- **AutomaticRestartTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Time at which the DB cluster will be automatically restarted.
- **AvailabilityZones** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster can be created in.
- **BacktrackConsumedChangeRecords** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.
- **BacktrackWindow** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.

- **BackupRetentionPeriod** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.
- **Capacity** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Not supported by Neptune.
- **CloneGroupId** – a String, of type: `string` (a UTF-8 encoded string).

  Identifies the clone group to which the DB cluster is associated.
- **ClusterCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).
- **CopyTagsToSnapshot** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, tags are copied to any snapshot of the DB cluster that is created.
- **CrossAccountClone** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, the DB cluster can be cloned across accounts.
- **DatabaseName** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the initial database of this DB cluster that was provided at create time, if one was specified when the DB cluster was created. This same name is returned for the life of the DB cluster.
- **DBClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster.
- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied DB cluster identifier. This identifier is the unique key that identifies a DB cluster.
- **DBClusterMembers** – An array of DBClusterMember objects.

  Provides the list of instances that make up the DB cluster.
- **DBClusterParameterGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB cluster parameter group for the DB cluster.
- **DbClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

The Amazon Region-unique, immutable identifier for the DB cluster. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB cluster is accessed.

- **DBSubnetGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies information on the subnet group associated with the DB cluster, including the name, description, and subnets in the subnet group.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled.

- **EarliestBacktrackTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Not supported by Neptune.

- **EarliestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the earliest time to which a database can be restored with point-in-time restore.

- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

  A list of the log types that this DB cluster is configured to export to CloudWatch Logs. Valid log types are: `audit` (to publish audit logs to CloudWatch) and slowquery (to publish slow-query logs to CloudWatch). See Publishing Neptune logs to Amazon CloudWatch logs.

- **Endpoint** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the connection endpoint for the primary instance of the DB cluster.

- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB cluster.

- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.

- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.

- **HostedZoneId** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the ID that Amazon Route 53 assigns when you create a hosted zone.

- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.

- **IOOptimizedNextAllowedModificationTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The next time you can modify the DB cluster to use the `iopt1` storage type.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster.

- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.

- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster has instances in multiple Availability Zones.

- **PendingModifiedValues** – A ClusterPendingModifiedValues object.

  This data type is used as a response element in the `ModifyDBCluster` operation and contains changes that will be applied during the next maintenance window.

- **PercentProgress** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the progress of the operation as a percentage.

- **Port** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the database engine is listening on.

- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.

- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

- **ReaderEndpoint** – a String, of type: `string` (a UTF-8 encoded string).

  The reader endpoint for the DB cluster. The reader endpoint for a DB cluster load-balances connections across the Read Replicas that are available in a DB cluster. As clients request new connections to the reader endpoint, Neptune distributes the connection requests among the Read Replicas in the DB cluster. This functionality can help balance your read workload across multiple Read Replicas in your DB cluster.

  If a failover occurs, and the Read Replica that you are connected to is promoted to be the primary instance, your connection is dropped. To continue sending your read workload to other Read Replicas in the cluster, you can then reconnect to the reader endpoint.

- **ReadReplicaIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB cluster.

- **ReplicationSourceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ReplicationType** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ServerlessV2ScalingConfiguration** – A ServerlessV2ScalingConfigurationInfo object.

  Shows the scaling configuration for a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this DB cluster.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster is encrypted.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  The storage type used by the DB cluster.

  Valid Values:

  - **`standard`** – (*the default*) Provides cost-effective database storage for applications with moderate to small I/O usage.
  - **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

    Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.
- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

  Provides a list of VPC security groups that the DB cluster belongs to.

Errors

- DBClusterNotFoundFault
- InvalidDBClusterStateFault
- InvalidDBInstanceStateFault

StopDBCluster (action)

The AWS CLI name for this API is: `stop-db-cluster`.

Stops an Amazon Neptune DB cluster. When you stop a DB cluster, Neptune retains the DB cluster's metadata, including its endpoints and DB parameter groups.

Neptune also retains the transaction logs so you can do a point-in-time restore if necessary.

Request

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The DB cluster identifier of the Neptune DB cluster to be stopped. This parameter is stored as a lowercase string.

Response

Contains the details of an Amazon Neptune DB cluster.

This data type is used as a response element in the the section called "DescribeDBClusters".

- **AllocatedStorage** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  `AllocatedStorage` always returns 1, because Neptune DB cluster storage size is not fixed, but instead automatically adjusts as needed.
- **AssociatedRoles** – An array of DBClusterRole objects.

  Provides a list of the Amazon Identity and Access Management (IAM) roles that are associated with the DB cluster. IAM roles that are associated with a DB cluster grant permission for the DB cluster to access other Amazon services on your behalf.
- **AutomaticRestartTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Time at which the DB cluster will be automatically restarted.
- **AvailabilityZones** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster can be created in.
- **BacktrackConsumedChangeRecords** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.
- **BacktrackWindow** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.
- **BackupRetentionPeriod** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.
- **Capacity** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Not supported by Neptune.
- **CloneGroupId** – a String, of type: `string` (a UTF-8 encoded string).

  Identifies the clone group to which the DB cluster is associated.
- **ClusterCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).

- **CopyTagsToSnapshot** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, tags are copied to any snapshot of the DB cluster that is created.

- **CrossAccountClone** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, the DB cluster can be cloned across accounts.

- **DatabaseName** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the initial database of this DB cluster that was provided at create time, if one was specified when the DB cluster was created. This same name is returned for the life of the DB cluster.

- **DBClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster.

- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied DB cluster identifier. This identifier is the unique key that identifies a DB cluster.

- **DBClusterMembers** – An array of DBClusterMember objects.

  Provides the list of instances that make up the DB cluster.

- **DBClusterParameterGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB cluster parameter group for the DB cluster.

- **DbClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Region-unique, immutable identifier for the DB cluster. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB cluster is accessed.

- **DBSubnetGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies information on the subnet group associated with the DB cluster, including the name, description, and subnets in the subnet group.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled.

- **EarliestBacktrackTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Not supported by Neptune.
- **EarliestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the earliest time to which a database can be restored with point-in-time restore.
- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

  A list of the log types that this DB cluster is configured to export to CloudWatch Logs. Valid log types are: `audit` (to publish audit logs to CloudWatch) and slowquery (to publish slow-query logs to CloudWatch). See Publishing Neptune logs to Amazon CloudWatch logs.
- **Endpoint** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the connection endpoint for the primary instance of the DB cluster.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB cluster.
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.
- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.
- **HostedZoneId** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the ID that Amazon Route 53 assigns when you create a hosted zone.
- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.
- **IOOptimizedNextAllowedModificationTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

The next time you can modify the DB cluster to use the `iopt1` storage type.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster.

- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.

- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster has instances in multiple Availability Zones.

- **PendingModifiedValues** – A ClusterPendingModifiedValues object.

  This data type is used as a response element in the `ModifyDBCluster` operation and contains changes that will be applied during the next maintenance window.

- **PercentProgress** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the progress of the operation as a percentage.

- **Port** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the database engine is listening on.

- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.

- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

- **ReaderEndpoint** – a String, of type: `string` (a UTF-8 encoded string).

  The reader endpoint for the DB cluster. The reader endpoint for a DB cluster load-balances connections across the Read Replicas that are available in a DB cluster. As clients request new connections to the reader endpoint, Neptune distributes the connection requests among the Read Replicas in the DB cluster. This functionality can help balance your read workload across multiple Read Replicas in your DB cluster.

If a failover occurs, and the Read Replica that you are connected to is promoted to be the primary instance, your connection is dropped. To continue sending your read workload to other Read Replicas in the cluster, you can then reconnect to the reader endpoint.

- **ReadReplicaIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB cluster.

- **ReplicationSourceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ReplicationType** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ServerlessV2ScalingConfiguration** – A ServerlessV2ScalingConfigurationInfo object.

  Shows the scaling configuration for a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this DB cluster.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster is encrypted.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  The storage type used by the DB cluster.

  Valid Values:

  - **`standard`** – (*the default*) Provides cost-effective database storage for applications with moderate to small I/O usage.
  - **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

    Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.

- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

Provides a list of VPC security groups that the DB cluster belongs to.

Errors

- DBClusterNotFoundFault
- InvalidDBClusterStateFault
- InvalidDBInstanceStateFault

AddRoleToDBCluster (action)

The AWS CLI name for this API is: `add-role-to-db-cluster`.

Associates an Identity and Access Management (IAM) role with an Neptune DB cluster.

Request

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster to associate the IAM role with.
- **FeatureName** (in the CLI: `--feature-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the feature for the Neptune DB cluster that the IAM role is to be associated with. For the list of supported feature names, see the section called "DBEngineVersion".
- **RoleArn** (in the CLI: `--role-arn`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of the IAM role to associate with the Neptune DB cluster, for example `arn:aws:iam::123456789012:role/NeptuneAccessRole`.

Response

- *No Response parameters.*

Errors

- DBClusterNotFoundFault
- DBClusterRoleAlreadyExistsFault
- InvalidDBClusterStateFault
- DBClusterRoleQuotaExceededFault

RemoveRoleFromDBCluster (action)

The AWS CLI name for this API is: `remove-role-from-db-cluster`.

Disassociates an Identity and Access Management (IAM) role from a DB cluster.

Request

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster to disassociate the IAM role from.
- **FeatureName** (in the CLI: `--feature-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the feature for the DB cluster that the IAM role is to be disassociated from. For the list of supported feature names, see the section called "DescribeDBEngineVersions".
- **RoleArn** (in the CLI: `--role-arn`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of the IAM role to disassociate from the DB cluster, for example `arn:aws:iam::123456789012:role/NeptuneAccessRole`.

Response

- *No Response parameters.*

Errors

- DBClusterNotFoundFault

- DBClusterRoleNotFoundFault
- InvalidDBClusterStateFault

FailoverDBCluster (action)

The AWS CLI name for this API is: `failover-db-cluster`.

Forces a failover for a DB cluster.

A failover for a DB cluster promotes one of the Read Replicas (read-only instances) in the DB cluster to be the primary instance (the cluster writer).

Amazon Neptune will automatically fail over to a Read Replica, if one exists, when the primary instance fails. You can force a failover when you want to simulate a failure of a primary instance for testing. Because each instance in a DB cluster has its own endpoint address, you will need to clean up and re-establish any existing connections that use those endpoint addresses when the failover is complete.

Request

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  A DB cluster identifier to force a failover for. This parameter is not case-sensitive.

  Constraints:

  - Must match the identifier of an existing DBCluster.
- **TargetDBInstanceIdentifier** (in the CLI: `--target-db-instance-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the instance to promote to the primary instance.

  You must specify the instance identifier for an Read Replica in the DB cluster. For example, `mydbcluster-replica1`.

Response

Contains the details of an Amazon Neptune DB cluster.

This data type is used as a response element in the the section called "DescribeDBClusters".

- **AllocatedStorage** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  `AllocatedStorage` always returns 1, because Neptune DB cluster storage size is not fixed, but instead automatically adjusts as needed.
- **AssociatedRoles** – An array of DBClusterRole objects.

  Provides a list of the Amazon Identity and Access Management (IAM) roles that are associated with the DB cluster. IAM roles that are associated with a DB cluster grant permission for the DB cluster to access other Amazon services on your behalf.
- **AutomaticRestartTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Time at which the DB cluster will be automatically restarted.
- **AvailabilityZones** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster can be created in.
- **BacktrackConsumedChangeRecords** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.
- **BacktrackWindow** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.
- **BackupRetentionPeriod** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.
- **Capacity** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Not supported by Neptune.
- **CloneGroupId** – a String, of type: `string` (a UTF-8 encoded string).

  Identifies the clone group to which the DB cluster is associated.
- **ClusterCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).
- **CopyTagsToSnapshot** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

If set to `true`, tags are copied to any snapshot of the DB cluster that is created.

- **CrossAccountClone** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, the DB cluster can be cloned across accounts.

- **DatabaseName** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the initial database of this DB cluster that was provided at create time, if one was specified when the DB cluster was created. This same name is returned for the life of the DB cluster.

- **DBClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster.

- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied DB cluster identifier. This identifier is the unique key that identifies a DB cluster.

- **DBClusterMembers** – An array of DBClusterMember objects.

  Provides the list of instances that make up the DB cluster.

- **DBClusterParameterGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB cluster parameter group for the DB cluster.

- **DbClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Region-unique, immutable identifier for the DB cluster. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB cluster is accessed.

- **DBSubnetGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies information on the subnet group associated with the DB cluster, including the name, description, and subnets in the subnet group.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled.

- **EarliestBacktrackTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

Not supported by Neptune.

- **EarliestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the earliest time to which a database can be restored with point-in-time restore.
- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

  A list of the log types that this DB cluster is configured to export to CloudWatch Logs. Valid log types are: `audit` (to publish audit logs to CloudWatch) and slowquery (to publish slow-query logs to CloudWatch). See Publishing Neptune logs to Amazon CloudWatch logs.
- **Endpoint** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the connection endpoint for the primary instance of the DB cluster.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB cluster.
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.
- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.
- **HostedZoneId** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the ID that Amazon Route 53 assigns when you create a hosted zone.
- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.
- **IOOptimizedNextAllowedModificationTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The next time you can modify the DB cluster to use the `iopt1` storage type.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster.
- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.
- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster has instances in multiple Availability Zones.
- **PendingModifiedValues** – A ClusterPendingModifiedValues object.

  This data type is used as a response element in the `ModifyDBCluster` operation and contains changes that will be applied during the next maintenance window.
- **PercentProgress** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the progress of the operation as a percentage.
- **Port** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the database engine is listening on.
- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.
- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).
- **ReaderEndpoint** – a String, of type: `string` (a UTF-8 encoded string).

  The reader endpoint for the DB cluster. The reader endpoint for a DB cluster load-balances connections across the Read Replicas that are available in a DB cluster. As clients request new connections to the reader endpoint, Neptune distributes the connection requests among the Read Replicas in the DB cluster. This functionality can help balance your read workload across multiple Read Replicas in your DB cluster.

If a failover occurs, and the Read Replica that you are connected to is promoted to be the primary instance, your connection is dropped. To continue sending your read workload to other Read Replicas in the cluster, you can then reconnect to the reader endpoint.

- **ReadReplicaIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB cluster.

- **ReplicationSourceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ReplicationType** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ServerlessV2ScalingConfiguration** – A ServerlessV2ScalingConfigurationInfo object.

  Shows the scaling configuration for a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this DB cluster.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster is encrypted.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  The storage type used by the DB cluster.

  Valid Values:

  - **`standard`** – (*the default*) Provides cost-effective database storage for applications with moderate to small I/O usage.
  - **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

    Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.

- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

Provides a list of VPC security groups that the DB cluster belongs to.

Errors

- DBClusterNotFoundFault
- InvalidDBClusterStateFault
- InvalidDBInstanceStateFault

PromoteReadReplicaDBCluster (action)

The AWS CLI name for this API is: `promote-read-replica-db-cluster`.

Not supported.

Request

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  Not supported.

Response

Contains the details of an Amazon Neptune DB cluster.

This data type is used as a response element in the the section called "DescribeDBClusters".

- **AllocatedStorage** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  `AllocatedStorage` always returns 1, because Neptune DB cluster storage size is not fixed, but instead automatically adjusts as needed.
- **AssociatedRoles** – An array of DBClusterRole objects.

  Provides a list of the Amazon Identity and Access Management (IAM) roles that are associated with the DB cluster. IAM roles that are associated with a DB cluster grant permission for the DB cluster to access other Amazon services on your behalf.
- **AutomaticRestartTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

Time at which the DB cluster will be automatically restarted.

- **AvailabilityZones** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster can be created in.

- **BacktrackConsumedChangeRecords** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.

- **BacktrackWindow** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.

- **BackupRetentionPeriod** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.

- **Capacity** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Not supported by Neptune.

- **CloneGroupId** – a String, of type: `string` (a UTF-8 encoded string).

  Identifies the clone group to which the DB cluster is associated.

- **ClusterCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).

- **CopyTagsToSnapshot** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, tags are copied to any snapshot of the DB cluster that is created.

- **CrossAccountClone** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, the DB cluster can be cloned across accounts.

- **DatabaseName** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the initial database of this DB cluster that was provided at create time, if one was specified when the DB cluster was created. This same name is returned for the life of the DB cluster.

- **DBClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster.

- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied DB cluster identifier. This identifier is the unique key that identifies a DB cluster.
- **DBClusterMembers** – An array of DBClusterMember objects.

  Provides the list of instances that make up the DB cluster.
- **DBClusterParameterGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB cluster parameter group for the DB cluster.
- **DbClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Region-unique, immutable identifier for the DB cluster. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB cluster is accessed.
- **DBSubnetGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies information on the subnet group associated with the DB cluster, including the name, description, and subnets in the subnet group.
- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled.
- **EarliestBacktrackTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Not supported by Neptune.
- **EarliestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the earliest time to which a database can be restored with point-in-time restore.
- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

  A list of the log types that this DB cluster is configured to export to CloudWatch Logs. Valid log types are: `audit` (to publish audit logs to CloudWatch) and slowquery (to publish slow-query logs to CloudWatch). See Publishing Neptune logs to Amazon CloudWatch logs.
- **Endpoint** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the connection endpoint for the primary instance of the DB cluster.

- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB cluster.
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.
- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.
- **HostedZoneId** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the ID that Amazon Route 53 assigns when you create a hosted zone.
- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.
- **IOOptimizedNextAllowedModificationTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The next time you can modify the DB cluster to use the `iopt1` storage type.
- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster.
- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.
- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster has instances in multiple Availability Zones.
- **PendingModifiedValues** – A ClusterPendingModifiedValues object.

  This data type is used as a response element in the `ModifyDBCluster` operation and contains changes that will be applied during the next maintenance window.

- **PercentProgress** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the progress of the operation as a percentage.
- **Port** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the database engine is listening on.
- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.
- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).
- **ReaderEndpoint** – a String, of type: `string` (a UTF-8 encoded string).

  The reader endpoint for the DB cluster. The reader endpoint for a DB cluster load-balances connections across the Read Replicas that are available in a DB cluster. As clients request new connections to the reader endpoint, Neptune distributes the connection requests among the Read Replicas in the DB cluster. This functionality can help balance your read workload across multiple Read Replicas in your DB cluster.

  If a failover occurs, and the Read Replica that you are connected to is promoted to be the primary instance, your connection is dropped. To continue sending your read workload to other Read Replicas in the cluster, you can then reconnect to the reader endpoint.
- **ReadReplicaIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB cluster.
- **ReplicationSourceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.
- **ReplicationType** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.
- **ServerlessV2ScalingConfiguration** – A ServerlessV2ScalingConfigurationInfo object.

  Shows the scaling configuration for a Neptune Serverless DB cluster.

For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this DB cluster.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster is encrypted.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  The storage type used by the DB cluster.

  Valid Values:

  - **`standard`** – (*the default*) Provides cost-effective database storage for applications with moderate to small I/O usage.
  - **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

    Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.

- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

  Provides a list of VPC security groups that the DB cluster belongs to.

Errors

- DBClusterNotFoundFault
- InvalidDBClusterStateFault

DescribeDBClusters (action)

The AWS CLI name for this API is: `describe-db-clusters`.

Returns information about provisioned DB clusters, and supports pagination.

> **Note**
>
> This operation can also return information for Amazon RDS clusters and Amazon DocDB clusters.

Request

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  The user-supplied DB cluster identifier. If this parameter is specified, information from only the specific DB cluster is returned. This parameter isn't case-sensitive.

  Constraints:

  - If supplied, must match an existing DBClusterIdentifier.
- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  A filter that specifies one or more DB clusters to describe.

  Supported filters:

  - `db-cluster-id` - Accepts DB cluster identifiers and DB cluster Amazon Resource Names (ARNs). The results list will only include information about the DB clusters identified by these ARNs.
  - `engine` - Accepts an engine name (such as `neptune`), and restricts the results list to DB clusters created by that engine.

  For example, to invoke this API from the Amazon CLI and filter so that only Neptune DB clusters are returned, you could use the following command:

  Example

  ```
  aws neptune describe-db-clusters \
              --filters  Name=engine,Values=neptune
  ```

- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

An optional pagination token provided by a previous the section called "DescribeDBClusters" request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

  Default: 100

  Constraints: Minimum 20, maximum 100.

Response

- **DBClusters** – An array of DBCluster objects.

  Contains a list of DB clusters for the user.
- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  A pagination token that can be used in a subsequent DescribeDBClusters request.

Errors

- DBClusterNotFoundFault

Structures:

DBCluster (structure)

Contains the details of an Amazon Neptune DB cluster.

This data type is used as a response element in the the section called "DescribeDBClusters".

Fields

- **AllocatedStorage** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

`AllocatedStorage` always returns 1, because Neptune DB cluster storage size is not fixed, but instead automatically adjusts as needed.

- **AssociatedRoles** – This is An array of DBClusterRole objects.

  Provides a list of the Amazon Identity and Access Management (IAM) roles that are associated with the DB cluster. IAM roles that are associated with a DB cluster grant permission for the DB cluster to access other Amazon services on your behalf.

- **AutomaticRestartTime** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Time at which the DB cluster will be automatically restarted.

- **AvailabilityZones** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster can be created in.

- **BacktrackConsumedChangeRecords** – This is a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.

- **BacktrackWindow** – This is a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.

- **BackupRetentionPeriod** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.

- **Capacity** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Not supported by Neptune.

- **CloneGroupId** – This is a String, of type: `string` (a UTF-8 encoded string).

  Identifies the clone group to which the DB cluster is associated.

- **ClusterCreateTime** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).

- **CopyTagsToSnapshot** – This is a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

If set to `true`, tags are copied to any snapshot of the DB cluster that is created.

- **CrossAccountClone** – This is a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, the DB cluster can be cloned across accounts.

- **DatabaseName** – This is a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the initial database of this DB cluster that was provided at create time, if one was specified when the DB cluster was created. This same name is returned for the life of the DB cluster.

- **DBClusterArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster.

- **DBClusterIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied DB cluster identifier. This identifier is the unique key that identifies a DB cluster.

- **DBClusterMembers** – This is An array of DBClusterMember objects.

  Provides the list of instances that make up the DB cluster.

- **DBClusterParameterGroup** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB cluster parameter group for the DB cluster.

- **DbClusterResourceId** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Region-unique, immutable identifier for the DB cluster. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB cluster is accessed.

- **DBSubnetGroup** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies information on the subnet group associated with the DB cluster, including the name, description, and subnets in the subnet group.

- **DeletionProtection** – This is a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled.

- **EarliestBacktrackTime** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Not supported by Neptune.
- **EarliestRestorableTime** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the earliest time to which a database can be restored with point-in-time restore.
- **EnabledCloudwatchLogsExports** – This is a String, of type: `string` (a UTF-8 encoded string).

  A list of the log types that this DB cluster is configured to export to CloudWatch Logs. Valid log types are: `audit` (to publish audit logs to CloudWatch) and slowquery (to publish slow-query logs to CloudWatch). See Publishing Neptune logs to Amazon CloudWatch logs.
- **Endpoint** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the connection endpoint for the primary instance of the DB cluster.
- **Engine** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB cluster.
- **EngineVersion** – This is a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.
- **GlobalClusterIdentifier** – This is a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.
- **HostedZoneId** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the ID that Amazon Route 53 assigns when you create a hosted zone.
- **IAMDatabaseAuthenticationEnabled** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.
- **IOOptimizedNextAllowedModificationTime** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

The next time you can modify the DB cluster to use the `iopt1` storage type.

- **KmsKeyId** – This is a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster.

- **LatestRestorableTime** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.

- **MultiAZ** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster has instances in multiple Availability Zones.

- **PendingModifiedValues** – This is A ClusterPendingModifiedValues object.

  This data type is used as a response element in the `ModifyDBCluster` operation and contains changes that will be applied during the next maintenance window.

- **PercentProgress** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the progress of the operation as a percentage.

- **Port** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the database engine is listening on.

- **PreferredBackupWindow** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.

- **PreferredMaintenanceWindow** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

- **ReaderEndpoint** – This is a String, of type: `string` (a UTF-8 encoded string).

  The reader endpoint for the DB cluster. The reader endpoint for a DB cluster load-balances connections across the Read Replicas that are available in a DB cluster. As clients request new connections to the reader endpoint, Neptune distributes the connection requests among the Read Replicas in the DB cluster. This functionality can help balance your read workload across multiple Read Replicas in your DB cluster.

If a failover occurs, and the Read Replica that you are connected to is promoted to be the primary instance, your connection is dropped. To continue sending your read workload to other Read Replicas in the cluster, you can then reconnect to the reader endpoint.

- **ReadReplicaIdentifiers** – This is a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB cluster.

- **ReplicationSourceIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ReplicationType** – This is a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ServerlessV2ScalingConfiguration** – This is A ServerlessV2ScalingConfigurationInfo object.

  Shows the scaling configuration for a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

- **Status** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this DB cluster.

- **StorageEncrypted** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster is encrypted.

- **StorageType** – This is a String, of type: `string` (a UTF-8 encoded string).

  The storage type used by the DB cluster.

  Valid Values:

  - **`standard`** – (*the default*) Provides cost-effective database storage for applications with moderate to small I/O usage.
  - **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

    Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.

- **VpcSecurityGroups** – This is An array of VpcSecurityGroupMembership objects.

Provides a list of VPC security groups that the DB cluster belongs to.

`DBCluster` is used as the response element for:

- CreateDBCluster
- DeleteDBCluster
- FailoverDBCluster
- ModifyDBCluster
- PromoteReadReplicaDBCluster
- RestoreDBClusterFromSnapshot
- RestoreDBClusterToPointInTime
- StartDBCluster
- StopDBCluster

DBClusterMember (structure)

Contains information about an instance that is part of a DB cluster.

Fields

- **DBClusterParameterGroupStatus** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the status of the DB cluster parameter group for this member of the DB cluster.
- **DBInstanceIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the instance identifier for this member of the DB cluster.
- **IsClusterWriter** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Value that is `true` if the cluster member is the primary instance for the DB cluster and `false` otherwise.
- **PromotionTier** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  A value that specifies the order in which a Read Replica is promoted to the primary instance after a failure of the existing primary instance.

DBClusterRole (structure)

Describes an Amazon Identity and Access Management (IAM) role that is associated with a DB cluster.

Fields

- **FeatureName** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the feature associated with the Amazon Identity and Access Management (IAM) role. For the list of supported feature names, see the section called "DescribeDBEngineVersions".
- **RoleArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of the IAM role that is associated with the DB cluster.
- **Status** – This is a String, of type: `string` (a UTF-8 encoded string).

  Describes the state of association between the IAM role and the DB cluster. The Status property returns one of the following values:
  - `ACTIVE` - the IAM role ARN is associated with the DB cluster and can be used to access other Amazon services on your behalf.
  - `PENDING` - the IAM role ARN is being associated with the DB cluster.
  - `INVALID` - the IAM role ARN is associated with the DB cluster, but the DB cluster is unable to assume the IAM role in order to access other Amazon services on your behalf.

CloudwatchLogsExportConfiguration (structure)

The configuration setting for the log types to be enabled for export to CloudWatch Logs for a specific DB instance or DB cluster.

The `EnableLogTypes` and `DisableLogTypes` arrays determine which logs will be exported (or not exported) to CloudWatch Logs.

Valid log types are: `audit` (to publish audit logs) and `slowquery` (to publish slow-query logs). See Publishing Neptune logs to Amazon CloudWatch logs.

Fields

- **DisableLogTypes** – This is a String, of type: `string` (a UTF-8 encoded string).

The list of log types to disable.

- **EnableLogTypes** – This is a String, of type: `string` (a UTF-8 encoded string).

  The list of log types to enable.

PendingCloudwatchLogsExports (structure)

A list of the log types whose configuration is still pending. In other words, these log types are in the process of being activated or deactivated.

Valid log types are: `audit` (to publish audit logs) and `slowquery` (to publish slow-query logs). See Publishing Neptune logs to Amazon CloudWatch logs.

Fields

- **LogTypesToDisable** – This is a String, of type: `string` (a UTF-8 encoded string).

  Log types that are in the process of being enabled. After they are enabled, these log types are exported to CloudWatch Logs.

- **LogTypesToEnable** – This is a String, of type: `string` (a UTF-8 encoded string).

  Log types that are in the process of being deactivated. After they are deactivated, these log types aren't exported to CloudWatch Logs.

ClusterPendingModifiedValues (structure)

This data type is used as a response element in the `ModifyDBCluster` operation and contains changes that will be applied during the next maintenance window.

Fields

- **AllocatedStorage** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The allocated storage size in gibibytes (GiB) for database engines. For Neptune, `AllocatedStorage` always returns 1, because Neptune DB cluster storage size isn't fixed, but instead automatically adjusts as needed.

- **BackupRetentionPeriod** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

The number of days for which automatic DB snapshots are retained.

- **DBClusterIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  The DBClusterIdentifier value for the DB cluster.

- **EngineVersion** – This is a String, of type: `string` (a UTF-8 encoded string).

  The database engine version.

- **IAMDatabaseAuthenticationEnabled** – This is a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether mapping of AWS Identity and Access Management (IAM) accounts to database accounts is enabled.

- **Iops** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The Provisioned IOPS (I/O operations per second) value. This setting is only for Multi-AZ DB clusters.

- **PendingCloudwatchLogsExports** – This is A PendingCloudwatchLogsExports object.

  This `PendingCloudwatchLogsExports` structure specifies pending changes to which CloudWatch logs are enabled and which are disabled.

- **StorageType** – This is a String, of type: `string` (a UTF-8 encoded string).

  The pending change in storage type for the DB cluster. Valid Values:

  - **`standard`** – (*the default*) Configures cost-effective database storage for applications with moderate to small I/O usage.
  - **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

    Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.

Neptune Global Database API

Actions:

- CreateGlobalCluster (action)

- DeleteGlobalCluster (action)
- ModifyGlobalCluster (action)
- DescribeGlobalClusters (action)
- FailoverGlobalCluster (action)
- RemoveFromGlobalCluster (action)

Structures:

- GlobalCluster (structure)
- GlobalClusterMember (structure)

CreateGlobalCluster (action)

The AWS CLI name for this API is: `create-global-cluster`.

Creates a Neptune global database spread across multiple Amazon Regions. The global database contains a single primary cluster with read-write capability, and read-only secondary clusters that receive data from the primary cluster through high-speed replication performed by the Neptune storage subsystem.

You can create a global database that is initially empty, and then add a primary cluster and secondary clusters to it, or you can specify an existing Neptune cluster during the create operation to become the primary cluster of the global database.

Request

- **DatabaseName** (in the CLI: `--database-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name for the new global database (up to 64 alpha-numeric characters.
- **DeletionProtection** (in the CLI: `--deletion-protection`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The deletion protection setting for the new global database. The global database can't be deleted when deletion protection is enabled.
- **Engine** (in the CLI: `--engine`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the database engine to be used in the global database.

Valid values: `neptune`

- **EngineVersion** (in the CLI: `--engine-version`) – a String, of type: `string` (a UTF-8 encoded string).

  The Neptune engine version to be used by the global database.

  Valid values: `1.2.0.0` or above.
- **GlobalClusterIdentifier** (in the CLI: `--global-cluster-identifier`) – *Required:* a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  The cluster identifier of the new global database cluster.
- **SourceDBClusterIdentifier** (in the CLI: `--source-db-cluster-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  (*Optional*) The Amazon Resource Name (ARN) of an existing Neptune DB cluster to use as the primary cluster of the new global database.
- **StorageEncrypted** (in the CLI: `--storage-encrypted`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The storage encryption setting for the new global database cluster.

Response

Contains the details of an Amazon Neptune global database.

This data type is used as a response element for the the section called "CreateGlobalCluster", the section called "DescribeGlobalClusters", the section called "ModifyGlobalCluster", the section called "DeleteGlobalCluster", the section called "FailoverGlobalCluster", and the section called "RemoveFromGlobalCluster" actions.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The deletion protection setting for the global database.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  The Neptune database engine used by the global database (`"neptune"`).
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

The Neptune engine version used by the global database.

- **GlobalClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the global database.

- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.

- **GlobalClusterMembers** – An array of GlobalClusterMember objects.

  A list of cluster ARNs and instance ARNs for all the DB clusters that are part of the global database.

- **GlobalClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  An immutable identifier for the global database that is unique within in all regions. This identifier is found in CloudTrail log entries whenever the KMS key for the DB cluster is accessed.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this global database.

- **StorageEncrypted** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The storage encryption setting for the global database.

Errors

- GlobalClusterAlreadyExistsFault
- GlobalClusterQuotaExceededFault
- InvalidDBClusterStateFault
- DBClusterNotFoundFault

DeleteGlobalCluster (action)

The AWS CLI name for this API is: `delete-global-cluster`.

Deletes a global database. The primary and all secondary clusters must already be detached or deleted first.

Request

- **GlobalClusterIdentifier** (in the CLI: `--global-cluster-identifier`) – *Required:* a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  The cluster identifier of the global database cluster being deleted.

Response

Contains the details of an Amazon Neptune global database.

This data type is used as a response element for the the section called "CreateGlobalCluster", the section called "DescribeGlobalClusters", the section called "ModifyGlobalCluster", the section called "DeleteGlobalCluster", the section called "FailoverGlobalCluster", and the section called "RemoveFromGlobalCluster" actions.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The deletion protection setting for the global database.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  The Neptune database engine used by the global database (`"neptune"`).
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  The Neptune engine version used by the global database.
- **GlobalClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the global database.
- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.
- **GlobalClusterMembers** – An array of GlobalClusterMember objects.

A list of cluster ARNs and instance ARNs for all the DB clusters that are part of the global database.

- **GlobalClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  An immutable identifier for the global database that is unique within in all regions. This identifier is found in CloudTrail log entries whenever the KMS key for the DB cluster is accessed.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this global database.

- **StorageEncrypted** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The storage encryption setting for the global database.

Errors

- GlobalClusterNotFoundFault
- InvalidGlobalClusterStateFault

ModifyGlobalCluster (action)

The AWS CLI name for this API is: `modify-global-cluster`.

Modify a setting for an Amazon Neptune global cluster. You can change one or more database configuration parameters by specifying these parameters and their new values in the request.

Request

- **AllowMajorVersionUpgrade** (in the CLI: `--allow-major-version-upgrade`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether major version upgrades are allowed.

  Constraints: You must allow major version upgrades if you specify a value for the `EngineVersion` parameter that is a different major version than the DB cluster's current version.

If you upgrade the major version of a global database, the cluster and DB instance parameter groups are set to the default parameter groups for the new version, so you will need to apply any custom parameter groups after completing the upgrade.

- **DeletionProtection** (in the CLI: `--deletion-protection`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether the global database has deletion protection enabled. The global database cannot be deleted when deletion protection is enabled.

- **EngineVersion** (in the CLI: `--engine-version`) – a String, of type: `string` (a UTF-8 encoded string).

  The version number of the database engine to which you want to upgrade. Changing this parameter will result in an outage. The change is applied during the next maintenance window unless `ApplyImmediately` is enabled.

  To list all of the available Neptune engine versions, use the following command:

  Example

  ```
  aws neptune describe-db-engine-versions \
            --engine neptune \
            --query '*[]|[?SupportsGlobalDatabases == 'true'].[EngineVersion]'
  ```

- **GlobalClusterIdentifier** (in the CLI: `--global-cluster-identifier`) – *Required:* a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  The DB cluster identifier for the global cluster being modified. This parameter is not case-sensitive.

  Constraints: Must match the identifier of an existing global database cluster.

- **NewGlobalClusterIdentifier** (in the CLI: `--new-global-cluster-identifier`) – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  A new cluster identifier to assign to the global database. This value is stored as a lowercase string.

  Constraints:

- Must contain from 1 to 63 letters, numbers, or hyphens.
- The first character must be a letter.
- Can't end with a hyphen or contain two consecutive hyphens

Example: `my-cluster2`

Response

Contains the details of an Amazon Neptune global database.

This data type is used as a response element for the the section called "CreateGlobalCluster", the section called "DescribeGlobalClusters", the section called "ModifyGlobalCluster", the section called "DeleteGlobalCluster", the section called "FailoverGlobalCluster", and the section called "RemoveFromGlobalCluster" actions.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The deletion protection setting for the global database.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  The Neptune database engine used by the global database (`"neptune"`).
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  The Neptune engine version used by the global database.
- **GlobalClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the global database.
- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.
- **GlobalClusterMembers** – An array of GlobalClusterMember objects.

  A list of cluster ARNs and instance ARNs for all the DB clusters that are part of the global database.
- **GlobalClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

An immutable identifier for the global database that is unique within in all regions. This identifier is found in CloudTrail log entries whenever the KMS key for the DB cluster is accessed.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this global database.

- **StorageEncrypted** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The storage encryption setting for the global database.

Errors

- GlobalClusterNotFoundFault
- InvalidGlobalClusterStateFault

DescribeGlobalClusters (action)

The AWS CLI name for this API is: `describe-global-clusters`.

Returns information about Neptune global database clusters. This API supports pagination.

Request

- **GlobalClusterIdentifier** (in the CLI: `--global-cluster-identifier`) – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  The user-supplied DB cluster identifier. If this parameter is specified, only information about the specified DB cluster is returned. This parameter is not case-sensitive.

  Constraints: If supplied, must match an existing DB cluster identifier.

- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  (*Optional*) A pagination token returned by a previous call to `DescribeGlobalClusters`. If this parameter is specified, the response will only include records beyond the marker, up to the number specified by `MaxRecords`.

- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination marker token is included in the response that you can use to retrieve the remaining results.

Default: `100`

Constraints: Minimum 20, maximum 100.

Response

- **GlobalClusters** – An array of GlobalCluster objects.

  The list of global clusters and instances returned by this request.
- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  A pagination token. If this parameter is returned in the response, more records are available, which can be retrieved by one or more additional calls to `DescribeGlobalClusters`.

Errors

- GlobalClusterNotFoundFault

FailoverGlobalCluster (action)

The AWS CLI name for this API is: `failover-global-cluster`.

Initiates the failover process for a Neptune global database.

A failover for a Neptune global database promotes one of secondary read-only DB clusters to be the primary DB cluster and demotes the primary DB cluster to being a secondary (read-only) DB cluster. In other words, the role of the current primary DB cluster and the selected target secondary DB cluster are switched. The selected secondary DB cluster assumes full read/write capabilities for the Neptune global database.

> **Note**
>
> This action applies **only** to Neptune global databases. This action is only intended for use on healthy Neptune global databases with healthy Neptune DB clusters and no region-wide outages, to test disaster recovery scenarios or to reconfigure the global database topology.

Request

- **GlobalClusterIdentifier** (in the CLI: `--global-cluster-identifier`) – *Required:* a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Identifier of the Neptune global database that should be failed over. The identifier is the unique key assigned by the user when the Neptune global database was created. In other words, it's the name of the global database that you want to fail over.

  Constraints: Must match the identifier of an existing Neptune global database.
- **TargetDbClusterIdentifier** (in the CLI: `--target-db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of the secondary Neptune DB cluster that you want to promote to primary for the global database.

Response

Contains the details of an Amazon Neptune global database.

This data type is used as a response element for the the section called “CreateGlobalCluster”, the section called “DescribeGlobalClusters”, the section called “ModifyGlobalCluster”, the section called “DeleteGlobalCluster”, the section called “FailoverGlobalCluster”, and the section called “RemoveFromGlobalCluster” actions.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The deletion protection setting for the global database.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  The Neptune database engine used by the global database (`"neptune"`).

- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  The Neptune engine version used by the global database.
- **GlobalClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the global database.
- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.
- **GlobalClusterMembers** – An array of GlobalClusterMember objects.

  A list of cluster ARNs and instance ARNs for all the DB clusters that are part of the global database.
- **GlobalClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  An immutable identifier for the global database that is unique within in all regions. This identifier is found in CloudTrail log entries whenever the KMS key for the DB cluster is accessed.
- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this global database.
- **StorageEncrypted** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The storage encryption setting for the global database.

Errors

- GlobalClusterNotFoundFault
- InvalidGlobalClusterStateFault
- InvalidDBClusterStateFault
- DBClusterNotFoundFault

RemoveFromGlobalCluster (action)

The AWS CLI name for this API is: `remove-from-global-cluster`.

Detaches a Neptune DB cluster from a Neptune global database. A secondary cluster becomes a normal standalone cluster with read-write capability instead of being read-only, and no longer receives data from a the primary cluster.

Request

- **DbClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) identifying the cluster to be detached from the Neptune global database cluster.
- **GlobalClusterIdentifier** (in the CLI: `--global-cluster-identifier`) – *Required:* a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  The identifier of the Neptune global database from which to detach the specified Neptune DB cluster.

Response

Contains the details of an Amazon Neptune global database.

This data type is used as a response element for the the section called "CreateGlobalCluster", the section called "DescribeGlobalClusters", the section called "ModifyGlobalCluster", the section called "DeleteGlobalCluster", the section called "FailoverGlobalCluster", and the section called "RemoveFromGlobalCluster" actions.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The deletion protection setting for the global database.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  The Neptune database engine used by the global database (`"neptune"`).
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  The Neptune engine version used by the global database.
- **GlobalClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the global database.

- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.
- **GlobalClusterMembers** – An array of GlobalClusterMember objects.

  A list of cluster ARNs and instance ARNs for all the DB clusters that are part of the global database.
- **GlobalClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  An immutable identifier for the global database that is unique within in all regions. This identifier is found in CloudTrail log entries whenever the KMS key for the DB cluster is accessed.
- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this global database.
- **StorageEncrypted** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The storage encryption setting for the global database.

Errors

- GlobalClusterNotFoundFault
- InvalidGlobalClusterStateFault
- DBClusterNotFoundFault

Structures:

GlobalCluster (structure)

Contains the details of an Amazon Neptune global database.

This data type is used as a response element for the the section called “CreateGlobalCluster”, the section called “DescribeGlobalClusters”, the section called “ModifyGlobalCluster”, the section called “DeleteGlobalCluster”, the section called “FailoverGlobalCluster”, and the section called “RemoveFromGlobalCluster” actions.

Fields

- **DeletionProtection** – This is a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The deletion protection setting for the global database.
- **Engine** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Neptune database engine used by the global database (`"neptune"`).
- **EngineVersion** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Neptune engine version used by the global database.
- **GlobalClusterArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the global database.
- **GlobalClusterIdentifier** – This is a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.
- **GlobalClusterMembers** – This is An array of GlobalClusterMember objects.

  A list of cluster ARNs and instance ARNs for all the DB clusters that are part of the global database.
- **GlobalClusterResourceId** – This is a String, of type: `string` (a UTF-8 encoded string).

  An immutable identifier for the global database that is unique within in all regions. This identifier is found in CloudTrail log entries whenever the KMS key for the DB cluster is accessed.
- **Status** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this global database.
- **StorageEncrypted** – This is a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The storage encryption setting for the global database.

`GlobalCluster` is used as the response element for:

- CreateGlobalCluster
- ModifyGlobalCluster
- DeleteGlobalCluster
- RemoveFromGlobalCluster
- FailoverGlobalCluster

GlobalClusterMember (structure)

A data structure with information about any primary and secondary clusters associated with an Neptune global database.

Fields

- **DBClusterArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for each Neptune cluster.
- **IsWriter** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the Neptune cluster is the primary cluster (that is, has read-write capability) for the Neptune global database with which it is associated.
- **Readers** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for each read-only secondary cluster associated with the Neptune global database.

Neptune Instances API

Actions:

- CreateDBInstance (action)
- DeleteDBInstance (action)
- ModifyDBInstance (action)
- RebootDBInstance (action)
- DescribeDBInstances (action)
- DescribeOrderableDBInstanceOptions (action)
- DescribeValidDBInstanceModifications (action)

Structures:

- DBInstance (structure)
- DBInstanceStatusInfo (structure)
- OrderableDBInstanceOption (structure)
- PendingModifiedValues (structure)
- ValidStorageOptions (structure)
- ValidDBInstanceModificationsMessage (structure)

CreateDBInstance (action)

The AWS CLI name for this API is: `create-db-instance`.

Creates a new DB instance.

Request

- **AutoMinorVersionUpgrade** (in the CLI: `--auto-minor-version-upgrade`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates that minor engine upgrades are applied automatically to the DB instance during the maintenance window.

  Default: `true`
- **AvailabilityZone** (in the CLI: `--availability-zone`) – a String, of type: `string` (a UTF-8 encoded string).

  The EC2 Availability Zone that the DB instance is created in

  Default: A random, system-chosen Availability Zone in the endpoint's Amazon Region.

  Example: `us-east-1d`

  Constraint: The AvailabilityZone parameter can't be specified if the MultiAZ parameter is set to `true`. The specified Availability Zone must be in the same Amazon Region as the current endpoint.
- **BackupRetentionPeriod** (in the CLI: `--backup-retention-period`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

The number of days for which automated backups are retained.

Not applicable. The retention period for automated backups is managed by the DB cluster. For more information, see the section called "CreateDBCluster".

Default: 1

Constraints:

- Must be a value from 0 to 35
- Cannot be set to 0 if the DB instance is a source to Read Replicas

- **CopyTagsToSnapshot** (in the CLI: `--copy-tags-to-snapshot`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  True to copy all tags from the DB instance to snapshots of the DB instance, and otherwise false. The default is false.

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the DB cluster that the instance will belong to.

  For information on creating a DB cluster, see the section called "CreateDBCluster".

  Type: String

- **DBInstanceClass** (in the CLI: `--db-instance-class`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The compute and memory capacity of the DB instance, for example, `db.m4.large`. Not all DB instance classes are available in all Amazon Regions.

- **DBInstanceIdentifier** (in the CLI: `--db-instance-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The DB instance identifier. This parameter is stored as a lowercase string.

  Constraints:

  - Must contain from 1 to 63 letters, numbers, or hyphens.
  - First character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.

Example: `mydbinstance`

- **DBName** (in the CLI: `--db-name`) – a String, of type: `string` (a UTF-8 encoded string).

  Not supported.

- **DBParameterGroupName** (in the CLI: `--db-parameter-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB parameter group to associate with this DB instance. If this argument is omitted, the default DBParameterGroup for the specified engine is used.

  Constraints:

  - Must be 1 to 255 letters, numbers, or hyphens.
  - First character must be a letter
  - Cannot end with a hyphen or contain two consecutive hyphens

- **DBSecurityGroups** (in the CLI: `--db-security-groups`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of DB security groups to associate with this DB instance.

  Default: The default DB security group for the database engine.

- **DBSubnetGroupName** (in the CLI: `--db-subnet-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  A DB subnet group to associate with this DB instance.

  If there is no DB subnet group, then it is a non-VPC DB instance.

- **DeletionProtection** (in the CLI: `--deletion-protection`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether the DB instance has deletion protection enabled. The database can't be deleted when deletion protection is enabled. By default, deletion protection is disabled. See Deleting a DB Instance.

  DB instances in a DB cluster can be deleted even when deletion protection is enabled in their parent DB cluster.

- **Domain** (in the CLI: `--domain`) – a String, of type: `string` (a UTF-8 encoded string).

Specify the Active Directory Domain to create the instance in.

- **DomainIAMRoleName** (in the CLI: `--domain-iam-role-name`) – a String, of type: `string` (a UTF-8 encoded string).

  Specify the name of the IAM role to be used when making API calls to the Directory Service.
- **EnableCloudwatchLogsExports** (in the CLI: `--enable-cloudwatch-logs-exports`) – a String, of type: `string` (a UTF-8 encoded string).

  The list of log types that need to be enabled for exporting to CloudWatch Logs.
- **EnableIAMDatabaseAuthentication** (in the CLI: `--enable-iam-database-authentication`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Not supported by Neptune (ignored).
- **Engine** (in the CLI: `--engine`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the database engine to be used for this instance.

  Valid Values: `neptune`
- **EngineVersion** (in the CLI: `--engine-version`) – a String, of type: `string` (a UTF-8 encoded string).

  The version number of the database engine to use. Currently, setting this parameter has no effect.
- **Iops** (in the CLI: `--iops`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The amount of Provisioned IOPS (input/output operations per second) to be initially allocated for the DB instance.
- **KmsKeyId** (in the CLI: `--kms-key-id`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon KMS key identifier for an encrypted DB instance.

  The KMS key identifier is the Amazon Resource Name (ARN) for the KMS encryption key. If you are creating a DB instance with the same Amazon account that owns the KMS encryption key used to encrypt the new DB instance, then you can use the KMS key alias instead of the ARN for the KM encryption key.

  Not applicable. The KMS key identifier is managed by the DB cluster. For more information, see the section called "CreateDBCluster".

If the `StorageEncrypted` parameter is true, and you do not specify a value for the `KmsKeyId` parameter, then Amazon Neptune will use your default encryption key. Amazon KMS creates the default encryption key for your Amazon account. Your Amazon account has a different default encryption key for each Amazon Region.

- **LicenseModel** (in the CLI: `--license-model`) – a String, of type: `string` (a UTF-8 encoded string).

  License model information for this DB instance.

  Valid values: `license-included` | `bring-your-own-license` | `general-public-license`

- **MonitoringInterval** (in the CLI: `--monitoring-interval`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The interval, in seconds, between points when Enhanced Monitoring metrics are collected for the DB instance. To disable collecting Enhanced Monitoring metrics, specify 0. The default is 0.

  If `MonitoringRoleArn` is specified, then you must also set `MonitoringInterval` to a value other than 0.

  Valid Values: `0, 1, 5, 10, 15, 30, 60`

- **MonitoringRoleArn** (in the CLI: `--monitoring-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN for the IAM role that permits Neptune to send enhanced monitoring metrics to Amazon CloudWatch Logs. For example, `arn:aws:iam:123456789012:role/emaccess`.

  If `MonitoringInterval` is set to a value other than 0, then you must supply a `MonitoringRoleArn` value.

- **MultiAZ** (in the CLI: `--multi-az`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Specifies if the DB instance is a Multi-AZ deployment. You can't set the AvailabilityZone parameter if the MultiAZ parameter is set to true.

- **Port** (in the CLI: `--port`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The port number on which the database accepts connections.

Not applicable. The port is managed by the DB cluster. For more information, see the section called "CreateDBCluster".

Default: 8182

Type: Integer

- **PreferredBackupWindow** (in the CLI: `--preferred-backup-window`) – a String, of type: `string` (a UTF-8 encoded string).

  The daily time range during which automated backups are created.

  Not applicable. The daily time range for creating automated backups is managed by the DB cluster. For more information, see the section called "CreateDBCluster".

- **PreferredMaintenanceWindow** (in the CLI: `--preferred-maintenance-window`) – a String, of type: `string` (a UTF-8 encoded string).

  The time range each week during which system maintenance can occur, in Universal Coordinated Time (UTC).

  Format: `ddd:hh24:mi-ddd:hh24:mi`

  The default is a 30-minute window selected at random from an 8-hour block of time for each Amazon Region, occurring on a random day of the week.

  Valid Days: Mon, Tue, Wed, Thu, Fri, Sat, Sun.

  Constraints: Minimum 30-minute window.

- **PromotionTier** (in the CLI: `--promotion-tier`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  A value that specifies the order in which an Read Replica is promoted to the primary instance after a failure of the existing primary instance.

  Default: 1

  Valid Values: 0 - 15

- **PubliclyAccessible** (in the CLI: `--publicly-accessible`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

This flag should no longer be used.

- **StorageEncrypted** (in the CLI: `--storage-encrypted`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB instance is encrypted.

  Not applicable. The encryption for DB instances is managed by the DB cluster. For more information, see the section called "CreateDBCluster".

  Default: false

- **StorageType** (in the CLI: `--storage-type`) – a String, of type: `string` (a UTF-8 encoded string).

  Not applicable. In Neptune the storage type is managed at the DB Cluster level.

- **Tags** (in the CLI: `--tags`) – An array of Tag objects.

  The tags to assign to the new instance.

- **TdeCredentialArn** (in the CLI: `--tde-credential-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN from the key store with which to associate the instance for TDE encryption.

- **TdeCredentialPassword** (in the CLI: `--tde-credential-password`) – a SensitiveString, of type: `string` (a UTF-8 encoded string).

  The password for the given ARN from the key store in order to access the device.

- **Timezone** (in the CLI: `--timezone`) – a String, of type: `string` (a UTF-8 encoded string).

  The time zone of the DB instance.

- **VpcSecurityGroupIds** (in the CLI: `--vpc-security-group-ids`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of EC2 VPC security groups to associate with this DB instance.

  Not applicable. The associated list of EC2 VPC security groups is managed by the DB cluster. For more information, see the section called "CreateDBCluster".

  Default: The default EC2 VPC security group for the DB subnet group's VPC.

Response

Contains the details of an Amazon Neptune DB instance.

This data type is used as a response element in the the section called "DescribeDBInstances" action.

- **AutoMinorVersionUpgrade** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates that minor version patches are applied automatically.
- **AvailabilityZone** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the Availability Zone the DB instance is located in.
- **BackupRetentionPeriod** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.
- **CACertificateIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the CA certificate for this DB instance.
- **CopyTagsToSnapshot** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether tags are copied from the DB instance to snapshots of the DB instance.
- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  If the DB instance is a member of a DB cluster, contains the name of the DB cluster that the DB instance is a member of.
- **DBInstanceArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB instance.
- **DBInstanceClass** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the compute and memory capacity class of the DB instance.
- **DBInstanceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied database identifier. This identifier is the unique key that identifies a DB instance.
- **DbInstancePort** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the DB instance listens on. If the DB instance is part of a DB cluster, this can be a different port than the DB cluster port.

- **DBInstanceStatus** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this database.
- **DbiResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Region-unique, immutable identifier for the DB instance. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB instance is accessed.
- **DBName** – a String, of type: `string` (a UTF-8 encoded string).

  The database name.
- **DBParameterGroups** – An array of DBParameterGroupStatus objects.

  Provides the list of DB parameter groups applied to this DB instance.
- **DBSecurityGroups** – An array of DBSecurityGroupMembership objects.

  Provides List of DB security group elements containing only `DBSecurityGroup.Name` and `DBSecurityGroup.Status` subelements.
- **DBSubnetGroup** – A DBSubnetGroup object.

  Specifies information on the subnet group associated with the DB instance, including the name, description, and subnets in the subnet group.
- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB instance has deletion protection enabled. The instance can't be deleted when deletion protection is enabled. See Deleting a DB Instance.
- **DomainMemberships** – An array of DomainMembership objects.

  Not supported
- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

  A list of log types that this DB instance is configured to export to CloudWatch Logs.
- **Endpoint** – An Endpoint object.

  Specifies the connection endpoint.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB instance.
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

Indicates the database engine version.

- **EnhancedMonitoringResourceArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of the Amazon CloudWatch Logs log stream that receives the Enhanced Monitoring metrics data for the DB instance.

- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if Amazon Identity and Access Management (IAM) authentication is enabled, and otherwise false.

- **InstanceCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Provides the date and time the DB instance was created.

- **Iops** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the Provisioned IOPS (I/O operations per second) value.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported: The encryption for DB instances is managed by the DB cluster.

- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.

- **LicenseModel** – a String, of type: `string` (a UTF-8 encoded string).

  License model information for this DB instance.

- **MonitoringInterval** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The interval, in seconds, between points when Enhanced Monitoring metrics are collected for the DB instance.

- **MonitoringRoleArn** – a String, of type: `string` (a UTF-8 encoded string).

  The ARN for the IAM role that permits Neptune to send Enhanced Monitoring metrics to Amazon CloudWatch Logs.

- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

Specifies if the DB instance is a Multi-AZ deployment.

- **PendingModifiedValues** – A PendingModifiedValues object.

  Specifies that changes to the DB instance are pending. This element is only included when changes are pending. Specific changes are identified by subelements.

- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.

- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

- **PromotionTier** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  A value that specifies the order in which a Read Replica is promoted to the primary instance after a failure of the existing primary instance.

- **PubliclyAccessible** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  This flag should no longer be used.

- **ReadReplicaDBClusterIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of DB clusters that are Read Replicas of this DB instance.

- **ReadReplicaDBInstanceIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB instance.

- **ReadReplicaSourceDBInstanceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the identifier of the source DB instance if this DB instance is a Read Replica.

- **SecondaryAvailabilityZone** – a String, of type: `string` (a UTF-8 encoded string).

  If present, specifies the name of the secondary Availability Zone for a DB instance with multi-AZ support.

- **StatusInfos** – An array of DBInstanceStatusInfo objects.

  The status of a Read Replica. If the instance is not a Read Replica, this is blank.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

Not supported: The encryption for DB instances is managed by the DB cluster.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the storage type associated with the DB instance.

- **TdeCredentialArn** – a String, of type: `string` (a UTF-8 encoded string).

  The ARN from the key store with which the instance is associated for TDE encryption.

- **Timezone** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported.

- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

  Provides a list of VPC security group elements that the DB instance belongs to.

Errors

- DBInstanceAlreadyExistsFault
- InsufficientDBInstanceCapacityFault
- DBParameterGroupNotFoundFault
- DBSecurityGroupNotFoundFault
- InstanceQuotaExceededFault
- StorageQuotaExceededFault
- DBSubnetGroupNotFoundFault
- DBSubnetGroupDoesNotCoverEnoughAZs
- InvalidDBClusterStateFault
- InvalidSubnet
- InvalidVPCNetworkStateFault
- ProvisionedIopsNotAvailableInAZFault
- OptionGroupNotFoundFault
- DBClusterNotFoundFault
- StorageTypeNotSupportedFault
- AuthorizationNotFoundFault
- KMSKeyNotAccessibleFault

- DomainNotFoundFault

DeleteDBInstance (action)

The AWS CLI name for this API is: `delete-db-instance`.

The DeleteDBInstance action deletes a previously provisioned DB instance. When you delete a DB instance, all automated backups for that instance are deleted and can't be recovered. Manual DB snapshots of the DB instance to be deleted by `DeleteDBInstance` are not deleted.

If you request a final DB snapshot the status of the Amazon Neptune DB instance is `deleting` until the DB snapshot is created. The API action `DescribeDBInstance` is used to monitor the status of this operation. The action can't be canceled or reverted once submitted.

Note that when a DB instance is in a failure state and has a status of `failed`, `incompatible-restore`, or `incompatible-network`, you can only delete it when the `SkipFinalSnapshot` parameter is set to `true`.

You can't delete a DB instance if it is the only instance in the DB cluster, or if it has deletion protection enabled.

Request

- **DBInstanceIdentifier** (in the CLI: `--db-instance-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The DB instance identifier for the DB instance to be deleted. This parameter isn't case-sensitive.

  Constraints:

  - Must match the name of an existing DB instance.
- **FinalDBSnapshotIdentifier** (in the CLI: `--final-db-snapshot-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  The DBSnapshotIdentifier of the new DBSnapshot created when SkipFinalSnapshot is set to `false`.

> **Note**
>
> Specifying this parameter and also setting the SkipFinalShapshot parameter to true results in an error.

Constraints:

- Must be 1 to 255 letters or numbers.
- First character must be a letter
- Cannot end with a hyphen or contain two consecutive hyphens
- Cannot be specified when deleting a Read Replica.

- **SkipFinalSnapshot** (in the CLI: `--skip-final-snapshot`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Determines whether a final DB snapshot is created before the DB instance is deleted. If `true` is specified, no DBSnapshot is created. If `false` is specified, a DB snapshot is created before the DB instance is deleted.

  Note that when a DB instance is in a failure state and has a status of 'failed', 'incompatible-restore', or 'incompatible-network', it can only be deleted when the SkipFinalSnapshot parameter is set to "true".

  Specify `true` when deleting a Read Replica.

  > **Note**
  >
  > The FinalDBSnapshotIdentifier parameter must be specified if SkipFinalSnapshot is `false`.

  Default: `false`

Response

Contains the details of an Amazon Neptune DB instance.

This data type is used as a response element in the the section called “DescribeDBInstances” action.

- **AutoMinorVersionUpgrade** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates that minor version patches are applied automatically.
- **AvailabilityZone** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the Availability Zone the DB instance is located in.
- **BackupRetentionPeriod** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.
- **CACertificateIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the CA certificate for this DB instance.
- **CopyTagsToSnapshot** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether tags are copied from the DB instance to snapshots of the DB instance.
- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  If the DB instance is a member of a DB cluster, contains the name of the DB cluster that the DB instance is a member of.
- **DBInstanceArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB instance.
- **DBInstanceClass** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the compute and memory capacity class of the DB instance.
- **DBInstanceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied database identifier. This identifier is the unique key that identifies a DB instance.
- **DbInstancePort** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the DB instance listens on. If the DB instance is part of a DB cluster, this can be a different port than the DB cluster port.
- **DBInstanceStatus** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this database.
- **DbiResourceId** – a String, of type: `string` (a UTF-8 encoded string).

The Amazon Region-unique, immutable identifier for the DB instance. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB instance is accessed.

- **DBName** – a String, of type: `string` (a UTF-8 encoded string).

  The database name.

- **DBParameterGroups** – An array of DBParameterGroupStatus objects.

  Provides the list of DB parameter groups applied to this DB instance.

- **DBSecurityGroups** – An array of DBSecurityGroupMembership objects.

  Provides List of DB security group elements containing only `DBSecurityGroup.Name` and `DBSecurityGroup.Status` subelements.

- **DBSubnetGroup** – A DBSubnetGroup object.

  Specifies information on the subnet group associated with the DB instance, including the name, description, and subnets in the subnet group.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB instance has deletion protection enabled. The instance can't be deleted when deletion protection is enabled. See Deleting a DB Instance.

- **DomainMemberships** – An array of DomainMembership objects.

  Not supported

- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

  A list of log types that this DB instance is configured to export to CloudWatch Logs.

- **Endpoint** – An Endpoint object.

  Specifies the connection endpoint.

- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB instance.

- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.

- **EnhancedMonitoringResourceArn** – a String, of type: `string` (a UTF-8 encoded string).

The Amazon Resource Name (ARN) of the Amazon CloudWatch Logs log stream that receives the Enhanced Monitoring metrics data for the DB instance.

- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if Amazon Identity and Access Management (IAM) authentication is enabled, and otherwise false.

- **InstanceCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Provides the date and time the DB instance was created.

- **Iops** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the Provisioned IOPS (I/O operations per second) value.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported: The encryption for DB instances is managed by the DB cluster.

- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.

- **LicenseModel** – a String, of type: `string` (a UTF-8 encoded string).

  License model information for this DB instance.

- **MonitoringInterval** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The interval, in seconds, between points when Enhanced Monitoring metrics are collected for the DB instance.

- **MonitoringRoleArn** – a String, of type: `string` (a UTF-8 encoded string).

  The ARN for the IAM role that permits Neptune to send Enhanced Monitoring metrics to Amazon CloudWatch Logs.

- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies if the DB instance is a Multi-AZ deployment.

- **PendingModifiedValues** – A PendingModifiedValues object.

Specifies that changes to the DB instance are pending. This element is only included when changes are pending. Specific changes are identified by subelements.

- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.

- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

- **PromotionTier** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  A value that specifies the order in which a Read Replica is promoted to the primary instance after a failure of the existing primary instance.

- **PubliclyAccessible** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  This flag should no longer be used.

- **ReadReplicaDBClusterIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of DB clusters that are Read Replicas of this DB instance.

- **ReadReplicaDBInstanceIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB instance.

- **ReadReplicaSourceDBInstanceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the identifier of the source DB instance if this DB instance is a Read Replica.

- **SecondaryAvailabilityZone** – a String, of type: `string` (a UTF-8 encoded string).

  If present, specifies the name of the secondary Availability Zone for a DB instance with multi-AZ support.

- **StatusInfos** – An array of DBInstanceStatusInfo objects.

  The status of a Read Replica. If the instance is not a Read Replica, this is blank.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Not supported: The encryption for DB instances is managed by the DB cluster.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

Specifies the storage type associated with the DB instance.

- **TdeCredentialArn** – a String, of type: `string` (a UTF-8 encoded string).

  The ARN from the key store with which the instance is associated for TDE encryption.

- **Timezone** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported.

- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

  Provides a list of VPC security group elements that the DB instance belongs to.

Errors

- DBInstanceNotFoundFault
- InvalidDBInstanceStateFault
- DBSnapshotAlreadyExistsFault
- SnapshotQuotaExceededFault
- InvalidDBClusterStateFault

ModifyDBInstance (action)

The AWS CLI name for this API is: `modify-db-instance`.

Modifies settings for a DB instance. You can change one or more database configuration parameters by specifying these parameters and the new values in the request. To learn what modifications you can make to your DB instance, call the section called "DescribeValidDBInstanceModifications" before you call the section called "ModifyDBInstance".

Request

- **AllowMajorVersionUpgrade** (in the CLI: `--allow-major-version-upgrade`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates that major version upgrades are allowed. Changing this parameter doesn't result in an outage and the change is asynchronously applied as soon as possible.

- **ApplyImmediately** (in the CLI: `--apply-immediately`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

Specifies whether the modifications in this request and any pending modifications are asynchronously applied as soon as possible, regardless of the `PreferredMaintenanceWindow` setting for the DB instance.

If this parameter is set to `false`, changes to the DB instance are applied during the next maintenance window. Some parameter changes can cause an outage and are applied on the next call to the section called "RebootDBInstance", or the next failure reboot.

Default: `false`

- **AutoMinorVersionUpgrade** (in the CLI: `--auto-minor-version-upgrade`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates that minor version upgrades are applied automatically to the DB instance during the maintenance window. Changing this parameter doesn't result in an outage except in the following case and the change is asynchronously applied as soon as possible. An outage will result if this parameter is set to `true` during the maintenance window, and a newer minor version is available, and Neptune has enabled auto patching for that engine version.
- **BackupRetentionPeriod** (in the CLI: `--backup-retention-period`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Not applicable. The retention period for automated backups is managed by the DB cluster. For more information, see the section called "ModifyDBCluster".

  Default: Uses existing setting
- **CACertificateIdentifier** (in the CLI: `--ca-certificate-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the certificate that needs to be associated with the instance.
- **CloudwatchLogsExportConfiguration** (in the CLI: `--cloudwatch-logs-export-configuration`) – A CloudwatchLogsExportConfiguration object.

  The configuration setting for the log types to be enabled for export to CloudWatch Logs for a specific DB instance or DB cluster.
- **CopyTagsToSnapshot** (in the CLI: `--copy-tags-to-snapshot`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

True to copy all tags from the DB instance to snapshots of the DB instance, and otherwise false. The default is false.

- **DBInstanceClass** (in the CLI: `--db-instance-class`) – a String, of type: `string` (a UTF-8 encoded string).

  The new compute and memory capacity of the DB instance, for example, `db.m4.large`. Not all DB instance classes are available in all Amazon Regions.

  If you modify the DB instance class, an outage occurs during the change. The change is applied during the next maintenance window, unless `ApplyImmediately` is specified as `true` for this request.

  Default: Uses existing setting

- **DBInstanceIdentifier** (in the CLI: `--db-instance-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The DB instance identifier. This value is stored as a lowercase string.

  Constraints:

  - Must match the identifier of an existing DBInstance.

- **DBParameterGroupName** (in the CLI: `--db-parameter-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB parameter group to apply to the DB instance. Changing this setting doesn't result in an outage. The parameter group name itself is changed immediately, but the actual parameter changes are not applied until you reboot the instance without failover. The db instance will NOT be rebooted automatically and the parameter changes will NOT be applied during the next maintenance window.

  Default: Uses existing setting

  Constraints: The DB parameter group must be in the same DB parameter group family as this DB instance.

- **DBPortNumber** (in the CLI: `--db-port-number`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The port number on which the database accepts connections.

The value of the `DBPortNumber` parameter must not match any of the port values specified for options in the option group for the DB instance.

Your database will restart when you change the `DBPortNumber` value regardless of the value of the `ApplyImmediately` parameter.

Default: `8182`

- **DBSecurityGroups** (in the CLI: `--db-security-groups`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of DB security groups to authorize on this DB instance. Changing this setting doesn't result in an outage and the change is asynchronously applied as soon as possible.

  Constraints:

  - If supplied, must match existing DBSecurityGroups.
- **DBSubnetGroupName** (in the CLI: `--db-subnet-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The new DB subnet group for the DB instance. You can use this parameter to move your DB instance to a different VPC.

  Changing the subnet group causes an outage during the change. The change is applied during the next maintenance window, unless you specify `true` for the `ApplyImmediately` parameter.

  Constraints: If supplied, must match the name of an existing DBSubnetGroup.

  Example: `mySubnetGroup`
- **DeletionProtection** (in the CLI: `--deletion-protection`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether the DB instance has deletion protection enabled. The database can't be deleted when deletion protection is enabled. By default, deletion protection is disabled. See Deleting a DB Instance.
- **Domain** (in the CLI: `--domain`) – a String, of type: `string` (a UTF-8 encoded string).

  Not supported.
- **DomainIAMRoleName** (in the CLI: `--domain-iam-role-name`) – a String, of type: `string` (a UTF-8 encoded string).

Not supported

- **EnableIAMDatabaseAuthentication** (in the CLI: `--enable-iam-database-authentication`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  True to enable mapping of Amazon Identity and Access Management (IAM) accounts to database accounts, and otherwise false.

  You can enable IAM database authentication for the following database engines

  Not applicable. Mapping Amazon IAM accounts to database accounts is managed by the DB cluster. For more information, see the section called "ModifyDBCluster".

  Default: `false`

- **EngineVersion** (in the CLI: `--engine-version`) – a String, of type: `string` (a UTF-8 encoded string).

  The version number of the database engine to upgrade to. Currently, setting this parameter has no effect. To upgrade your database engine to the most recent release, use the the section called "ApplyPendingMaintenanceAction" API.

- **Iops** (in the CLI: `--iops`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The new Provisioned IOPS (I/O operations per second) value for the instance.

  Changing this setting doesn't result in an outage and the change is applied during the next maintenance window unless the `ApplyImmediately` parameter is set to `true` for this request.

  Default: Uses existing setting

- **MonitoringInterval** (in the CLI: `--monitoring-interval`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The interval, in seconds, between points when Enhanced Monitoring metrics are collected for the DB instance. To disable collecting Enhanced Monitoring metrics, specify 0. The default is 0.

  If `MonitoringRoleArn` is specified, then you must also set `MonitoringInterval` to a value other than 0.

  Valid Values: `0, 1, 5, 10, 15, 30, 60`

- **MonitoringRoleArn** (in the CLI: `--monitoring-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN for the IAM role that permits Neptune to send enhanced monitoring metrics to Amazon CloudWatch Logs. For example, `arn:aws:iam:123456789012:role/emaccess`.

  If `MonitoringInterval` is set to a value other than 0, then you must supply a `MonitoringRoleArn` value.
- **MultiAZ** (in the CLI: `--multi-az`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Specifies if the DB instance is a Multi-AZ deployment. Changing this parameter doesn't result in an outage and the change is applied during the next maintenance window unless the `ApplyImmediately` parameter is set to `true` for this request.
- **NewDBInstanceIdentifier** (in the CLI: `--new-db-instance-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  The new DB instance identifier for the DB instance when renaming a DB instance. When you change the DB instance identifier, an instance reboot will occur immediately if you set `Apply Immediately` to true, or will occur during the next maintenance window if `Apply Immediately` to false. This value is stored as a lowercase string.

  Constraints:

  - Must contain from 1 to 63 letters, numbers, or hyphens.
  - The first character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.

  Example: `mydbinstance`
- **PreferredBackupWindow** (in the CLI: `--preferred-backup-window`) – a String, of type: `string` (a UTF-8 encoded string).

  The daily time range during which automated backups are created if automated backups are enabled.

  Not applicable. The daily time range for creating automated backups is managed by the DB cluster. For more information, see the section called "ModifyDBCluster".

  Constraints:

- Must be in the format hh24:mi-hh24:mi
- Must be in Universal Time Coordinated (UTC)
- Must not conflict with the preferred maintenance window
- Must be at least 30 minutes

- **PreferredMaintenanceWindow** (in the CLI: `--preferred-maintenance-window`) – a String, of type: `string` (a UTF-8 encoded string).

  The weekly time range (in UTC) during which system maintenance can occur, which might result in an outage. Changing this parameter doesn't result in an outage, except in the following situation, and the change is asynchronously applied as soon as possible. If there are pending actions that cause a reboot, and the maintenance window is changed to include the current time, then changing this parameter will cause a reboot of the DB instance. If moving this window to the current time, there must be at least 30 minutes between the current time and end of the window to ensure pending changes are applied.

  Default: Uses existing setting

  Format: ddd:hh24:mi-ddd:hh24:mi

  Valid Days: Mon | Tue | Wed | Thu | Fri | Sat | Sun

  Constraints: Must be at least 30 minutes

- **PromotionTier** (in the CLI: `--promotion-tier`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  A value that specifies the order in which a Read Replica is promoted to the primary instance after a failure of the existing primary instance.

  Default: 1

  Valid Values: 0 - 15

- **PubliclyAccessible** (in the CLI: `--publicly-accessible`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  This flag should no longer be used.

- **StorageType** (in the CLI: `--storage-type`) – a String, of type: `string` (a UTF-8 encoded string).

Not applicable. In Neptune the storage type is managed at the DB Cluster level.

- **TdeCredentialArn** (in the CLI: `--tde-credential-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN from the key store with which to associate the instance for TDE encryption.

- **TdeCredentialPassword** (in the CLI: `--tde-credential-password`) – a SensitiveString, of type: `string` (a UTF-8 encoded string).

  The password for the given ARN from the key store in order to access the device.

- **VpcSecurityGroupIds** (in the CLI: `--vpc-security-group-ids`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of EC2 VPC security groups to authorize on this DB instance. This change is asynchronously applied as soon as possible.

  Not applicable. The associated list of EC2 VPC security groups is managed by the DB cluster. For more information, see the section called "ModifyDBCluster".

  Constraints:

  - If supplied, must match existing VpcSecurityGroupIds.

Response

Contains the details of an Amazon Neptune DB instance.

This data type is used as a response element in the the section called "DescribeDBInstances" action.

- **AutoMinorVersionUpgrade** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates that minor version patches are applied automatically.

- **AvailabilityZone** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the Availability Zone the DB instance is located in.

- **BackupRetentionPeriod** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.

- **CACertificateIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

The identifier of the CA certificate for this DB instance.

- **CopyTagsToSnapshot** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether tags are copied from the DB instance to snapshots of the DB instance.

- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  If the DB instance is a member of a DB cluster, contains the name of the DB cluster that the DB instance is a member of.

- **DBInstanceArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB instance.

- **DBInstanceClass** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the compute and memory capacity class of the DB instance.

- **DBInstanceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied database identifier. This identifier is the unique key that identifies a DB instance.

- **DbInstancePort** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the DB instance listens on. If the DB instance is part of a DB cluster, this can be a different port than the DB cluster port.

- **DBInstanceStatus** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this database.

- **DbiResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Region-unique, immutable identifier for the DB instance. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB instance is accessed.

- **DBName** – a String, of type: `string` (a UTF-8 encoded string).

  The database name.

- **DBParameterGroups** – An array of DBParameterGroupStatus objects.

  Provides the list of DB parameter groups applied to this DB instance.

- **DBSecurityGroups** – An array of DBSecurityGroupMembership objects.

Provides List of DB security group elements containing only `DBSecurityGroup.Name` and `DBSecurityGroup.Status` subelements.

- **DBSubnetGroup** – A DBSubnetGroup object.

  Specifies information on the subnet group associated with the DB instance, including the name, description, and subnets in the subnet group.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB instance has deletion protection enabled. The instance can't be deleted when deletion protection is enabled. See Deleting a DB Instance.

- **DomainMemberships** – An array of DomainMembership objects.

  Not supported

- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

  A list of log types that this DB instance is configured to export to CloudWatch Logs.

- **Endpoint** – An Endpoint object.

  Specifies the connection endpoint.

- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB instance.

- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.

- **EnhancedMonitoringResourceArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of the Amazon CloudWatch Logs log stream that receives the Enhanced Monitoring metrics data for the DB instance.

- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if Amazon Identity and Access Management (IAM) authentication is enabled, and otherwise false.

- **InstanceCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

Provides the date and time the DB instance was created.

- **Iops** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the Provisioned IOPS (I/O operations per second) value.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported: The encryption for DB instances is managed by the DB cluster.

- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.

- **LicenseModel** – a String, of type: `string` (a UTF-8 encoded string).

  License model information for this DB instance.

- **MonitoringInterval** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The interval, in seconds, between points when Enhanced Monitoring metrics are collected for the DB instance.

- **MonitoringRoleArn** – a String, of type: `string` (a UTF-8 encoded string).

  The ARN for the IAM role that permits Neptune to send Enhanced Monitoring metrics to Amazon CloudWatch Logs.

- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies if the DB instance is a Multi-AZ deployment.

- **PendingModifiedValues** – A PendingModifiedValues object.

  Specifies that changes to the DB instance are pending. This element is only included when changes are pending. Specific changes are identified by subelements.

- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.

- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

- **PromotionTier** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  A value that specifies the order in which a Read Replica is promoted to the primary instance after a failure of the existing primary instance.
- **PubliclyAccessible** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  This flag should no longer be used.
- **ReadReplicaDBClusterIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of DB clusters that are Read Replicas of this DB instance.
- **ReadReplicaDBInstanceIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB instance.
- **ReadReplicaSourceDBInstanceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the identifier of the source DB instance if this DB instance is a Read Replica.
- **SecondaryAvailabilityZone** – a String, of type: `string` (a UTF-8 encoded string).

  If present, specifies the name of the secondary Availability Zone for a DB instance with multi-AZ support.
- **StatusInfos** – An array of DBInstanceStatusInfo objects.

  The status of a Read Replica. If the instance is not a Read Replica, this is blank.
- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Not supported: The encryption for DB instances is managed by the DB cluster.
- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the storage type associated with the DB instance.
- **TdeCredentialArn** – a String, of type: `string` (a UTF-8 encoded string).

  The ARN from the key store with which the instance is associated for TDE encryption.
- **Timezone** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported.
- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

  Provides a list of VPC security group elements that the DB instance belongs to.

Errors

- InvalidDBInstanceStateFault
- InvalidDBSecurityGroupStateFault
- DBInstanceAlreadyExistsFault
- DBInstanceNotFoundFault
- DBSecurityGroupNotFoundFault
- DBParameterGroupNotFoundFault
- InsufficientDBInstanceCapacityFault
- StorageQuotaExceededFault
- InvalidVPCNetworkStateFault
- ProvisionedIopsNotAvailableInAZFault
- OptionGroupNotFoundFault
- DBUpgradeDependencyFailureFault
- StorageTypeNotSupportedFault
- AuthorizationNotFoundFault
- CertificateNotFoundFault
- DomainNotFoundFault

RebootDBInstance (action)

The AWS CLI name for this API is: `reboot-db-instance`.

You might need to reboot your DB instance, usually for maintenance reasons. For example, if you make certain modifications, or if you change the DB parameter group associated with the DB instance, you must reboot the instance for the changes to take effect.

Rebooting a DB instance restarts the database engine service. Rebooting a DB instance results in a momentary outage, during which the DB instance status is set to rebooting.

Request

- **DBInstanceIdentifier** (in the CLI: `--db-instance-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The DB instance identifier. This parameter is stored as a lowercase string.

Constraints:

- Must match the identifier of an existing DBInstance.

- **ForceFailover** (in the CLI: `--force-failover`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  When `true`, the reboot is conducted through a MultiAZ failover.

  Constraint: You can't specify `true` if the instance is not configured for MultiAZ.

Response

Contains the details of an Amazon Neptune DB instance.

This data type is used as a response element in the the section called "DescribeDBInstances" action.

- **AutoMinorVersionUpgrade** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates that minor version patches are applied automatically.
- **AvailabilityZone** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the Availability Zone the DB instance is located in.
- **BackupRetentionPeriod** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.
- **CACertificateIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the CA certificate for this DB instance.
- **CopyTagsToSnapshot** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether tags are copied from the DB instance to snapshots of the DB instance.
- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  If the DB instance is a member of a DB cluster, contains the name of the DB cluster that the DB instance is a member of.
- **DBInstanceArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB instance.
- **DBInstanceClass** – a String, of type: `string` (a UTF-8 encoded string).

Contains the name of the compute and memory capacity class of the DB instance.

- **DBInstanceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied database identifier. This identifier is the unique key that identifies a DB instance.

- **DbInstancePort** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the DB instance listens on. If the DB instance is part of a DB cluster, this can be a different port than the DB cluster port.

- **DBInstanceStatus** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this database.

- **DbiResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Region-unique, immutable identifier for the DB instance. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB instance is accessed.

- **DBName** – a String, of type: `string` (a UTF-8 encoded string).

  The database name.

- **DBParameterGroups** – An array of DBParameterGroupStatus objects.

  Provides the list of DB parameter groups applied to this DB instance.

- **DBSecurityGroups** – An array of DBSecurityGroupMembership objects.

  Provides List of DB security group elements containing only `DBSecurityGroup.Name` and `DBSecurityGroup.Status` subelements.

- **DBSubnetGroup** – A DBSubnetGroup object.

  Specifies information on the subnet group associated with the DB instance, including the name, description, and subnets in the subnet group.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB instance has deletion protection enabled. The instance can't be deleted when deletion protection is enabled. See Deleting a DB Instance.

- **DomainMemberships** – An array of DomainMembership objects.

  Not supported

- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

  A list of log types that this DB instance is configured to export to CloudWatch Logs.
- **Endpoint** – An Endpoint object.

  Specifies the connection endpoint.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB instance.
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.
- **EnhancedMonitoringResourceArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of the Amazon CloudWatch Logs log stream that receives the Enhanced Monitoring metrics data for the DB instance.
- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if Amazon Identity and Access Management (IAM) authentication is enabled, and otherwise false.
- **InstanceCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Provides the date and time the DB instance was created.
- **Iops** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the Provisioned IOPS (I/O operations per second) value.
- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported: The encryption for DB instances is managed by the DB cluster.
- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.
- **LicenseModel** – a String, of type: `string` (a UTF-8 encoded string).

  License model information for this DB instance.

- **MonitoringInterval** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The interval, in seconds, between points when Enhanced Monitoring metrics are collected for the DB instance.

- **MonitoringRoleArn** – a String, of type: `string` (a UTF-8 encoded string).

  The ARN for the IAM role that permits Neptune to send Enhanced Monitoring metrics to Amazon CloudWatch Logs.

- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies if the DB instance is a Multi-AZ deployment.

- **PendingModifiedValues** – A PendingModifiedValues object.

  Specifies that changes to the DB instance are pending. This element is only included when changes are pending. Specific changes are identified by subelements.

- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.

- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

- **PromotionTier** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  A value that specifies the order in which a Read Replica is promoted to the primary instance after a failure of the existing primary instance.

- **PubliclyAccessible** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  This flag should no longer be used.

- **ReadReplicaDBClusterIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of DB clusters that are Read Replicas of this DB instance.

- **ReadReplicaDBInstanceIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB instance.

- **ReadReplicaSourceDBInstanceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

Contains the identifier of the source DB instance if this DB instance is a Read Replica.

- **SecondaryAvailabilityZone** – a String, of type: `string` (a UTF-8 encoded string).

  If present, specifies the name of the secondary Availability Zone for a DB instance with multi-AZ support.

- **StatusInfos** – An array of DBInstanceStatusInfo objects.

  The status of a Read Replica. If the instance is not a Read Replica, this is blank.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Not supported: The encryption for DB instances is managed by the DB cluster.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the storage type associated with the DB instance.

- **TdeCredentialArn** – a String, of type: `string` (a UTF-8 encoded string).

  The ARN from the key store with which the instance is associated for TDE encryption.

- **Timezone** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported.

- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

  Provides a list of VPC security group elements that the DB instance belongs to.

Errors

- InvalidDBInstanceStateFault
- DBInstanceNotFoundFault

DescribeDBInstances (action)

The AWS CLI name for this API is: `describe-db-instances`.

Returns information about provisioned instances, and supports pagination.

> **Note**
>
> This operation can also return information for Amazon RDS instances and Amazon DocDB instances.

Request

- **DBInstanceIdentifier** (in the CLI: `--db-instance-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  The user-supplied instance identifier. If this parameter is specified, information from only the specific DB instance is returned. This parameter isn't case-sensitive.

  Constraints:

  - If supplied, must match the identifier of an existing DBInstance.
- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  A filter that specifies one or more DB instances to describe.

  Supported filters:

  - `db-cluster-id` - Accepts DB cluster identifiers and DB cluster Amazon Resource Names (ARNs). The results list will only include information about the DB instances associated with the DB clusters identified by these ARNs.
  - `engine` - Accepts an engine name (such as `neptune`), and restricts the results list to DB instances created by that engine.

  For example, to invoke this API from the Amazon CLI and filter so that only Neptune DB instances are returned, you could use the following command:

  Example

  ```
  aws neptune describe-db-instances \
              --filters  Name=engine,Values=neptune
  ```

- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

An optional pagination token provided by a previous `DescribeDBInstances` request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

  Default: 100

  Constraints: Minimum 20, maximum 100.

Response

- **DBInstances** – An array of DBInstance objects.

  A list of the section called "DBInstance" instances.
- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords` .

Errors

- DBInstanceNotFoundFault

DescribeOrderableDBInstanceOptions (action)

The AWS CLI name for this API is: `describe-orderable-db-instance-options`.

Returns a list of orderable DB instance options for the specified engine.

Request

- **DBInstanceClass** (in the CLI: `--db-instance-class`) – a String, of type: `string` (a UTF-8 encoded string).

The DB instance class filter value. Specify this parameter to show only the available offerings matching the specified DB instance class.

- **Engine** (in the CLI: `--engine`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the engine to retrieve DB instance options for.

- **EngineVersion** (in the CLI: `--engine-version`) – a String, of type: `string` (a UTF-8 encoded string).

  The engine version filter value. Specify this parameter to show only the available offerings matching the specified engine version.

- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  This parameter is not currently supported.

- **LicenseModel** (in the CLI: `--license-model`) – a String, of type: `string` (a UTF-8 encoded string).

  The license model filter value. Specify this parameter to show only the available offerings matching the specified license model.

- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous DescribeOrderableDBInstanceOptions request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords` .

- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

  Default: 100

  Constraints: Minimum 20, maximum 100.

- **Vpc** (in the CLI: `--vpc`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  The VPC filter value. Specify this parameter to show only the available VPC or non-VPC offerings.

Response

- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous OrderableDBInstanceOptions request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords` .

- **OrderableDBInstanceOptions** – An array of OrderableDBInstanceOption objects.

  An the section called "OrderableDBInstanceOption" structure containing information about orderable options for the DB instance.

DescribeValidDBInstanceModifications (action)

The AWS CLI name for this API is: `describe-valid-db-instance-modifications`.

You can call the section called "DescribeValidDBInstanceModifications" to learn what modifications you can make to your DB instance. You can use this information when you call the section called "ModifyDBInstance".

Request

- **DBInstanceIdentifier** (in the CLI: `--db-instance-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The customer identifier or the ARN of your DB instance.

Response

Information about valid modifications that you can make to your DB instance. Contains the result of a successful call to the the section called "DescribeValidDBInstanceModifications" action. You can use this information when you call the section called "ModifyDBInstance".

- **Storage** – An array of ValidStorageOptions objects.

  Valid storage options for your DB instance.

Errors

- DBInstanceNotFoundFault
- InvalidDBInstanceStateFault

Structures:

DBInstance (structure)

Contains the details of an Amazon Neptune DB instance.

This data type is used as a response element in the the section called "DescribeDBInstances" action.

Fields

- **AutoMinorVersionUpgrade** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates that minor version patches are applied automatically.
- **AvailabilityZone** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the Availability Zone the DB instance is located in.
- **BackupRetentionPeriod** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.
- **CACertificateIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the CA certificate for this DB instance.
- **CopyTagsToSnapshot** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether tags are copied from the DB instance to snapshots of the DB instance.
- **DBClusterIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  If the DB instance is a member of a DB cluster, contains the name of the DB cluster that the DB instance is a member of.
- **DBInstanceArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB instance.

- **DBInstanceClass** – This is a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the compute and memory capacity class of the DB instance.
- **DBInstanceIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied database identifier. This identifier is the unique key that identifies a DB instance.
- **DbInstancePort** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the DB instance listens on. If the DB instance is part of a DB cluster, this can be a different port than the DB cluster port.
- **DBInstanceStatus** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this database.
- **DbiResourceId** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Region-unique, immutable identifier for the DB instance. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB instance is accessed.
- **DBName** – This is a String, of type: `string` (a UTF-8 encoded string).

  The database name.
- **DBParameterGroups** – This is An array of DBParameterGroupStatus objects.

  Provides the list of DB parameter groups applied to this DB instance.
- **DBSecurityGroups** – This is An array of DBSecurityGroupMembership objects.

  Provides List of DB security group elements containing only `DBSecurityGroup.Name` and `DBSecurityGroup.Status` subelements.
- **DBSubnetGroup** – This is A DBSubnetGroup object.

  Specifies information on the subnet group associated with the DB instance, including the name, description, and subnets in the subnet group.
- **DeletionProtection** – This is a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB instance has deletion protection enabled. The instance can't be deleted when deletion protection is enabled. See Deleting a DB Instance.
- **DomainMemberships** – This is An array of DomainMembership objects.

Not supported

- **EnabledCloudwatchLogsExports** – This is a String, of type: `string` (a UTF-8 encoded string).

  A list of log types that this DB instance is configured to export to CloudWatch Logs.
- **Endpoint** – This is An Endpoint object.

  Specifies the connection endpoint.
- **Engine** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB instance.
- **EngineVersion** – This is a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.
- **EnhancedMonitoringResourceArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of the Amazon CloudWatch Logs log stream that receives the Enhanced Monitoring metrics data for the DB instance.
- **IAMDatabaseAuthenticationEnabled** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if Amazon Identity and Access Management (IAM) authentication is enabled, and otherwise false.
- **InstanceCreateTime** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Provides the date and time the DB instance was created.
- **Iops** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the Provisioned IOPS (I/O operations per second) value.
- **KmsKeyId** – This is a String, of type: `string` (a UTF-8 encoded string).

  Not supported: The encryption for DB instances is managed by the DB cluster.
- **LatestRestorableTime** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.
- **LicenseModel** – This is a String, of type: `string` (a UTF-8 encoded string).

License model information for this DB instance.

- **MonitoringInterval** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The interval, in seconds, between points when Enhanced Monitoring metrics are collected for the DB instance.

- **MonitoringRoleArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The ARN for the IAM role that permits Neptune to send Enhanced Monitoring metrics to Amazon CloudWatch Logs.

- **MultiAZ** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies if the DB instance is a Multi-AZ deployment.

- **PendingModifiedValues** – This is A PendingModifiedValues object.

  Specifies that changes to the DB instance are pending. This element is only included when changes are pending. Specific changes are identified by subelements.

- **PreferredBackupWindow** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.

- **PreferredMaintenanceWindow** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

- **PromotionTier** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  A value that specifies the order in which a Read Replica is promoted to the primary instance after a failure of the existing primary instance.

- **PubliclyAccessible** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  This flag should no longer be used.

- **ReadReplicaDBClusterIdentifiers** – This is a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of DB clusters that are Read Replicas of this DB instance.

- **ReadReplicaDBInstanceIdentifiers** – This is a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB instance.

- **ReadReplicaSourceDBInstanceIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  Contains the identifier of the source DB instance if this DB instance is a Read Replica.
- **SecondaryAvailabilityZone** – This is a String, of type: `string` (a UTF-8 encoded string).

  If present, specifies the name of the secondary Availability Zone for a DB instance with multi-AZ support.
- **StatusInfos** – This is An array of DBInstanceStatusInfo objects.

  The status of a Read Replica. If the instance is not a Read Replica, this is blank.
- **StorageEncrypted** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Not supported: The encryption for DB instances is managed by the DB cluster.
- **StorageType** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the storage type associated with the DB instance.
- **TdeCredentialArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The ARN from the key store with which the instance is associated for TDE encryption.
- **Timezone** – This is a String, of type: `string` (a UTF-8 encoded string).

  Not supported.
- **VpcSecurityGroups** – This is An array of VpcSecurityGroupMembership objects.

  Provides a list of VPC security group elements that the DB instance belongs to.

`DBInstance` is used as the response element for:

- CreateDBInstance
- DeleteDBInstance
- ModifyDBInstance
- RebootDBInstance

DBInstanceStatusInfo (structure)

Provides a list of status information for a DB instance.

Fields

- **Message** – This is a String, of type: `string` (a UTF-8 encoded string).

  Details of the error if there is an error for the instance. If the instance is not in an error state, this value is blank.
- **Normal** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Boolean value that is true if the instance is operating normally, or false if the instance is in an error state.
- **Status** – This is a String, of type: `string` (a UTF-8 encoded string).

  Status of the DB instance. For a StatusType of read replica, the values can be replicating, error, stopped, or terminated.
- **StatusType** – This is a String, of type: `string` (a UTF-8 encoded string).

  This value is currently "read replication."

OrderableDBInstanceOption (structure)

Contains a list of available options for a DB instance.

This data type is used as a response element in the the section called "DescribeOrderableDBInstanceOptions" action.

Fields

- **AvailabilityZones** – This is An array of AvailabilityZone objects.

  A list of Availability Zones for a DB instance.
- **DBInstanceClass** – This is a String, of type: `string` (a UTF-8 encoded string).

  The DB instance class for a DB instance.
- **Engine** – This is a String, of type: `string` (a UTF-8 encoded string).

  The engine type of a DB instance.
- **EngineVersion** – This is a String, of type: `string` (a UTF-8 encoded string).

  The engine version of a DB instance.

- **LicenseModel** – This is a String, of type: `string` (a UTF-8 encoded string).

  The license model for a DB instance.
- **MaxIopsPerDbInstance** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Maximum total provisioned IOPS for a DB instance.
- **MaxIopsPerGib** – This is a DoubleOptional, of type: `double` (a double-precisionn IEEE 754 floating-point number).

  Maximum provisioned IOPS per GiB for a DB instance.
- **MaxStorageSize** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Maximum storage size for a DB instance.
- **MinIopsPerDbInstance** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Minimum total provisioned IOPS for a DB instance.
- **MinIopsPerGib** – This is a DoubleOptional, of type: `double` (a double-precisionn IEEE 754 floating-point number).

  Minimum provisioned IOPS per GiB for a DB instance.
- **MinStorageSize** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Minimum storage size for a DB instance.
- **MultiAZCapable** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether a DB instance is Multi-AZ capable.
- **ReadReplicaCapable** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether a DB instance can have a Read Replica.
- **StorageType** – This is a String, of type: `string` (a UTF-8 encoded string).

  Not applicable. In Neptune the storage type is managed at the DB Cluster level.
- **SupportsEnhancedMonitoring** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether a DB instance supports Enhanced Monitoring at intervals from 1 to 60 seconds.
- **SupportsGlobalDatabases** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

A value that indicates whether you can use Neptune global databases with a specific combination of other DB engine attributes.

- **SupportsIAMDatabaseAuthentication** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether a DB instance supports IAM database authentication.

- **SupportsIops** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether a DB instance supports provisioned IOPS.

- **SupportsStorageEncryption** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether a DB instance supports encrypted storage.

- **Vpc** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether a DB instance is in a VPC.

PendingModifiedValues (structure)

This data type is used as a response element in the the section called "ModifyDBInstance" action.

Fields

- **AllocatedStorage** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Contains the new `AllocatedStorage` size for the DB instance that will be applied or is currently being applied.

- **BackupRetentionPeriod** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the pending number of days for which automated backups are retained.

- **CACertificateIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the identifier of the CA certificate for the DB instance.

- **DBInstanceClass** – This is a String, of type: `string` (a UTF-8 encoded string).

  Contains the new `DBInstanceClass` for the DB instance that will be applied or is currently being applied.

- **DBInstanceIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  Contains the new `DBInstanceIdentifier` for the DB instance that will be applied or is currently being applied.
- **DBSubnetGroupName** – This is a String, of type: `string` (a UTF-8 encoded string).

  The new DB subnet group for the DB instance.
- **EngineVersion** – This is a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.
- **Iops** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the new Provisioned IOPS value for the DB instance that will be applied or is currently being applied.
- **MultiAZ** – This is a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates that the Single-AZ DB instance is to change to a Multi-AZ deployment.
- **PendingCloudwatchLogsExports** – This is A PendingCloudwatchLogsExports object.

  This `PendingCloudwatchLogsExports` structure specifies pending changes to which CloudWatch logs are enabled and which are disabled.
- **Port** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the pending port for the DB instance.
- **StorageType** – This is a String, of type: `string` (a UTF-8 encoded string).

  Not applicable. In Neptune the storage type is managed at the DB Cluster level.

ValidStorageOptions (structure)

Not applicable. In Neptune the storage type is managed at the DB Cluster level.

Fields

- **IopsToStorageRatio** – This is An array of DoubleRange objects.

  Not applicable. In Neptune the storage type is managed at the DB Cluster level.
- **ProvisionedIops** – This is An array of Range objects.

Not applicable. In Neptune the storage type is managed at the DB Cluster level.

- **StorageSize** – This is An array of Range objects.

  Not applicable. In Neptune the storage type is managed at the DB Cluster level.

- **StorageType** – This is a String, of type: `string` (a UTF-8 encoded string).

  Not applicable. In Neptune the storage type is managed at the DB Cluster level.

ValidDBInstanceModificationsMessage (structure)

Information about valid modifications that you can make to your DB instance. Contains the result of a successful call to the the section called "DescribeValidDBInstanceModifications" action. You can use this information when you call the section called "ModifyDBInstance".

Fields

- **Storage** – This is An array of ValidStorageOptions objects.

  Valid storage options for your DB instance.

`ValidDBInstanceModificationsMessage` is used as the response element for:

- DescribeValidDBInstanceModifications

Neptune Parameters API

Actions:

- CopyDBParameterGroup (action)
- CopyDBClusterParameterGroup (action)
- CreateDBParameterGroup (action)
- CreateDBClusterParameterGroup (action)
- DeleteDBParameterGroup (action)
- DeleteDBClusterParameterGroup (action)
- ModifyDBParameterGroup (action)

- ModifyDBClusterParameterGroup (action)
- ResetDBParameterGroup (action)
- ResetDBClusterParameterGroup (action)
- DescribeDBParameters (action)
- DescribeDBParameterGroups (action)
- DescribeDBClusterParameters (action)
- DescribeDBClusterParameterGroups (action)
- DescribeEngineDefaultParameters (action)
- DescribeEngineDefaultClusterParameters (action)

Structures:

- Parameter (structure)
- DBParameterGroup (structure)
- DBClusterParameterGroup (structure)
- DBParameterGroupStatus (structure)

CopyDBParameterGroup (action)

The AWS CLI name for this API is: `copy-db-parameter-group`.

Copies the specified DB parameter group.

Request

- **SourceDBParameterGroupIdentifier** (in the CLI: `--source-db-parameter-group-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier or ARN for the source DB parameter group. For information about creating an ARN, see Constructing an Amazon Resource Name (ARN).

  Constraints:

  - Must specify a valid DB parameter group.
  - Must specify a valid DB parameter group identifier, for example `my-db-param-group`, or a valid ARN.

- **Tags** (in the CLI: `--tags`) – An array of Tag objects.

  The tags to be assigned to the copied DB parameter group.
- **TargetDBParameterGroupDescription** (in the CLI: `--target-db-parameter-group-description`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A description for the copied DB parameter group.
- **TargetDBParameterGroupIdentifier** (in the CLI: `--target-db-parameter-group-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier for the copied DB parameter group.

  Constraints:
  - Cannot be null, empty, or blank.
  - Must contain from 1 to 255 letters, numbers, or hyphens.
  - First character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.

  Example: `my-db-parameter-group`

Response

Contains the details of an Amazon Neptune DB parameter group.

This data type is used as a response element in the the section called "DescribeDBParameterGroups" action.

- **DBParameterGroupArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB parameter group.
- **DBParameterGroupFamily** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB parameter group family that this DB parameter group is compatible with.
- **DBParameterGroupName** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB parameter group.
- **Description** – a String, of type: `string` (a UTF-8 encoded string).

Provides the customer-specified description for this DB parameter group.

Errors

- DBParameterGroupNotFoundFault
- DBParameterGroupAlreadyExistsFault
- DBParameterGroupQuotaExceededFault

CopyDBClusterParameterGroup (action)

The AWS CLI name for this API is: `copy-db-cluster-parameter-group`.

Copies the specified DB cluster parameter group.

Request

- **SourceDBClusterParameterGroupIdentifier** (in the CLI: `--source-db-cluster-parameter-group-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier or Amazon Resource Name (ARN) for the source DB cluster parameter group. For information about creating an ARN, see Constructing an Amazon Resource Name (ARN).

  Constraints:

  - Must specify a valid DB cluster parameter group.
  - If the source DB cluster parameter group is in the same Amazon Region as the copy, specify a valid DB parameter group identifier, for example `my-db-cluster-param-group`, or a valid ARN.
  - If the source DB parameter group is in a different Amazon Region than the copy, specify a valid DB cluster parameter group ARN, for example `arn:aws:rds:us-east-1:123456789012:cluster-pg:custom-cluster-group1`.
- **Tags** (in the CLI: `--tags`) – An array of Tag objects.

  The tags to be assigned to the copied DB cluster parameter group.
- **TargetDBClusterParameterGroupDescription** (in the CLI: `--target-db-cluster-parameter-group-description`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

A description for the copied DB cluster parameter group.

- **TargetDBClusterParameterGroupIdentifier** (in the CLI: `--target-db-cluster-parameter-group-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier for the copied DB cluster parameter group.

  Constraints:

  - Cannot be null, empty, or blank
  - Must contain from 1 to 255 letters, numbers, or hyphens
  - First character must be a letter
  - Cannot end with a hyphen or contain two consecutive hyphens

  Example: `my-cluster-param-group1`

Response

Contains the details of an Amazon Neptune DB cluster parameter group.

This data type is used as a response element in the the section called "DescribeDBClusterParameterGroups" action.

- **DBClusterParameterGroupArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster parameter group.
- **DBClusterParameterGroupName** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB cluster parameter group.
- **DBParameterGroupFamily** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB parameter group family that this DB cluster parameter group is compatible with.
- **Description** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the customer-specified description for this DB cluster parameter group.

Errors

- DBParameterGroupNotFoundFault
- DBParameterGroupQuotaExceededFault
- DBParameterGroupAlreadyExistsFault

CreateDBParameterGroup (action)

The AWS CLI name for this API is: `create-db-parameter-group`.

Creates a new DB parameter group.

A DB parameter group is initially created with the default parameters for the database engine used by the DB instance. To provide custom values for any of the parameters, you must modify the group after creating it using *ModifyDBParameterGroup*. Once you've created a DB parameter group, you need to associate it with your DB instance using *ModifyDBInstance*. When you associate a new DB parameter group with a running DB instance, you need to reboot the DB instance without failover for the new DB parameter group and associated settings to take effect.

> **Important**
>
> After you create a DB parameter group, you should wait at least 5 minutes before creating your first DB instance that uses that DB parameter group as the default parameter group. This allows Amazon Neptune to fully complete the create action before the parameter group is used as the default for a new DB instance. This is especially important for parameters that are critical when creating the default database for a DB instance, such as the character set for the default database defined by the `character_set_database` parameter. You can use the *Parameter Groups* option of the Amazon Neptune console or the *DescribeDBParameters* command to verify that your DB parameter group has been created or modified.

Request

- **DBParameterGroupFamily** (in the CLI: `--db-parameter-group-family`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

The DB parameter group family name. A DB parameter group can be associated with one and only one DB parameter group family, and can be applied only to a DB instance running a database engine and engine version compatible with that DB parameter group family.

- **DBParameterGroupName** (in the CLI: `--db-parameter-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB parameter group.

  Constraints:

  - Must be 1 to 255 letters, numbers, or hyphens.
  - First character must be a letter
  - Cannot end with a hyphen or contain two consecutive hyphens

  > **Note**
  >
  > This value is stored as a lowercase string.

- **Description** (in the CLI: `--description`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The description for the DB parameter group.

- **Tags** (in the CLI: `--tags`) – An array of Tag objects.

  The tags to be assigned to the new DB parameter group.

Response

Contains the details of an Amazon Neptune DB parameter group.

This data type is used as a response element in the the section called "DescribeDBParameterGroups" action.

- **DBParameterGroupArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB parameter group.

- **DBParameterGroupFamily** – a String, of type: `string` (a UTF-8 encoded string).

Provides the name of the DB parameter group family that this DB parameter group is compatible with.

- **DBParameterGroupName** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB parameter group.

- **Description** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the customer-specified description for this DB parameter group.

Errors

- DBParameterGroupQuotaExceededFault
- DBParameterGroupAlreadyExistsFault

CreateDBClusterParameterGroup (action)

The AWS CLI name for this API is: `create-db-cluster-parameter-group`.

Creates a new DB cluster parameter group.

Parameters in a DB cluster parameter group apply to all of the instances in a DB cluster.

A DB cluster parameter group is initially created with the default parameters for the database engine used by instances in the DB cluster. To provide custom values for any of the parameters, you must modify the group after creating it using the section called "ModifyDBClusterParameterGroup". Once you've created a DB cluster parameter group, you need to associate it with your DB cluster using the section called "ModifyDBCluster". When you associate a new DB cluster parameter group with a running DB cluster, you need to reboot the DB instances in the DB cluster without failover for the new DB cluster parameter group and associated settings to take effect.

> **Important**
>
> After you create a DB cluster parameter group, you should wait at least 5 minutes before creating your first DB cluster that uses that DB cluster parameter group as the default parameter group. This allows Amazon Neptune to fully complete the create action before the DB cluster parameter group is used as the default for a new DB cluster. This is especially important for parameters that are critical when creating the default

database for a DB cluster, such as the character set for the default database defined by the `character_set_database` parameter. You can use the *Parameter Groups* option of the Amazon Neptune console or the the section called "DescribeDBClusterParameters" command to verify that your DB cluster parameter group has been created or modified.

Request

- **DBClusterParameterGroupName** (in the CLI: `--db-cluster-parameter-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster parameter group.

  Constraints:

  - Must match the name of an existing DBClusterParameterGroup.

  > **Note**
  >
  > This value is stored as a lowercase string.

- **DBParameterGroupFamily** (in the CLI: `--db-parameter-group-family`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The DB cluster parameter group family name. A DB cluster parameter group can be associated with one and only one DB cluster parameter group family, and can be applied only to a DB cluster running a database engine and engine version compatible with that DB cluster parameter group family.

- **Description** (in the CLI: `--description`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The description for the DB cluster parameter group.

- **Tags** (in the CLI: `--tags`) – An array of Tag objects.

  The tags to be assigned to the new DB cluster parameter group.

Response

Contains the details of an Amazon Neptune DB cluster parameter group.

This data type is used as a response element in the the section called "DescribeDBClusterParameterGroups" action.

- **DBClusterParameterGroupArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster parameter group.
- **DBClusterParameterGroupName** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB cluster parameter group.
- **DBParameterGroupFamily** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB parameter group family that this DB cluster parameter group is compatible with.
- **Description** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the customer-specified description for this DB cluster parameter group.

Errors

- DBParameterGroupQuotaExceededFault
- DBParameterGroupAlreadyExistsFault

DeleteDBParameterGroup (action)

The AWS CLI name for this API is: `delete-db-parameter-group`.

Deletes a specified DBParameterGroup. The DBParameterGroup to be deleted can't be associated with any DB instances.

Request

- **DBParameterGroupName** (in the CLI: `--db-parameter-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB parameter group.

  Constraints:
  - Must be the name of an existing DB parameter group
  - You can't delete a default DB parameter group

- Cannot be associated with any DB instances

Response

- *No Response parameters.*

Errors

- InvalidDBParameterGroupStateFault
- DBParameterGroupNotFoundFault

DeleteDBClusterParameterGroup (action)

The AWS CLI name for this API is: `delete-db-cluster-parameter-group`.

Deletes a specified DB cluster parameter group. The DB cluster parameter group to be deleted can't be associated with any DB clusters.

Request

- **DBClusterParameterGroupName** (in the CLI: `--db-cluster-parameter-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster parameter group.

  Constraints:

  - Must be the name of an existing DB cluster parameter group.
  - You can't delete a default DB cluster parameter group.
  - Cannot be associated with any DB clusters.

Response

- *No Response parameters.*

Errors

- InvalidDBParameterGroupStateFault

- DBParameterGroupNotFoundFault

ModifyDBParameterGroup (action)

The AWS CLI name for this API is: `modify-db-parameter-group`.

Modifies the parameters of a DB parameter group. To modify more than one parameter, submit a list of the following: `ParameterName`, `ParameterValue`, and `ApplyMethod`. A maximum of 20 parameters can be modified in a single request.

> **Note**
>
> Changes to dynamic parameters are applied immediately. Changes to static parameters require a reboot without failover to the DB instance associated with the parameter group before the change can take effect.

> **Important**
>
> After you modify a DB parameter group, you should wait at least 5 minutes before creating your first DB instance that uses that DB parameter group as the default parameter group. This allows Amazon Neptune to fully complete the modify action before the parameter group is used as the default for a new DB instance. This is especially important for parameters that are critical when creating the default database for a DB instance, such as the character set for the default database defined by the `character_set_database` parameter. You can use the *Parameter Groups* option of the Amazon Neptune console or the *DescribeDBParameters* command to verify that your DB parameter group has been created or modified.

Request

- **DBParameterGroupName** (in the CLI: `--db-parameter-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB parameter group.

  Constraints:

- If supplied, must match the name of an existing DBParameterGroup.
- **Parameters** (in the CLI: `--parameters`) – *Required:* An array of Parameter objects.

  An array of parameter names, values, and the apply method for the parameter update. At least one parameter name, value, and apply method must be supplied; subsequent arguments are optional. A maximum of 20 parameters can be modified in a single request.

  Valid Values (for the application method): `immediate | pending-reboot`

  > **Note**
  > You can use the immediate value with dynamic parameters only. You can use the pending-reboot value for both dynamic and static parameters, and changes are applied when you reboot the DB instance without failover.

Response

- **DBParameterGroupName** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB parameter group.

Errors

- DBParameterGroupNotFoundFault
- InvalidDBParameterGroupStateFault

ModifyDBClusterParameterGroup (action)

The AWS CLI name for this API is: `modify-db-cluster-parameter-group`.

Modifies the parameters of a DB cluster parameter group. To modify more than one parameter, submit a list of the following: `ParameterName`, `ParameterValue`, and `ApplyMethod`. A maximum of 20 parameters can be modified in a single request.

> **Note**
>
> Changes to dynamic parameters are applied immediately. Changes to static parameters require a reboot without failover to the DB cluster associated with the parameter group before the change can take effect.

> **Important**
>
> After you create a DB cluster parameter group, you should wait at least 5 minutes before creating your first DB cluster that uses that DB cluster parameter group as the default parameter group. This allows Amazon Neptune to fully complete the create action before the parameter group is used as the default for a new DB cluster. This is especially important for parameters that are critical when creating the default database for a DB cluster, such as the character set for the default database defined by the `character_set_database` parameter. You can use the *Parameter Groups* option of the Amazon Neptune console or the the section called "DescribeDBClusterParameters" command to verify that your DB cluster parameter group has been created or modified.

Request

- **DBClusterParameterGroupName** (in the CLI: `--db-cluster-parameter-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster parameter group to modify.
- **Parameters** (in the CLI: `--parameters`) – *Required:* An array of Parameter objects.

  A list of parameters in the DB cluster parameter group to modify.

Response

- **DBClusterParameterGroupName** – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster parameter group.

  Constraints:

  - Must be 1 to 255 letters or numbers.

- First character must be a letter
- Cannot end with a hyphen or contain two consecutive hyphens

> **Note**
>
> This value is stored as a lowercase string.

Errors

- DBParameterGroupNotFoundFault
- InvalidDBParameterGroupStateFault

ResetDBParameterGroup (action)

The AWS CLI name for this API is: `reset-db-parameter-group`.

Modifies the parameters of a DB parameter group to the engine/system default value. To reset specific parameters, provide a list of the following: `ParameterName` and `ApplyMethod`. To reset the entire DB parameter group, specify the `DBParameterGroup` name and `ResetAllParameters` parameters. When resetting the entire group, dynamic parameters are updated immediately and static parameters are set to `pending-reboot` to take effect on the next DB instance restart or `RebootDBInstance` request.

Request

- **DBParameterGroupName** (in the CLI: `--db-parameter-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB parameter group.

  Constraints:

  - Must match the name of an existing DBParameterGroup.
- **Parameters** (in the CLI: `--parameters`) – An array of Parameter objects.

  To reset the entire DB parameter group, specify the `DBParameterGroup` name and `ResetAllParameters` parameters. To reset specific parameters, provide a list of the following:

`ParameterName` and `ApplyMethod`. A maximum of 20 parameters can be modified in a single request.

Valid Values (for Apply method): `pending-reboot`

- **ResetAllParameters** (in the CLI: `--reset-all-parameters`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether (`true`) or not (`false`) to reset all parameters in the DB parameter group to default values.

  Default: `true`

Response

- **DBParameterGroupName** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB parameter group.

Errors

- InvalidDBParameterGroupStateFault
- DBParameterGroupNotFoundFault

ResetDBClusterParameterGroup (action)

The AWS CLI name for this API is: `reset-db-cluster-parameter-group`.

Modifies the parameters of a DB cluster parameter group to the default value. To reset specific parameters submit a list of the following: `ParameterName` and `ApplyMethod`. To reset the entire DB cluster parameter group, specify the `DBClusterParameterGroupName` and `ResetAllParameters` parameters.

When resetting the entire group, dynamic parameters are updated immediately and static parameters are set to `pending-reboot` to take effect on the next DB instance restart or the section called "RebootDBInstance" request. You must call the section called "RebootDBInstance" for every DB instance in your DB cluster that you want the updated static parameter to apply to.

Request

- **DBClusterParameterGroupName** (in the CLI: `--db-cluster-parameter-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster parameter group to reset.

- **Parameters** (in the CLI: `--parameters`) – An array of Parameter objects.

  A list of parameter names in the DB cluster parameter group to reset to the default values. You can't use this parameter if the `ResetAllParameters` parameter is set to `true`.

- **ResetAllParameters** (in the CLI: `--reset-all-parameters`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A value that is set to `true` to reset all parameters in the DB cluster parameter group to their default values, and `false` otherwise. You can't use this parameter if there is a list of parameter names specified for the `Parameters` parameter.

Response

- **DBClusterParameterGroupName** – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster parameter group.

  Constraints:

  - Must be 1 to 255 letters or numbers.
  - First character must be a letter
  - Cannot end with a hyphen or contain two consecutive hyphens

  > **Note**
  >
  > This value is stored as a lowercase string.

Errors

- InvalidDBParameterGroupStateFault
- DBParameterGroupNotFoundFault

DescribeDBParameters (action)

The AWS CLI name for this API is: `describe-db-parameters`.

Returns the detailed parameter list for a particular DB parameter group.

Request

- **DBParameterGroupName** (in the CLI: `--db-parameter-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of a specific DB parameter group to return details for.

  Constraints:

  - If supplied, must match the name of an existing DBParameterGroup.
- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  This parameter is not currently supported.
- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous `DescribeDBParameters` request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.
- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

  Default: 100

  Constraints: Minimum 20, maximum 100.
- **Source** (in the CLI: `--source`) – a String, of type: `string` (a UTF-8 encoded string).

  The parameter types to return.

  Default: All parameter types returned

  Valid Values: `user | system | engine-default`

Response

- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.
- **Parameters** – An array of Parameter objects.

  A list of the section called "Parameter" values.

Errors

- DBParameterGroupNotFoundFault

DescribeDBParameterGroups (action)

The AWS CLI name for this API is: `describe-db-parameter-groups`.

Returns a list of `DBParameterGroup` descriptions. If a `DBParameterGroupName` is specified, the list will contain only the description of the specified DB parameter group.

Request

- **DBParameterGroupName** (in the CLI: `--db-parameter-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of a specific DB parameter group to return details for.

  Constraints:

  - If supplied, must match the name of an existing DBClusterParameterGroup.
- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  This parameter is not currently supported.
- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous `DescribeDBParameterGroups` request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

  Default: 100

  Constraints: Minimum 20, maximum 100.

Response

- **DBParameterGroups** – An array of DBParameterGroup objects.

  A list of the section called "DBParameterGroup" instances.
- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

Errors

- DBParameterGroupNotFoundFault

DescribeDBClusterParameters (action)

The AWS CLI name for this API is: `describe-db-cluster-parameters`.

Returns the detailed parameter list for a particular DB cluster parameter group.

Request

- **DBClusterParameterGroupName** (in the CLI: `--db-cluster-parameter-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of a specific DB cluster parameter group to return parameter details for.

  Constraints:

  - If supplied, must match the name of an existing DBClusterParameterGroup.
- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  This parameter is not currently supported.
- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous `DescribeDBClusterParameters` request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.
- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

  Default: 100

  Constraints: Minimum 20, maximum 100.
- **Source** (in the CLI: `--source`) – a String, of type: `string` (a UTF-8 encoded string).

  A value that indicates to return only parameters for a specific source. Parameter sources can be `engine`, `service`, or `customer`.

Response

- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous DescribeDBClusterParameters request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords` .
- **Parameters** – An array of Parameter objects.

  Provides a list of parameters for the DB cluster parameter group.

Errors

- DBParameterGroupNotFoundFault

DescribeDBClusterParameterGroups (action)

The AWS CLI name for this API is: `describe-db-cluster-parameter-groups`.

Returns a list of `DBClusterParameterGroup` descriptions. If a `DBClusterParameterGroupName` parameter is specified, the list will contain only the description of the specified DB cluster parameter group.

Request

- **DBClusterParameterGroupName** (in the CLI: `--db-cluster-parameter-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of a specific DB cluster parameter group to return details for.

  Constraints:

  - If supplied, must match the name of an existing DBClusterParameterGroup.
- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  This parameter is not currently supported.
- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous `DescribeDBClusterParameterGroups` request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.
- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

  Default: 100

  Constraints: Minimum 20, maximum 100.

Response

- **DBClusterParameterGroups** – An array of DBClusterParameterGroup objects.

A list of DB cluster parameter groups.

- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous `DescribeDBClusterParameterGroups` request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

Errors

- DBParameterGroupNotFoundFault

DescribeEngineDefaultParameters (action)

The AWS CLI name for this API is: `describe-engine-default-parameters`.

Returns the default engine and system parameter information for the specified database engine.

Request

- **DBParameterGroupFamily** (in the CLI: `--db-parameter-group-family`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB parameter group family.
- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  Not currently supported.
- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous `DescribeEngineDefaultParameters` request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.
- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

Default: 100

Constraints: Minimum 20, maximum 100.

Response

Contains the result of a successful invocation of the the section called "DescribeEngineDefaultParameters" action.

- **DBParameterGroupFamily** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB parameter group family that the engine default parameters apply to.
- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous EngineDefaults request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords` .
- **Parameters** – An array of Parameter objects.

  Contains a list of engine default parameters.

DescribeEngineDefaultClusterParameters (action)

The AWS CLI name for this API is: `describe-engine-default-cluster-parameters`.

Returns the default engine and system parameter information for the cluster database engine.

Request

- **DBParameterGroupFamily** (in the CLI: `--db-parameter-group-family`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster parameter group family to return engine parameter information for.
- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  This parameter is not currently supported.
- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

An optional pagination token provided by a previous `DescribeEngineDefaultClusterParameters` request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

  Default: 100

  Constraints: Minimum 20, maximum 100.

Response

Contains the result of a successful invocation of the the section called "DescribeEngineDefaultParameters" action.

- **DBParameterGroupFamily** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB parameter group family that the engine default parameters apply to.

- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous EngineDefaults request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords` .

- **Parameters** – An array of Parameter objects.

  Contains a list of engine default parameters.

Structures:

Parameter (structure)

Specifies a parameter.

Fields

- **AllowedValues** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the valid range of values for the parameter.
- **ApplyMethod** – This is an ApplyMethod, of type: `string` (a UTF-8 encoded string).

  Indicates when to apply parameter updates.
- **ApplyType** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the engine specific parameters type.
- **DataType** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the valid data type for the parameter.
- **Description** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides a description of the parameter.
- **IsModifiable** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether (`true`) or not (`false`) the parameter can be modified. Some parameters have security or operational implications that prevent them from being changed.
- **MinimumEngineVersion** – This is a String, of type: `string` (a UTF-8 encoded string).

  The earliest engine version to which the parameter can apply.
- **ParameterName** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the parameter.
- **ParameterValue** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the value of the parameter.
- **Source** – This is a String, of type: `string` (a UTF-8 encoded string).

  Indicates the source of the parameter value.

DBParameterGroup (structure)

Contains the details of an Amazon Neptune DB parameter group.

This data type is used as a response element in the the section called "DescribeDBParameterGroups" action.

Fields

- **DBParameterGroupArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB parameter group.
- **DBParameterGroupFamily** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB parameter group family that this DB parameter group is compatible with.
- **DBParameterGroupName** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB parameter group.
- **Description** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the customer-specified description for this DB parameter group.

`DBParameterGroup` is used as the response element for:

- CopyDBParameterGroup
- CreateDBParameterGroup

DBClusterParameterGroup (structure)

Contains the details of an Amazon Neptune DB cluster parameter group.

This data type is used as a response element in the the section called "DescribeDBClusterParameterGroups" action.

Fields

- **DBClusterParameterGroupArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster parameter group.
- **DBClusterParameterGroupName** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB cluster parameter group.

- **DBParameterGroupFamily** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the DB parameter group family that this DB cluster parameter group is compatible with.
- **Description** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the customer-specified description for this DB cluster parameter group.

`DBClusterParameterGroup` is used as the response element for:

- CopyDBClusterParameterGroup
- CreateDBClusterParameterGroup

DBParameterGroupStatus (structure)

The status of the DB parameter group.

This data type is used as a response element in the following actions:

- the section called "CreateDBInstance"
- the section called "DeleteDBInstance"
- the section called "ModifyDBInstance"
- the section called "RebootDBInstance"

Fields

- **DBParameterGroupName** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the DP parameter group.
- **ParameterApplyStatus** – This is a String, of type: `string` (a UTF-8 encoded string).

  The status of parameter updates.

Neptune Subnet API

Actions:

- CreateDBSubnetGroup (action)
- DeleteDBSubnetGroup (action)
- ModifyDBSubnetGroup (action)
- DescribeDBSubnetGroups (action)

Structures:

- Subnet (structure)
- DBSubnetGroup (structure)

CreateDBSubnetGroup (action)

The AWS CLI name for this API is: `create-db-subnet-group`.

Creates a new DB subnet group. DB subnet groups must contain at least one subnet in at least two AZs in the Amazon Region.

Request

- **DBSubnetGroupDescription** (in the CLI: `--db-subnet-group-description`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The description for the DB subnet group.
- **DBSubnetGroupName** (in the CLI: `--db-subnet-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name for the DB subnet group. This value is stored as a lowercase string.

  Constraints: Must contain no more than 255 letters, numbers, periods, underscores, spaces, or hyphens. Must not be default.

  Example: `mySubnetgroup`
- **SubnetIds** (in the CLI: `--subnet-ids`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The EC2 Subnet IDs for the DB subnet group.
- **Tags** (in the CLI: `--tags`) – An array of Tag objects.

The tags to be assigned to the new DB subnet group.

Response

Contains the details of an Amazon Neptune DB subnet group.

This data type is used as a response element in the the section called "DescribeDBSubnetGroups" action.

- **DBSubnetGroupArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB subnet group.
- **DBSubnetGroupDescription** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the description of the DB subnet group.
- **DBSubnetGroupName** – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB subnet group.
- **SubnetGroupStatus** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the status of the DB subnet group.
- **Subnets** – An array of Subnet objects.

  Contains a list of the section called "Subnet" elements.
- **VpcId** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the VpcId of the DB subnet group.

Errors

- DBSubnetGroupAlreadyExistsFault
- DBSubnetGroupQuotaExceededFault
- DBSubnetQuotaExceededFault
- DBSubnetGroupDoesNotCoverEnoughAZs
- InvalidSubnet

DeleteDBSubnetGroup (action)

The AWS CLI name for this API is: `delete-db-subnet-group`.

Deletes a DB subnet group.

> **Note**
>
> The specified database subnet group must not be associated with any DB instances.

Request

- **DBSubnetGroupName** (in the CLI: `--db-subnet-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the database subnet group to delete.

  > **Note**
  >
  > You can't delete the default subnet group.

  Constraints:

  Constraints: Must match the name of an existing DBSubnetGroup. Must not be default.

  Example: `mySubnetgroup`

Response

- *No Response parameters.*

Errors

- InvalidDBSubnetGroupStateFault
- InvalidDBSubnetStateFault
- DBSubnetGroupNotFoundFault

ModifyDBSubnetGroup (action)

The AWS CLI name for this API is: `modify-db-subnet-group`.

Modifies an existing DB subnet group. DB subnet groups must contain at least one subnet in at least two AZs in the Amazon Region.

Request

- **DBSubnetGroupDescription** (in the CLI: `--db-subnet-group-description`) – a String, of type: `string` (a UTF-8 encoded string).

  The description for the DB subnet group.
- **DBSubnetGroupName** (in the CLI: `--db-subnet-group-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name for the DB subnet group. This value is stored as a lowercase string. You can't modify the default subnet group.

  Constraints: Must match the name of an existing DBSubnetGroup. Must not be default.

  Example: `mySubnetgroup`
- **SubnetIds** (in the CLI: `--subnet-ids`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The EC2 subnet IDs for the DB subnet group.

Response

Contains the details of an Amazon Neptune DB subnet group.

This data type is used as a response element in the the section called "DescribeDBSubnetGroups" action.

- **DBSubnetGroupArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB subnet group.
- **DBSubnetGroupDescription** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the description of the DB subnet group.

- **DBSubnetGroupName** – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB subnet group.
- **SubnetGroupStatus** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the status of the DB subnet group.
- **Subnets** – An array of Subnet objects.

  Contains a list of the section called "Subnet" elements.
- **VpcId** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the VpcId of the DB subnet group.

Errors

- DBSubnetGroupNotFoundFault
- DBSubnetQuotaExceededFault
- SubnetAlreadyInUse
- DBSubnetGroupDoesNotCoverEnoughAZs
- InvalidSubnet

DescribeDBSubnetGroups (action)

The AWS CLI name for this API is: `describe-db-subnet-groups`.

Returns a list of DBSubnetGroup descriptions. If a DBSubnetGroupName is specified, the list will contain only the descriptions of the specified DBSubnetGroup.

For an overview of CIDR ranges, go to the Wikipedia Tutorial.

Request

- **DBSubnetGroupName** (in the CLI: `--db-subnet-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB subnet group to return details for.
- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

This parameter is not currently supported.

- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous DescribeDBSubnetGroups request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

  Default: 100

  Constraints: Minimum 20, maximum 100.

Response

- **DBSubnetGroups** – An array of DBSubnetGroup objects.

  A list of the section called "DBSubnetGroup" instances.

- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

Errors

- DBSubnetGroupNotFoundFault

Structures:

Subnet (structure)

Specifies a subnet.

This data type is used as a response element in the the section called "DescribeDBSubnetGroups" action.

Fields

- **SubnetAvailabilityZone** – This is An AvailabilityZone object.

  Specifies the EC2 Availability Zone that the subnet is in.
- **SubnetIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the identifier of the subnet.
- **SubnetStatus** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the status of the subnet.

DBSubnetGroup (structure)

Contains the details of an Amazon Neptune DB subnet group.

This data type is used as a response element in the the section called "DescribeDBSubnetGroups" action.

Fields

- **DBSubnetGroupArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB subnet group.
- **DBSubnetGroupDescription** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the description of the DB subnet group.
- **DBSubnetGroupName** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB subnet group.
- **SubnetGroupStatus** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the status of the DB subnet group.
- **Subnets** – This is An array of Subnet objects.

  Contains a list of the section called "Subnet" elements.

- **VpcId** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the VpcId of the DB subnet group.

`DBSubnetGroup` is used as the response element for:

- CreateDBSubnetGroup
- ModifyDBSubnetGroup

Neptune Snapshots API

Actions:

- CreateDBClusterSnapshot (action)
- DeleteDBClusterSnapshot (action)
- CopyDBClusterSnapshot (action)
- ModifyDBClusterSnapshotAttribute (action)
- RestoreDBClusterFromSnapshot (action)
- RestoreDBClusterToPointInTime (action)
- DescribeDBClusterSnapshots (action)
- DescribeDBClusterSnapshotAttributes (action)

Structures:

- DBClusterSnapshot (structure)
- DBClusterSnapshotAttribute (structure)
- DBClusterSnapshotAttributesResult (structure)

CreateDBClusterSnapshot (action)

The AWS CLI name for this API is: `create-db-cluster-snapshot`.

Creates a snapshot of a DB cluster.

Request

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the DB cluster to create a snapshot for. This parameter is not case-sensitive.

  Constraints:

  - Must match the identifier of an existing DBCluster.

  Example: `my-cluster1`
- **DBClusterSnapshotIdentifier** (in the CLI: `--db-cluster-snapshot-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the DB cluster snapshot. This parameter is stored as a lowercase string.

  Constraints:

  - Must contain from 1 to 63 letters, numbers, or hyphens.
  - First character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.

  Example: `my-cluster1-snapshot1`
- **Tags** (in the CLI: `--tags`) – An array of Tag objects.

  The tags to be assigned to the DB cluster snapshot.

Response

Contains the details for an Amazon Neptune DB cluster snapshot

This data type is used as a response element in the the section called "DescribeDBClusterSnapshots" action.

- **AllocatedStorage** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the allocated storage size in gibibytes (GiB).
- **AvailabilityZones** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster snapshot can be restored in.

- **ClusterCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).
- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the DB cluster identifier of the DB cluster that this DB cluster snapshot was created from.
- **DBClusterSnapshotArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster snapshot.
- **DBClusterSnapshotIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the identifier for a DB cluster snapshot. Must match the identifier of an existing snapshot.

  After you restore a DB cluster using a `DBClusterSnapshotIdentifier`, you must specify the same `DBClusterSnapshotIdentifier` for any future updates to the DB cluster. When you specify this property for an update, the DB cluster is not restored from the snapshot again, and the data in the database is not changed.

  However, if you don't specify the `DBClusterSnapshotIdentifier`, an empty DB cluster is created, and the original DB cluster is deleted. If you specify a property that is different from the previous snapshot restore property, the DB cluster is restored from the snapshot specified by the `DBClusterSnapshotIdentifier`, and the original DB cluster is deleted.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the database engine.
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the version of the database engine for this DB cluster snapshot.
- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.
- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster snapshot.

- **LicenseModel** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the license model information for this DB cluster snapshot.

- **PercentProgress** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the percentage of the estimated data that has been transferred.

- **Port** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the DB cluster was listening on at the time of the snapshot.

- **SnapshotCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Provides the time when the snapshot was taken, in Universal Coordinated Time (UTC).

- **SnapshotType** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the type of the DB cluster snapshot.

- **SourceDBClusterSnapshotArn** – a String, of type: `string` (a UTF-8 encoded string).

  If the DB cluster snapshot was copied from a source DB cluster snapshot, the Amazon Resource Name (ARN) for the source DB cluster snapshot, otherwise, a null value.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the status of this DB cluster snapshot.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster snapshot is encrypted.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  The storage type associated with the DB cluster snapshot.

- **VpcId** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the VPC ID associated with the DB cluster snapshot.

Errors

- DBClusterSnapshotAlreadyExistsFault
- InvalidDBClusterStateFault
- DBClusterNotFoundFault
- SnapshotQuotaExceededFault
- InvalidDBClusterSnapshotStateFault

DeleteDBClusterSnapshot (action)

The AWS CLI name for this API is: `delete-db-cluster-snapshot`.

Deletes a DB cluster snapshot. If the snapshot is being copied, the copy operation is terminated.

> **Note**
>
> The DB cluster snapshot must be in the `available` state to be deleted.

Request

- **DBClusterSnapshotIdentifier** (in the CLI: `--db-cluster-snapshot-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the DB cluster snapshot to delete.

  Constraints: Must be the name of an existing DB cluster snapshot in the `available` state.

Response

Contains the details for an Amazon Neptune DB cluster snapshot

This data type is used as a response element in the the section called "DescribeDBClusterSnapshots" action.

- **AllocatedStorage** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the allocated storage size in gibibytes (GiB).

- **AvailabilityZones** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster snapshot can be restored in.
- **ClusterCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).
- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the DB cluster identifier of the DB cluster that this DB cluster snapshot was created from.
- **DBClusterSnapshotArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster snapshot.
- **DBClusterSnapshotIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the identifier for a DB cluster snapshot. Must match the identifier of an existing snapshot.

  After you restore a DB cluster using a `DBClusterSnapshotIdentifier`, you must specify the same `DBClusterSnapshotIdentifier` for any future updates to the DB cluster. When you specify this property for an update, the DB cluster is not restored from the snapshot again, and the data in the database is not changed.

  However, if you don't specify the `DBClusterSnapshotIdentifier`, an empty DB cluster is created, and the original DB cluster is deleted. If you specify a property that is different from the previous snapshot restore property, the DB cluster is restored from the snapshot specified by the `DBClusterSnapshotIdentifier`, and the original DB cluster is deleted.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the database engine.
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the version of the database engine for this DB cluster snapshot.
- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster snapshot.

- **LicenseModel** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the license model information for this DB cluster snapshot.

- **PercentProgress** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the percentage of the estimated data that has been transferred.

- **Port** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the DB cluster was listening on at the time of the snapshot.

- **SnapshotCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Provides the time when the snapshot was taken, in Universal Coordinated Time (UTC).

- **SnapshotType** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the type of the DB cluster snapshot.

- **SourceDBClusterSnapshotArn** – a String, of type: `string` (a UTF-8 encoded string).

  If the DB cluster snapshot was copied from a source DB cluster snapshot, the Amazon Resource Name (ARN) for the source DB cluster snapshot, otherwise, a null value.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the status of this DB cluster snapshot.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster snapshot is encrypted.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  The storage type associated with the DB cluster snapshot.

- **VpcId** – a String, of type: `string` (a UTF-8 encoded string).

Provides the VPC ID associated with the DB cluster snapshot.

Errors

- InvalidDBClusterSnapshotStateFault
- DBClusterSnapshotNotFoundFault

CopyDBClusterSnapshot (action)

The AWS CLI name for this API is: `copy-db-cluster-snapshot`.

Copies a snapshot of a DB cluster.

To copy a DB cluster snapshot from a shared manual DB cluster snapshot, `SourceDBClusterSnapshotIdentifier` must be the Amazon Resource Name (ARN) of the shared DB cluster snapshot.

Request

- **CopyTags** (in the CLI: `--copy-tags`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  True to copy all tags from the source DB cluster snapshot to the target DB cluster snapshot, and otherwise false. The default is false.
- **KmsKeyId** (in the CLI: `--kms-key-id`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Amazon KMS key ID for an encrypted DB cluster snapshot. The KMS key ID is the Amazon Resource Name (ARN), KMS key identifier, or the KMS key alias for the KMS encryption key.

  If you copy an encrypted DB cluster snapshot from your Amazon account, you can specify a value for `KmsKeyId` to encrypt the copy with a new KMS encryption key. If you don't specify a value for `KmsKeyId`, then the copy of the DB cluster snapshot is encrypted with the same KMS key as the source DB cluster snapshot.

  If you copy an encrypted DB cluster snapshot that is shared from another Amazon account, then you must specify a value for `KmsKeyId`.

KMS encryption keys are specific to the Amazon Region that they are created in, and you can't use encryption keys from one Amazon Region in another Amazon Region.

You cannot encrypt an unencrypted DB cluster snapshot when you copy it. If you try to copy an unencrypted DB cluster snapshot and specify a value for the KmsKeyId parameter, an error is returned.

- **PreSignedUrl** (in the CLI: `--pre-signed-url`) – a String, of type: `string` (a UTF-8 encoded string).

  Not currently supported.

- **SourceDBClusterSnapshotIdentifier** (in the CLI: `--source-db-cluster-snapshot-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the DB cluster snapshot to copy. This parameter is not case-sensitive.

  Constraints:

  - Must specify a valid system snapshot in the "available" state.
  - Specify a valid DB snapshot identifier.

  Example: `my-cluster-snapshot1`

- **Tags** (in the CLI: `--tags`) – An array of Tag objects.

  The tags to assign to the new DB cluster snapshot copy.

- **TargetDBClusterSnapshotIdentifier** (in the CLI: `--target-db-cluster-snapshot-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the new DB cluster snapshot to create from the source DB cluster snapshot. This parameter is not case-sensitive.

  Constraints:

  - Must contain from 1 to 63 letters, numbers, or hyphens.
  - First character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.

  Example: `my-cluster-snapshot2`

Response

Contains the details for an Amazon Neptune DB cluster snapshot

This data type is used as a response element in the the section called "DescribeDBClusterSnapshots" action.

- **AllocatedStorage** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the allocated storage size in gibibytes (GiB).
- **AvailabilityZones** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster snapshot can be restored in.
- **ClusterCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).
- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the DB cluster identifier of the DB cluster that this DB cluster snapshot was created from.
- **DBClusterSnapshotArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster snapshot.
- **DBClusterSnapshotIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the identifier for a DB cluster snapshot. Must match the identifier of an existing snapshot.

  After you restore a DB cluster using a `DBClusterSnapshotIdentifier`, you must specify the same `DBClusterSnapshotIdentifier` for any future updates to the DB cluster. When you specify this property for an update, the DB cluster is not restored from the snapshot again, and the data in the database is not changed.

  However, if you don't specify the `DBClusterSnapshotIdentifier`, an empty DB cluster is created, and the original DB cluster is deleted. If you specify a property that is different from the previous snapshot restore property, the DB cluster is restored from the snapshot specified by the `DBClusterSnapshotIdentifier`, and the original DB cluster is deleted.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

Specifies the name of the database engine.

- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the version of the database engine for this DB cluster snapshot.

- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster snapshot.

- **LicenseModel** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the license model information for this DB cluster snapshot.

- **PercentProgress** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the percentage of the estimated data that has been transferred.

- **Port** – an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the DB cluster was listening on at the time of the snapshot.

- **SnapshotCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Provides the time when the snapshot was taken, in Universal Coordinated Time (UTC).

- **SnapshotType** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the type of the DB cluster snapshot.

- **SourceDBClusterSnapshotArn** – a String, of type: `string` (a UTF-8 encoded string).

  If the DB cluster snapshot was copied from a source DB cluster snapshot, the Amazon Resource Name (ARN) for the source DB cluster snapshot, otherwise, a null value.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the status of this DB cluster snapshot.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

Specifies whether the DB cluster snapshot is encrypted.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  The storage type associated with the DB cluster snapshot.

- **VpcId** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the VPC ID associated with the DB cluster snapshot.

Errors

- DBClusterSnapshotAlreadyExistsFault
- DBClusterSnapshotNotFoundFault
- InvalidDBClusterStateFault
- InvalidDBClusterSnapshotStateFault
- SnapshotQuotaExceededFault
- KMSKeyNotAccessibleFault

ModifyDBClusterSnapshotAttribute (action)

The AWS CLI name for this API is: `modify-db-cluster-snapshot-attribute`.

Adds an attribute and values to, or removes an attribute and values from, a manual DB cluster snapshot.

To share a manual DB cluster snapshot with other Amazon accounts, specify `restore` as the `AttributeName` and use the `ValuesToAdd` parameter to add a list of IDs of the Amazon accounts that are authorized to restore the manual DB cluster snapshot. Use the value `all` to make the manual DB cluster snapshot public, which means that it can be copied or restored by all Amazon accounts. Do not add the `all` value for any manual DB cluster snapshots that contain private information that you don't want available to all Amazon accounts. If a manual DB cluster snapshot is encrypted, it can be shared, but only by specifying a list of authorized Amazon account IDs for the `ValuesToAdd` parameter. You can't use `all` as a value for that parameter in this case.

To view which Amazon accounts have access to copy or restore a manual DB cluster snapshot, or whether a manual DB cluster snapshot public or private, use the the section called "DescribeDBClusterSnapshotAttributes" API action.

Request

- **AttributeName** (in the CLI: `--attribute-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster snapshot attribute to modify.

  To manage authorization for other Amazon accounts to copy or restore a manual DB cluster snapshot, set this value to `restore`.
- **DBClusterSnapshotIdentifier** (in the CLI: `--db-cluster-snapshot-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier for the DB cluster snapshot to modify the attributes for.
- **ValuesToAdd** (in the CLI: `--values-to-add`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of DB cluster snapshot attributes to add to the attribute specified by `AttributeName`.

  To authorize other Amazon accounts to copy or restore a manual DB cluster snapshot, set this list to include one or more Amazon account IDs, or `all` to make the manual DB cluster snapshot restorable by any Amazon account. Do not add the `all` value for any manual DB cluster snapshots that contain private information that you don't want available to all Amazon accounts.
- **ValuesToRemove** (in the CLI: `--values-to-remove`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of DB cluster snapshot attributes to remove from the attribute specified by `AttributeName`.

  To remove authorization for other Amazon accounts to copy or restore a manual DB cluster snapshot, set this list to include one or more Amazon account identifiers, or `all` to remove authorization for any Amazon account to copy or restore the DB cluster snapshot. If you specify `all`, an Amazon account whose account ID is explicitly added to the `restore` attribute can still copy or restore a manual DB cluster snapshot.

Response

Contains the results of a successful call to the the section called "DescribeDBClusterSnapshotAttributes" API action.

Manual DB cluster snapshot attributes are used to authorize other Amazon accounts to copy or restore a manual DB cluster snapshot. For more information, see the the section called "ModifyDBClusterSnapshotAttribute" API action.

- **DBClusterSnapshotAttributes** – An array of DBClusterSnapshotAttribute objects.

  The list of attributes and values for the manual DB cluster snapshot.
- **DBClusterSnapshotIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the manual DB cluster snapshot that the attributes apply to.

Errors

- DBClusterSnapshotNotFoundFault
- InvalidDBClusterSnapshotStateFault
- SharedSnapshotQuotaExceededFault

RestoreDBClusterFromSnapshot (action)

The AWS CLI name for this API is: `restore-db-cluster-from-snapshot`.

Creates a new DB cluster from a DB snapshot or DB cluster snapshot.

If a DB snapshot is specified, the target DB cluster is created from the source DB snapshot with a default configuration and default security group.

If a DB cluster snapshot is specified, the target DB cluster is created from the source DB cluster restore point with the same configuration as the original source DB cluster, except that the new DB cluster is created with the default security group.

Request

- **AvailabilityZones** (in the CLI: `--availability-zones`) – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the restored DB cluster can be created in.
- **CopyTagsToSnapshot** (in the CLI: `--copy-tags-to-snapshot`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

If set to `true`, tags are copied to any snapshot of the restored DB cluster that is created.

- **DatabaseName** (in the CLI: `--database-name`) – a String, of type: `string` (a UTF-8 encoded string).

  Not supported.

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster to create from the DB snapshot or DB cluster snapshot. This parameter isn't case-sensitive.

  Constraints:

  - Must contain from 1 to 63 letters, numbers, or hyphens
  - First character must be a letter
  - Cannot end with a hyphen or contain two consecutive hyphens

  Example: `my-snapshot-id`

- **DBClusterParameterGroupName** (in the CLI: `--db-cluster-parameter-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster parameter group to associate with the new DB cluster.

  Constraints:

  - If supplied, must match the name of an existing DBClusterParameterGroup.

- **DBSubnetGroupName** (in the CLI: `--db-subnet-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB subnet group to use for the new DB cluster.

  Constraints: If supplied, must match the name of an existing DBSubnetGroup.

  Example: `mySubnetgroup`

- **DeletionProtection** (in the CLI: `--deletion-protection`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled. By default, deletion protection is disabled.

- **EnableCloudwatchLogsExports** (in the CLI: `--enable-cloudwatch-logs-exports`) – a String, of type: `string` (a UTF-8 encoded string).

  The list of logs that the restored DB cluster is to export to Amazon CloudWatch Logs.

- **EnableIAMDatabaseAuthentication** (in the CLI: `--enable-iam-database-authentication`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  True to enable mapping of Amazon Identity and Access Management (IAM) accounts to database accounts, and otherwise false.

  Default: `false`

- **Engine** (in the CLI: `--engine`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The database engine to use for the new DB cluster.

  Default: The same as source

  Constraint: Must be compatible with the engine of the source

- **EngineVersion** (in the CLI: `--engine-version`) – a String, of type: `string` (a UTF-8 encoded string).

  The version of the database engine to use for the new DB cluster.

- **KmsKeyId** (in the CLI: `--kms-key-id`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon KMS key identifier to use when restoring an encrypted DB cluster from a DB snapshot or DB cluster snapshot.

  The KMS key identifier is the Amazon Resource Name (ARN) for the KMS encryption key. If you are restoring a DB cluster with the same Amazon account that owns the KMS encryption key used to encrypt the new DB cluster, then you can use the KMS key alias instead of the ARN for the KMS encryption key.

  If you do not specify a value for the `KmsKeyId` parameter, then the following will occur:

  - If the DB snapshot or DB cluster snapshot in `SnapshotIdentifier` is encrypted, then the restored DB cluster is encrypted using the KMS key that was used to encrypt the DB snapshot or DB cluster snapshot.
  - If the DB snapshot or DB cluster snapshot in `SnapshotIdentifier` is not encrypted, then the restored DB cluster is not encrypted.

- **Port** (in the CLI: `--port`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The port number on which the new DB cluster accepts connections.

  Constraints: Value must be `1150-65535`

  Default: The same port as the original DB cluster.
- **ServerlessV2ScalingConfiguration** (in the CLI: `--serverless-v2-scaling-configuration`) – A ServerlessV2ScalingConfiguration object.

  Contains the scaling configuration of a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.
- **SnapshotIdentifier** (in the CLI: `--snapshot-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier for the DB snapshot or DB cluster snapshot to restore from.

  You can use either the name or the Amazon Resource Name (ARN) to specify a DB cluster snapshot. However, you can use only the ARN to specify a DB snapshot.

  Constraints:

  - Must match the identifier of an existing Snapshot.
- **StorageType** (in the CLI: `--storage-type`) – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the storage type to be associated with the DB cluster.

  Valid values: `standard`, `iopt1`

  Default: `standard`
- **Tags** (in the CLI: `--tags`) – An array of Tag objects.

  The tags to be assigned to the restored DB cluster.
- **VpcSecurityGroupIds** (in the CLI: `--vpc-security-group-ids`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of VPC security groups that the new DB cluster will belong to.

Response

Contains the details of an Amazon Neptune DB cluster.

This data type is used as a response element in the the section called "DescribeDBClusters".

- **AllocatedStorage** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  `AllocatedStorage` always returns 1, because Neptune DB cluster storage size is not fixed, but instead automatically adjusts as needed.
- **AssociatedRoles** – An array of DBClusterRole objects.

  Provides a list of the Amazon Identity and Access Management (IAM) roles that are associated with the DB cluster. IAM roles that are associated with a DB cluster grant permission for the DB cluster to access other Amazon services on your behalf.
- **AutomaticRestartTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Time at which the DB cluster will be automatically restarted.
- **AvailabilityZones** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster can be created in.
- **BacktrackConsumedChangeRecords** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.
- **BacktrackWindow** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.
- **BackupRetentionPeriod** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.
- **Capacity** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Not supported by Neptune.
- **CloneGroupId** – a String, of type: `string` (a UTF-8 encoded string).

  Identifies the clone group to which the DB cluster is associated.
- **ClusterCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).

- **CopyTagsToSnapshot** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, tags are copied to any snapshot of the DB cluster that is created.

- **CrossAccountClone** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, the DB cluster can be cloned across accounts.

- **DatabaseName** – a String, of type: `string` (a UTF-8 encoded string).

  Contains the name of the initial database of this DB cluster that was provided at create time, if one was specified when the DB cluster was created. This same name is returned for the life of the DB cluster.

- **DBClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster.

- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied DB cluster identifier. This identifier is the unique key that identifies a DB cluster.

- **DBClusterMembers** – An array of DBClusterMember objects.

  Provides the list of instances that make up the DB cluster.

- **DBClusterParameterGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB cluster parameter group for the DB cluster.

- **DbClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Region-unique, immutable identifier for the DB cluster. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB cluster is accessed.

- **DBSubnetGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies information on the subnet group associated with the DB cluster, including the name, description, and subnets in the subnet group.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled.

- **EarliestBacktrackTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Not supported by Neptune.
- **EarliestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the earliest time to which a database can be restored with point-in-time restore.
- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

  A list of the log types that this DB cluster is configured to export to CloudWatch Logs. Valid log types are: `audit` (to publish audit logs to CloudWatch) and slowquery (to publish slow-query logs to CloudWatch). See Publishing Neptune logs to Amazon CloudWatch logs.
- **Endpoint** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the connection endpoint for the primary instance of the DB cluster.
- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB cluster.
- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.
- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.
- **HostedZoneId** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the ID that Amazon Route 53 assigns when you create a hosted zone.
- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.
- **IOOptimizedNextAllowedModificationTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

The next time you can modify the DB cluster to use the `iopt1` storage type.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster.

- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.

- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster has instances in multiple Availability Zones.

- **PendingModifiedValues** – A ClusterPendingModifiedValues object.

  This data type is used as a response element in the `ModifyDBCluster` operation and contains changes that will be applied during the next maintenance window.

- **PercentProgress** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the progress of the operation as a percentage.

- **Port** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the database engine is listening on.

- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.

- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

- **ReaderEndpoint** – a String, of type: `string` (a UTF-8 encoded string).

  The reader endpoint for the DB cluster. The reader endpoint for a DB cluster load-balances connections across the Read Replicas that are available in a DB cluster. As clients request new connections to the reader endpoint, Neptune distributes the connection requests among the Read Replicas in the DB cluster. This functionality can help balance your read workload across multiple Read Replicas in your DB cluster.

If a failover occurs, and the Read Replica that you are connected to is promoted to be the primary instance, your connection is dropped. To continue sending your read workload to other Read Replicas in the cluster, you can then reconnect to the reader endpoint.

- **ReadReplicaIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB cluster.

- **ReplicationSourceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ReplicationType** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ServerlessV2ScalingConfiguration** – A ServerlessV2ScalingConfigurationInfo object.

  Shows the scaling configuration for a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this DB cluster.

- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster is encrypted.

- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  The storage type used by the DB cluster.

  Valid Values:

  - **`standard`** – (*the default*) Provides cost-effective database storage for applications with moderate to small I/O usage.
  - **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

    Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.

- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

Provides a list of VPC security groups that the DB cluster belongs to.

Errors

- DBClusterAlreadyExistsFault
- DBClusterQuotaExceededFault
- StorageQuotaExceededFault
- DBSubnetGroupNotFoundFault
- DBSnapshotNotFoundFault
- DBClusterSnapshotNotFoundFault
- InsufficientDBClusterCapacityFault
- InsufficientStorageClusterCapacityFault
- InvalidDBSnapshotStateFault
- InvalidDBClusterSnapshotStateFault
- StorageQuotaExceededFault
- InvalidVPCNetworkStateFault
- InvalidRestoreFault
- DBSubnetGroupNotFoundFault
- InvalidSubnet
- OptionGroupNotFoundFault
- KMSKeyNotAccessibleFault
- DBClusterParameterGroupNotFoundFault

RestoreDBClusterToPointInTime (action)

The AWS CLI name for this API is: `restore-db-cluster-to-point-in-time`.

Restores a DB cluster to an arbitrary point in time. Users can restore to any point in time before `LatestRestorableTime` for up to `BackupRetentionPeriod` days. The target DB cluster is created from the source DB cluster with the same configuration as the original DB cluster, except that the new DB cluster is created with the default DB security group.

> **Note**
>
> This action only restores the DB cluster, not the DB instances for that DB cluster. You must invoke the the section called "CreateDBInstance" action to create DB instances for the restored DB cluster, specifying the identifier of the restored DB cluster in `DBClusterIdentifier`. You can create DB instances only after the `RestoreDBClusterToPointInTime` action has completed and the DB cluster is available.

Request

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the new DB cluster to be created.

  Constraints:

  - Must contain from 1 to 63 letters, numbers, or hyphens
  - First character must be a letter
  - Cannot end with a hyphen or contain two consecutive hyphens
- **DBClusterParameterGroupName** (in the CLI: `--db-cluster-parameter-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB cluster parameter group to associate with the new DB cluster.

  Constraints:

  - If supplied, must match the name of an existing DBClusterParameterGroup.
- **DBSubnetGroupName** (in the CLI: `--db-subnet-group-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The DB subnet group name to use for the new DB cluster.

  Constraints: If supplied, must match the name of an existing DBSubnetGroup.

  Example: `mySubnetgroup`
- **DeletionProtection** (in the CLI: `--deletion-protection`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

A value that indicates whether the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled. By default, deletion protection is disabled.

- **EnableCloudwatchLogsExports** (in the CLI: `--enable-cloudwatch-logs-exports`) – a String, of type: `string` (a UTF-8 encoded string).

  The list of logs that the restored DB cluster is to export to CloudWatch Logs.

- **EnableIAMDatabaseAuthentication** (in the CLI: `--enable-iam-database-authentication`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  True to enable mapping of Amazon Identity and Access Management (IAM) accounts to database accounts, and otherwise false.

  Default: `false`

- **KmsKeyId** (in the CLI: `--kms-key-id`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon KMS key identifier to use when restoring an encrypted DB cluster from an encrypted DB cluster.

  The KMS key identifier is the Amazon Resource Name (ARN) for the KMS encryption key. If you are restoring a DB cluster with the same Amazon account that owns the KMS encryption key used to encrypt the new DB cluster, then you can use the KMS key alias instead of the ARN for the KMS encryption key.

  You can restore to a new DB cluster and encrypt the new DB cluster with a KMS key that is different than the KMS key used to encrypt the source DB cluster. The new DB cluster is encrypted with the KMS key identified by the `KmsKeyId` parameter.

  If you do not specify a value for the `KmsKeyId` parameter, then the following will occur:

  - If the DB cluster is encrypted, then the restored DB cluster is encrypted using the KMS key that was used to encrypt the source DB cluster.
  - If the DB cluster is not encrypted, then the restored DB cluster is not encrypted.

  If `DBClusterIdentifier` refers to a DB cluster that is not encrypted, then the restore request is rejected.

- **Port** (in the CLI: `--port`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The port number on which the new DB cluster accepts connections.

Constraints: Value must be `1150-65535`

Default: The same port as the original DB cluster.

- **RestoreToTime** (in the CLI: `--restore-to-time`) – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The date and time to restore the DB cluster to.

  Valid Values: Value must be a time in Universal Coordinated Time (UTC) format

  Constraints:

  - Must be before the latest restorable time for the DB instance
  - Must be specified if `UseLatestRestorableTime` parameter is not provided
  - Cannot be specified if `UseLatestRestorableTime` parameter is true
  - Cannot be specified if `RestoreType` parameter is `copy-on-write`

  Example: `2015-03-07T23:45:00Z`

- **RestoreType** (in the CLI: `--restore-type`) – a String, of type: `string` (a UTF-8 encoded string).

  The type of restore to be performed. You can specify one of the following values:

  - `full-copy` - The new DB cluster is restored as a full copy of the source DB cluster.
  - `copy-on-write` - The new DB cluster is restored as a clone of the source DB cluster.

  If you don't specify a `RestoreType` value, then the new DB cluster is restored as a full copy of the source DB cluster.

- **ServerlessV2ScalingConfiguration** (in the CLI: `--serverless-v2-scaling-configuration`) – A ServerlessV2ScalingConfiguration object.

  Contains the scaling configuration of a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

- **SourceDBClusterIdentifier** (in the CLI: `--source-db-cluster-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the source DB cluster from which to restore.

Constraints:

- Must match the identifier of an existing DBCluster.

- **StorageType** (in the CLI: `--storage-type`) – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the storage type to be associated with the DB cluster.

  Valid values: `standard`, `iopt1`

  Default: `standard`

- **Tags** (in the CLI: `--tags`) – An array of Tag objects.

  The tags to be applied to the restored DB cluster.

- **UseLatestRestorableTime** (in the CLI: `--use-latest-restorable-time`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A value that is set to `true` to restore the DB cluster to the latest restorable backup time, and `false` otherwise.

  Default: `false`

  Constraints: Cannot be specified if `RestoreToTime` parameter is provided.

- **VpcSecurityGroupIds** (in the CLI: `--vpc-security-group-ids`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of VPC security groups that the new DB cluster belongs to.

Response

Contains the details of an Amazon Neptune DB cluster.

This data type is used as a response element in the the section called "DescribeDBClusters".

- **AllocatedStorage** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  `AllocatedStorage` always returns 1, because Neptune DB cluster storage size is not fixed, but instead automatically adjusts as needed.

- **AssociatedRoles** – An array of DBClusterRole objects.

Provides a list of the Amazon Identity and Access Management (IAM) roles that are associated with the DB cluster. IAM roles that are associated with a DB cluster grant permission for the DB cluster to access other Amazon services on your behalf.

- **AutomaticRestartTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Time at which the DB cluster will be automatically restarted.

- **AvailabilityZones** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster can be created in.

- **BacktrackConsumedChangeRecords** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.

- **BacktrackWindow** – a LongOptional, of type: `long` (a signed 64-bit integer).

  Not supported by Neptune.

- **BackupRetentionPeriod** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the number of days for which automatic DB snapshots are retained.

- **Capacity** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Not supported by Neptune.

- **CloneGroupId** – a String, of type: `string` (a UTF-8 encoded string).

  Identifies the clone group to which the DB cluster is associated.

- **ClusterCreateTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).

- **CopyTagsToSnapshot** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, tags are copied to any snapshot of the DB cluster that is created.

- **CrossAccountClone** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, the DB cluster can be cloned across accounts.

- **DatabaseName** – a String, of type: `string` (a UTF-8 encoded string).

Contains the name of the initial database of this DB cluster that was provided at create time, if one was specified when the DB cluster was created. This same name is returned for the life of the DB cluster.

- **DBClusterArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster.

- **DBClusterIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Contains a user-supplied DB cluster identifier. This identifier is the unique key that identifies a DB cluster.

- **DBClusterMembers** – An array of DBClusterMember objects.

  Provides the list of instances that make up the DB cluster.

- **DBClusterParameterGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB cluster parameter group for the DB cluster.

- **DbClusterResourceId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Region-unique, immutable identifier for the DB cluster. This identifier is found in Amazon CloudTrail log entries whenever the Amazon KMS key for the DB cluster is accessed.

- **DBSubnetGroup** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies information on the subnet group associated with the DB cluster, including the name, description, and subnets in the subnet group.

- **DeletionProtection** – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not the DB cluster has deletion protection enabled. The database can't be deleted when deletion protection is enabled.

- **EarliestBacktrackTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Not supported by Neptune.

- **EarliestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the earliest time to which a database can be restored with point-in-time restore.

- **EnabledCloudwatchLogsExports** – a String, of type: `string` (a UTF-8 encoded string).

A list of the log types that this DB cluster is configured to export to CloudWatch Logs. Valid log types are: `audit` (to publish audit logs to CloudWatch) and slowquery (to publish slow-query logs to CloudWatch). See Publishing Neptune logs to Amazon CloudWatch logs.

- **Endpoint** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the connection endpoint for the primary instance of the DB cluster.

- **Engine** – a String, of type: `string` (a UTF-8 encoded string).

  Provides the name of the database engine to be used for this DB cluster.

- **EngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Indicates the database engine version.

- **GlobalClusterIdentifier** – a GlobalClusterIdentifier, of type: `string` (a UTF-8 encoded string), not less than 1 or more than 255 ?st?s, matching this regular expression: `[A-Za-z][0-9A-Za-z-:._]*`.

  Contains a user-supplied global database cluster identifier. This identifier is the unique key that identifies a global database.

- **HostedZoneId** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the ID that Amazon Route 53 assigns when you create a hosted zone.

- **IAMDatabaseAuthenticationEnabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.

- **IOOptimizedNextAllowedModificationTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The next time you can modify the DB cluster to use the `iopt1` storage type.

- **KmsKeyId** – a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster.

- **LatestRestorableTime** – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the latest time to which a database can be restored with point-in-time restore.

- **MultiAZ** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster has instances in multiple Availability Zones.
- **PendingModifiedValues** – A ClusterPendingModifiedValues object.

  This data type is used as a response element in the `ModifyDBCluster` operation and contains changes that will be applied during the next maintenance window.
- **PercentProgress** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the progress of the operation as a percentage.
- **Port** – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the database engine is listening on.
- **PreferredBackupWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by the `BackupRetentionPeriod`.
- **PreferredMaintenanceWindow** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).
- **ReaderEndpoint** – a String, of type: `string` (a UTF-8 encoded string).

  The reader endpoint for the DB cluster. The reader endpoint for a DB cluster load-balances connections across the Read Replicas that are available in a DB cluster. As clients request new connections to the reader endpoint, Neptune distributes the connection requests among the Read Replicas in the DB cluster. This functionality can help balance your read workload across multiple Read Replicas in your DB cluster.

  If a failover occurs, and the Read Replica that you are connected to is promoted to be the primary instance, your connection is dropped. To continue sending your read workload to other Read Replicas in the cluster, you can then reconnect to the reader endpoint.
- **ReadReplicaIdentifiers** – a String, of type: `string` (a UTF-8 encoded string).

  Contains one or more identifiers of the Read Replicas associated with this DB cluster.
- **ReplicationSourceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.

- **ReplicationType** – a String, of type: `string` (a UTF-8 encoded string).

  Not supported by Neptune.
- **ServerlessV2ScalingConfiguration** – A ServerlessV2ScalingConfigurationInfo object.

  Shows the scaling configuration for a Neptune Serverless DB cluster.

  For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.
- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  Specifies the current state of this DB cluster.
- **StorageEncrypted** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster is encrypted.
- **StorageType** – a String, of type: `string` (a UTF-8 encoded string).

  The storage type used by the DB cluster.

  Valid Values:

  - **`standard`** – (*the default*) Provides cost-effective database storage for applications with moderate to small I/O usage.
  - **`iopt1`** – Enables I/O-Optimized storage that's designed to meet the needs of I/O-intensive graph workloads that require predictable pricing with low I/O latency and consistent I/O throughput.

    Neptune I/O-Optimized storage is only available starting with engine release 1.3.0.0.
- **VpcSecurityGroups** – An array of VpcSecurityGroupMembership objects.

  Provides a list of VPC security groups that the DB cluster belongs to.

Errors

- DBClusterAlreadyExistsFault
- DBClusterNotFoundFault
- DBClusterQuotaExceededFault
- DBClusterSnapshotNotFoundFault
- DBSubnetGroupNotFoundFault

- InsufficientDBClusterCapacityFault
- InsufficientStorageClusterCapacityFault
- InvalidDBClusterSnapshotStateFault
- InvalidDBClusterStateFault
- InvalidDBSnapshotStateFault
- InvalidRestoreFault
- InvalidSubnet
- InvalidVPCNetworkStateFault
- KMSKeyNotAccessibleFault
- OptionGroupNotFoundFault
- StorageQuotaExceededFault
- DBClusterParameterGroupNotFoundFault

DescribeDBClusterSnapshots (action)

The AWS CLI name for this API is: `describe-db-cluster-snapshots`.

Returns information about DB cluster snapshots. This API action supports pagination.

Request

- **DBClusterIdentifier** (in the CLI: `--db-cluster-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  The ID of the DB cluster to retrieve the list of DB cluster snapshots for. This parameter can't be used in conjunction with the `DBClusterSnapshotIdentifier` parameter. This parameter is not case-sensitive.

  Constraints:

  - If supplied, must match the identifier of an existing DBCluster.
- **DBClusterSnapshotIdentifier** (in the CLI: `--db-cluster-snapshot-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  A specific DB cluster snapshot identifier to describe. This parameter can't be used in conjunction with the `DBClusterIdentifier` parameter. This value is stored as a lowercase string.

Constraints:

- If supplied, must match the identifier of an existing DBClusterSnapshot.
- If this identifier is for an automated snapshot, the `SnapshotType` parameter must also be specified.

- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  This parameter is not currently supported.

- **IncludePublic** (in the CLI: `--include-public`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True to include manual DB cluster snapshots that are public and can be copied or restored by any Amazon account, and otherwise false. The default is `false`. The default is false.

  You can share a manual DB cluster snapshot as public by using the the section called "ModifyDBClusterSnapshotAttribute" API action.

- **IncludeShared** (in the CLI: `--include-shared`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True to include shared manual DB cluster snapshots from other Amazon accounts that this Amazon account has been given permission to copy or restore, and otherwise false. The default is `false`.

  You can give an Amazon account permission to restore a manual DB cluster snapshot from another Amazon account by the the section called "ModifyDBClusterSnapshotAttribute" API action.

- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous `DescribeDBClusterSnapshots` request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

Default: 100

Constraints: Minimum 20, maximum 100.

- **SnapshotType** (in the CLI: `--snapshot-type`) – a String, of type: `string` (a UTF-8 encoded string).

  The type of DB cluster snapshots to be returned. You can specify one of the following values:

  - `automated` - Return all DB cluster snapshots that have been automatically taken by Amazon Neptune for my Amazon account.
  - `manual` - Return all DB cluster snapshots that have been taken by my Amazon account.
  - `shared` - Return all manual DB cluster snapshots that have been shared to my Amazon account.
  - `public` - Return all DB cluster snapshots that have been marked as public.

  If you don't specify a `SnapshotType` value, then both automated and manual DB cluster snapshots are returned. You can include shared DB cluster snapshots with these results by setting the `IncludeShared` parameter to `true`. You can include public DB cluster snapshots with these results by setting the `IncludePublic` parameter to `true`.

  The `IncludeShared` and `IncludePublic` parameters don't apply for `SnapshotType` values of `manual` or `automated`. The `IncludePublic` parameter doesn't apply when `SnapshotType` is set to `shared`. The `IncludeShared` parameter doesn't apply when `SnapshotType` is set to `public`.

Response

- **DBClusterSnapshots** – An array of DBClusterSnapshot objects.

  Provides a list of DB cluster snapshots for the user.

- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous the section called "DescribeDBClusterSnapshots" request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

Errors

- DBClusterSnapshotNotFoundFault

DescribeDBClusterSnapshotAttributes (action)

The AWS CLI name for this API is: `describe-db-cluster-snapshot-attributes`.

Returns a list of DB cluster snapshot attribute names and values for a manual DB cluster snapshot.

When sharing snapshots with other Amazon accounts, `DescribeDBClusterSnapshotAttributes` returns the `restore` attribute and a list of IDs for the Amazon accounts that are authorized to copy or restore the manual DB cluster snapshot. If `all` is included in the list of values for the `restore` attribute, then the manual DB cluster snapshot is public and can be copied or restored by all Amazon accounts.

To add or remove access for an Amazon account to copy or restore a manual DB cluster snapshot, or to make the manual DB cluster snapshot public or private, use the the section called "ModifyDBClusterSnapshotAttribute" API action.

Request

- **DBClusterSnapshotIdentifier** (in the CLI: `--db-cluster-snapshot-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier for the DB cluster snapshot to describe the attributes for.

Response

Contains the results of a successful call to the the section called "DescribeDBClusterSnapshotAttributes" API action.

Manual DB cluster snapshot attributes are used to authorize other Amazon accounts to copy or restore a manual DB cluster snapshot. For more information, see the the section called "ModifyDBClusterSnapshotAttribute" API action.

- **DBClusterSnapshotAttributes** – An array of DBClusterSnapshotAttribute objects.

  The list of attributes and values for the manual DB cluster snapshot.

- **DBClusterSnapshotIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the manual DB cluster snapshot that the attributes apply to.

Errors

- DBClusterSnapshotNotFoundFault

Structures:

DBClusterSnapshot (structure)

Contains the details for an Amazon Neptune DB cluster snapshot

This data type is used as a response element in the the section called "DescribeDBClusterSnapshots" action.

Fields

- **AllocatedStorage** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the allocated storage size in gibibytes (GiB).
- **AvailabilityZones** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the list of EC2 Availability Zones that instances in the DB cluster snapshot can be restored in.
- **ClusterCreateTime** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Specifies the time when the DB cluster was created, in Universal Coordinated Time (UTC).
- **DBClusterIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the DB cluster identifier of the DB cluster that this DB cluster snapshot was created from.
- **DBClusterSnapshotArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the DB cluster snapshot.
- **DBClusterSnapshotIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

Specifies the identifier for a DB cluster snapshot. Must match the identifier of an existing snapshot.

After you restore a DB cluster using a `DBClusterSnapshotIdentifier`, you must specify the same `DBClusterSnapshotIdentifier` for any future updates to the DB cluster. When you specify this property for an update, the DB cluster is not restored from the snapshot again, and the data in the database is not changed.

However, if you don't specify the `DBClusterSnapshotIdentifier`, an empty DB cluster is created, and the original DB cluster is deleted. If you specify a property that is different from the previous snapshot restore property, the DB cluster is restored from the snapshot specified by the `DBClusterSnapshotIdentifier`, and the original DB cluster is deleted.

- **Engine** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the database engine.

- **EngineVersion** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the version of the database engine for this DB cluster snapshot.

- **IAMDatabaseAuthenticationEnabled** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  True if mapping of Amazon Identity and Access Management (IAM) accounts to database accounts is enabled, and otherwise false.

- **KmsKeyId** – This is a String, of type: `string` (a UTF-8 encoded string).

  If `StorageEncrypted` is true, the Amazon KMS key identifier for the encrypted DB cluster snapshot.

- **LicenseModel** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the license model information for this DB cluster snapshot.

- **PercentProgress** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the percentage of the estimated data that has been transferred.

- **Port** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the DB cluster was listening on at the time of the snapshot.

- **SnapshotCreateTime** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  Provides the time when the snapshot was taken, in Universal Coordinated Time (UTC).
- **SnapshotType** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the type of the DB cluster snapshot.
- **SourceDBClusterSnapshotArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  If the DB cluster snapshot was copied from a source DB cluster snapshot, the Amazon Resource Name (ARN) for the source DB cluster snapshot, otherwise, a null value.
- **Status** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the status of this DB cluster snapshot.
- **StorageEncrypted** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Specifies whether the DB cluster snapshot is encrypted.
- **StorageType** – This is a String, of type: `string` (a UTF-8 encoded string).

  The storage type associated with the DB cluster snapshot.
- **VpcId** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the VPC ID associated with the DB cluster snapshot.

`DBClusterSnapshot` is used as the response element for:

- CreateDBClusterSnapshot
- CopyDBClusterSnapshot
- DeleteDBClusterSnapshot

DBClusterSnapshotAttribute (structure)

Contains the name and values of a manual DB cluster snapshot attribute.

Manual DB cluster snapshot attributes are used to authorize other Amazon accounts to restore a manual DB cluster snapshot. For more information, see the the section called "ModifyDBClusterSnapshotAttribute" API action.

Fields

- **AttributeName** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the manual DB cluster snapshot attribute.

  The attribute named `restore` refers to the list of Amazon accounts that have permission to copy or restore the manual DB cluster snapshot. For more information, see the the section called "ModifyDBClusterSnapshotAttribute" API action.
- **AttributeValues** – This is a String, of type: `string` (a UTF-8 encoded string).

  The value(s) for the manual DB cluster snapshot attribute.

  If the `AttributeName` field is set to `restore`, then this element returns a list of IDs of the Amazon accounts that are authorized to copy or restore the manual DB cluster snapshot. If a value of `all` is in the list, then the manual DB cluster snapshot is public and available for any Amazon account to copy or restore.

DBClusterSnapshotAttributesResult (structure)

Contains the results of a successful call to the the section called "DescribeDBClusterSnapshotAttributes" API action.

Manual DB cluster snapshot attributes are used to authorize other Amazon accounts to copy or restore a manual DB cluster snapshot. For more information, see the the section called "ModifyDBClusterSnapshotAttribute" API action.

Fields

- **DBClusterSnapshotAttributes** – This is An array of DBClusterSnapshotAttribute objects.

  The list of attributes and values for the manual DB cluster snapshot.
- **DBClusterSnapshotIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the manual DB cluster snapshot that the attributes apply to.

`DBClusterSnapshotAttributesResult` is used as the response element for:

- DescribeDBClusterSnapshotAttributes

- ModifyDBClusterSnapshotAttribute

Neptune Events API

Actions:

- CreateEventSubscription (action)
- DeleteEventSubscription (action)
- ModifyEventSubscription (action)
- DescribeEventSubscriptions (action)
- AddSourceIdentifierToSubscription (action)
- RemoveSourceIdentifierFromSubscription (action)
- DescribeEvents (action)
- DescribeEventCategories (action)

Structures:

- Event (structure)
- EventCategoriesMap (structure)
- EventSubscription (structure)

CreateEventSubscription (action)

The AWS CLI name for this API is: `create-event-subscription`.

Creates an event notification subscription. This action requires a topic ARN (Amazon Resource Name) created by either the Neptune console, the SNS console, or the SNS API. To obtain an ARN with SNS, you must create a topic in Amazon SNS and subscribe to the topic. The ARN is displayed in the SNS console.

You can specify the type of source (SourceType) you want to be notified of, provide a list of Neptune sources (SourceIds) that triggers the events, and provide a list of event categories (EventCategories) for events you want to be notified of. For example, you can specify SourceType = db-instance, SourceIds = mydbinstance1, mydbinstance2 and EventCategories = Availability, Backup.

If you specify both the SourceType and SourceIds, such as SourceType = db-instance and SourceIdentifier = myDBInstance1, you are notified of all the db-instance events for the specified source. If you specify a SourceType but do not specify a SourceIdentifier, you receive notice of the events for that source type for all your Neptune sources. If you do not specify either the SourceType nor the SourceIdentifier, you are notified of events generated from all Neptune sources belonging to your customer account.

Request

- **Enabled** (in the CLI: `--enabled`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  A Boolean value; set to **true** to activate the subscription, set to **false** to create the subscription but not active it.
- **EventCategories** (in the CLI: `--event-categories`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of event categories for a SourceType that you want to subscribe to. You can see a list of the categories for a given SourceType by using the **DescribeEventCategories** action.
- **SnsTopicArn** (in the CLI: `--sns-topic-arn`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of the SNS topic created for event notification. The ARN is created by Amazon SNS when you create a topic and subscribe to it.
- **SourceIds** (in the CLI: `--source-ids`) – a String, of type: `string` (a UTF-8 encoded string).

  The list of identifiers of the event sources for which events are returned. If not specified, then all sources are included in the response. An identifier must begin with a letter and must contain only ASCII letters, digits, and hyphens; it can't end with a hyphen or contain two consecutive hyphens.

  Constraints:

  - If SourceIds are supplied, SourceType must also be provided.
  - If the source type is a DB instance, then a `DBInstanceIdentifier` must be supplied.
  - If the source type is a DB security group, a `DBSecurityGroupName` must be supplied.
  - If the source type is a DB parameter group, a `DBParameterGroupName` must be supplied.
  - If the source type is a DB snapshot, a `DBSnapshotIdentifier` must be supplied.
- **SourceType** (in the CLI: `--source-type`) – a String, of type: `string` (a UTF-8 encoded string).

The type of source that is generating the events. For example, if you want to be notified of events generated by a DB instance, you would set this parameter to db-instance. if this value is not specified, all events are returned.

Valid values: `db-instance` | `db-cluster` | `db-parameter-group` | `db-security-group` | `db-snapshot` | `db-cluster-snapshot`

- **SubscriptionName** (in the CLI: `--subscription-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the subscription.

  Constraints: The name must be less than 255 characters.
- **Tags** (in the CLI: `--tags`) – An array of Tag objects.

  The tags to be applied to the new event subscription.

Response

Contains the results of a successful invocation of the the section called "DescribeEventSubscriptions" action.

- **CustomerAwsId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon customer account associated with the event notification subscription.
- **CustSubscriptionId** – a String, of type: `string` (a UTF-8 encoded string).

  The event notification subscription Id.
- **Enabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A Boolean value indicating if the subscription is enabled. True indicates the subscription is enabled.
- **EventCategoriesList** – a String, of type: `string` (a UTF-8 encoded string).

  A list of event categories for the event notification subscription.
- **EventSubscriptionArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the event subscription.
- **SnsTopicArn** – a String, of type: `string` (a UTF-8 encoded string).

The topic ARN of the event notification subscription.

- **SourceIdsList** – a String, of type: `string` (a UTF-8 encoded string).

  A list of source IDs for the event notification subscription.

- **SourceType** – a String, of type: `string` (a UTF-8 encoded string).

  The source type for the event notification subscription.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  The status of the event notification subscription.

  Constraints:

  Can be one of the following: creating | modifying | deleting | active | no-permission | topic-not-exist

  The status "no-permission" indicates that Neptune no longer has permission to post to the SNS topic. The status "topic-not-exist" indicates that the topic was deleted after the subscription was created.

- **SubscriptionCreationTime** – a String, of type: `string` (a UTF-8 encoded string).

  The time the event notification subscription was created.

Errors

- EventSubscriptionQuotaExceededFault
- SubscriptionAlreadyExistFault
- SNSInvalidTopicFault
- SNSNoAuthorizationFault
- SNSTopicArnNotFoundFault
- SubscriptionCategoryNotFoundFault
- SourceNotFoundFault

DeleteEventSubscription (action)

The AWS CLI name for this API is: `delete-event-subscription`.

Deletes an event notification subscription.

Request

- **SubscriptionName** (in the CLI: `--subscription-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the event notification subscription you want to delete.

Response

Contains the results of a successful invocation of the the section called "DescribeEventSubscriptions" action.

- **CustomerAwsId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon customer account associated with the event notification subscription.
- **CustSubscriptionId** – a String, of type: `string` (a UTF-8 encoded string).

  The event notification subscription Id.
- **Enabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A Boolean value indicating if the subscription is enabled. True indicates the subscription is enabled.
- **EventCategoriesList** – a String, of type: `string` (a UTF-8 encoded string).

  A list of event categories for the event notification subscription.
- **EventSubscriptionArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the event subscription.
- **SnsTopicArn** – a String, of type: `string` (a UTF-8 encoded string).

  The topic ARN of the event notification subscription.
- **SourceIdsList** – a String, of type: `string` (a UTF-8 encoded string).

  A list of source IDs for the event notification subscription.
- **SourceType** – a String, of type: `string` (a UTF-8 encoded string).

  The source type for the event notification subscription.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  The status of the event notification subscription.

  Constraints:

  Can be one of the following: creating | modifying | deleting | active | no-permission | topic-not-exist

  The status "no-permission" indicates that Neptune no longer has permission to post to the SNS topic. The status "topic-not-exist" indicates that the topic was deleted after the subscription was created.
- **SubscriptionCreationTime** – a String, of type: `string` (a UTF-8 encoded string).

  The time the event notification subscription was created.

Errors

- SubscriptionNotFoundFault
- InvalidEventSubscriptionStateFault

ModifyEventSubscription (action)

> The AWS CLI name for this API is: `modify-event-subscription`.

Modifies an existing event notification subscription. Note that you can't modify the source identifiers using this call; to change source identifiers for a subscription, use the the section called “AddSourceIdentifierToSubscription” and the section called “RemoveSourceIdentifierFromSubscription” calls.

You can see a list of the event categories for a given SourceType by using the **DescribeEventCategories** action.

Request

- **Enabled** (in the CLI: `--enabled`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  A Boolean value; set to **true** to activate the subscription.

- **EventCategories** (in the CLI: `--event-categories`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of event categories for a SourceType that you want to subscribe to. You can see a list of the categories for a given SourceType by using the **DescribeEventCategories** action.
- **SnsTopicArn** (in the CLI: `--sns-topic-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of the SNS topic created for event notification. The ARN is created by Amazon SNS when you create a topic and subscribe to it.
- **SourceType** (in the CLI: `--source-type`) – a String, of type: `string` (a UTF-8 encoded string).

  The type of source that is generating the events. For example, if you want to be notified of events generated by a DB instance, you would set this parameter to db-instance. if this value is not specified, all events are returned.

  Valid values: db-instance | db-parameter-group | db-security-group | db-snapshot
- **SubscriptionName** (in the CLI: `--subscription-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the event notification subscription.

Response

Contains the results of a successful invocation of the the section called "DescribeEventSubscriptions" action.

- **CustomerAwsId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon customer account associated with the event notification subscription.
- **CustSubscriptionId** – a String, of type: `string` (a UTF-8 encoded string).

  The event notification subscription Id.
- **Enabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A Boolean value indicating if the subscription is enabled. True indicates the subscription is enabled.
- **EventCategoriesList** – a String, of type: `string` (a UTF-8 encoded string).

A list of event categories for the event notification subscription.

- **EventSubscriptionArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the event subscription.

- **SnsTopicArn** – a String, of type: `string` (a UTF-8 encoded string).

  The topic ARN of the event notification subscription.

- **SourceIdsList** – a String, of type: `string` (a UTF-8 encoded string).

  A list of source IDs for the event notification subscription.

- **SourceType** – a String, of type: `string` (a UTF-8 encoded string).

  The source type for the event notification subscription.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  The status of the event notification subscription.

  Constraints:

  Can be one of the following: creating | modifying | deleting | active | no-permission | topic-not-exist

  The status "no-permission" indicates that Neptune no longer has permission to post to the SNS topic. The status "topic-not-exist" indicates that the topic was deleted after the subscription was created.

- **SubscriptionCreationTime** – a String, of type: `string` (a UTF-8 encoded string).

  The time the event notification subscription was created.

Errors

- EventSubscriptionQuotaExceededFault
- SubscriptionNotFoundFault
- SNSInvalidTopicFault
- SNSNoAuthorizationFault
- SNSTopicArnNotFoundFault
- SubscriptionCategoryNotFoundFault

DescribeEventSubscriptions (action)

The AWS CLI name for this API is: `describe-event-subscriptions`.

Lists all the subscription descriptions for a customer account. The description for a subscription includes SubscriptionName, SNSTopicARN, CustomerID, SourceType, SourceID, CreationTime, and Status.

If you specify a SubscriptionName, lists the description for that subscription.

Request

- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  This parameter is not currently supported.
- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous DescribeOrderableDBInstanceOptions request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords` .
- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

  Default: 100

  Constraints: Minimum 20, maximum 100.
- **SubscriptionName** (in the CLI: `--subscription-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of the event notification subscription you want to describe.

Response

- **EventSubscriptionsList** – An array of EventSubscription objects.

  A list of EventSubscriptions data types.

- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous DescribeOrderableDBInstanceOptions request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

Errors

- SubscriptionNotFoundFault

AddSourceIdentifierToSubscription (action)

The AWS CLI name for this API is: `add-source-identifier-to-subscription`.

Adds a source identifier to an existing event notification subscription.

Request

- **SourceIdentifier** (in the CLI: `--source-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the event source to be added.

  Constraints:

  - If the source type is a DB instance, then a `DBInstanceIdentifier` must be supplied.
  - If the source type is a DB security group, a `DBSecurityGroupName` must be supplied.
  - If the source type is a DB parameter group, a `DBParameterGroupName` must be supplied.
  - If the source type is a DB snapshot, a `DBSnapshotIdentifier` must be supplied.
- **SubscriptionName** (in the CLI: `--subscription-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the event notification subscription you want to add a source identifier to.

Response

Contains the results of a successful invocation of the the section called "DescribeEventSubscriptions" action.

- **CustomerAwsId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon customer account associated with the event notification subscription.
- **CustSubscriptionId** – a String, of type: `string` (a UTF-8 encoded string).

  The event notification subscription Id.
- **Enabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A Boolean value indicating if the subscription is enabled. True indicates the subscription is enabled.
- **EventCategoriesList** – a String, of type: `string` (a UTF-8 encoded string).

  A list of event categories for the event notification subscription.
- **EventSubscriptionArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the event subscription.
- **SnsTopicArn** – a String, of type: `string` (a UTF-8 encoded string).

  The topic ARN of the event notification subscription.
- **SourceIdsList** – a String, of type: `string` (a UTF-8 encoded string).

  A list of source IDs for the event notification subscription.
- **SourceType** – a String, of type: `string` (a UTF-8 encoded string).

  The source type for the event notification subscription.
- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  The status of the event notification subscription.

  Constraints:

  Can be one of the following: creating | modifying | deleting | active | no-permission | topic-not-exist

  The status "no-permission" indicates that Neptune no longer has permission to post to the SNS topic. The status "topic-not-exist" indicates that the topic was deleted after the subscription was created.
- **SubscriptionCreationTime** – a String, of type: `string` (a UTF-8 encoded string).

The time the event notification subscription was created.

Errors

- SubscriptionNotFoundFault
- SourceNotFoundFault

RemoveSourceIdentifierFromSubscription (action)

The AWS CLI name for this API is: `remove-source-identifier-from-subscription`.

Removes a source identifier from an existing event notification subscription.

Request

- **SourceIdentifier** (in the CLI: `--source-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The source identifier to be removed from the subscription, such as the **DB instance identifier** for a DB instance or the name of a security group.
- **SubscriptionName** (in the CLI: `--subscription-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The name of the event notification subscription you want to remove a source identifier from.

Response

Contains the results of a successful invocation of the the section called "DescribeEventSubscriptions" action.

- **CustomerAwsId** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon customer account associated with the event notification subscription.
- **CustSubscriptionId** – a String, of type: `string` (a UTF-8 encoded string).

  The event notification subscription Id.
- **Enabled** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

A Boolean value indicating if the subscription is enabled. True indicates the subscription is enabled.

- **EventCategoriesList** – a String, of type: `string` (a UTF-8 encoded string).

  A list of event categories for the event notification subscription.

- **EventSubscriptionArn** – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the event subscription.

- **SnsTopicArn** – a String, of type: `string` (a UTF-8 encoded string).

  The topic ARN of the event notification subscription.

- **SourceIdsList** – a String, of type: `string` (a UTF-8 encoded string).

  A list of source IDs for the event notification subscription.

- **SourceType** – a String, of type: `string` (a UTF-8 encoded string).

  The source type for the event notification subscription.

- **Status** – a String, of type: `string` (a UTF-8 encoded string).

  The status of the event notification subscription.

  Constraints:

  Can be one of the following: creating | modifying | deleting | active | no-permission | topic-not-exist

  The status "no-permission" indicates that Neptune no longer has permission to post to the SNS topic. The status "topic-not-exist" indicates that the topic was deleted after the subscription was created.

- **SubscriptionCreationTime** – a String, of type: `string` (a UTF-8 encoded string).

  The time the event notification subscription was created.

Errors

- SubscriptionNotFoundFault
- SourceNotFoundFault

DescribeEvents (action)

The AWS CLI name for this API is: `describe-events`.

Returns events related to DB instances, DB security groups, DB snapshots, and DB parameter groups for the past 14 days. Events specific to a particular DB instance, DB security group, database snapshot, or DB parameter group can be obtained by providing the name as a parameter. By default, the past hour of events are returned.

Request

- **Duration** (in the CLI: `--duration`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The number of minutes to retrieve events for.

  Default: 60

- **EndTime** (in the CLI: `--end-time`) – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The end of the time interval for which to retrieve events, specified in ISO 8601 format. For more information about ISO 8601, go to the ISO8601 Wikipedia page.

  Example: 2009-07-08T18:00Z

- **EventCategories** (in the CLI: `--event-categories`) – a String, of type: `string` (a UTF-8 encoded string).

  A list of event categories that trigger notifications for a event notification subscription.

- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  This parameter is not currently supported.

- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous DescribeEvents request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

Default: 100

Constraints: Minimum 20, maximum 100.

- **SourceIdentifier** (in the CLI: `--source-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the event source for which events are returned. If not specified, then all sources are included in the response.

  Constraints:

  - If SourceIdentifier is supplied, SourceType must also be provided.
  - If the source type is `DBInstance`, then a `DBInstanceIdentifier` must be supplied.
  - If the source type is `DBSecurityGroup`, a `DBSecurityGroupName` must be supplied.
  - If the source type is `DBParameterGroup`, a `DBParameterGroupName` must be supplied.
  - If the source type is `DBSnapshot`, a `DBSnapshotIdentifier` must be supplied.
  - Cannot end with a hyphen or contain two consecutive hyphens.
- **SourceType** (in the CLI: `--source-type`) – a SourceType, of type: `string` (a UTF-8 encoded string).

  The event source to retrieve events for. If no value is specified, all events are returned.
- **StartTime** (in the CLI: `--start-time`) – a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The beginning of the time interval to retrieve events for, specified in ISO 8601 format. For more information about ISO 8601, go to the ISO8601 Wikipedia page.

  Example: 2009-07-08T18:00Z

Response

- **Events** – An array of Event objects.

  A list of the section called “Event” instances.

- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous Events request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords` .

DescribeEventCategories (action)

The AWS CLI name for this API is: `describe-event-categories`.

Displays a list of categories for all event source types, or, if specified, for a specified source type.

Request

- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  This parameter is not currently supported.
- **SourceType** (in the CLI: `--source-type`) – a String, of type: `string` (a UTF-8 encoded string).

  The type of source that is generating the events.

  Valid values: db-instance | db-parameter-group | db-security-group | db-snapshot

Response

- **EventCategoriesMapList** – An array of EventCategoriesMap objects.

  A list of EventCategoriesMap data types.

Structures:

Event (structure)

This data type is used as a response element in the the section called "DescribeEvents" action.

Fields

- **Date** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

Specifies the date and time of the event.

- **EventCategories** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the category for the event.

- **Message** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the text of this event.

- **SourceArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the event.

- **SourceIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

  Provides the identifier for the source of the event.

- **SourceType** – This is a SourceType, of type: `string` (a UTF-8 encoded string).

  Specifies the source type for this event.

EventCategoriesMap (structure)

Contains the results of a successful invocation of the the section called "DescribeEventCategories" action.

Fields

- **EventCategories** – This is a String, of type: `string` (a UTF-8 encoded string).

  The event categories for the specified source type

- **SourceType** – This is a String, of type: `string` (a UTF-8 encoded string).

  The source type that the returned categories belong to

EventSubscription (structure)

Contains the results of a successful invocation of the the section called "DescribeEventSubscriptions" action.

Fields

- **CustomerAwsId** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon customer account associated with the event notification subscription.
- **CustSubscriptionId** – This is a String, of type: `string` (a UTF-8 encoded string).

  The event notification subscription Id.
- **Enabled** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A Boolean value indicating if the subscription is enabled. True indicates the subscription is enabled.
- **EventCategoriesList** – This is a String, of type: `string` (a UTF-8 encoded string).

  A list of event categories for the event notification subscription.
- **EventSubscriptionArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for the event subscription.
- **SnsTopicArn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The topic ARN of the event notification subscription.
- **SourceIdsList** – This is a String, of type: `string` (a UTF-8 encoded string).

  A list of source IDs for the event notification subscription.
- **SourceType** – This is a String, of type: `string` (a UTF-8 encoded string).

  The source type for the event notification subscription.
- **Status** – This is a String, of type: `string` (a UTF-8 encoded string).

  The status of the event notification subscription.

  Constraints:

  Can be one of the following: creating | modifying | deleting | active | no-permission | topic-not-exist

  The status "no-permission" indicates that Neptune no longer has permission to post to the SNS topic. The status "topic-not-exist" indicates that the topic was deleted after the subscription was created.

- **SubscriptionCreationTime** – This is a String, of type: `string` (a UTF-8 encoded string).

  The time the event notification subscription was created.

`EventSubscription` is used as the response element for:

- CreateEventSubscription
- ModifyEventSubscription
- AddSourceIdentifierToSubscription
- RemoveSourceIdentifierFromSubscription
- DeleteEventSubscription

Other Neptune APIs

Actions:

- AddTagsToResource (action)
- ListTagsForResource (action)
- RemoveTagsFromResource (action)
- ApplyPendingMaintenanceAction (action)
- DescribePendingMaintenanceActions (action)
- DescribeDBEngineVersions (action)

Structures:

- DBEngineVersion (structure)
- EngineDefaults (structure)
- PendingMaintenanceAction (structure)
- ResourcePendingMaintenanceActions (structure)
- UpgradeTarget (structure)
- Tag (structure)

AddTagsToResource (action)

The AWS CLI name for this API is: `add-tags-to-resource`.

Adds metadata tags to an Amazon Neptune resource. These tags can also be used with cost allocation reporting to track cost associated with Amazon Neptune resources, or used in a Condition statement in an IAM policy for Amazon Neptune.

Request

- **ResourceName** (in the CLI: `--resource-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Neptune resource that the tags are added to. This value is an Amazon Resource Name (ARN). For information about creating an ARN, see Constructing an Amazon Resource Name (ARN).
- **Tags** (in the CLI: `--tags`) – *Required:* An array of Tag objects.

  The tags to be assigned to the Amazon Neptune resource.

Response

- *No Response parameters.*

Errors

- DBInstanceNotFoundFault
- DBSnapshotNotFoundFault
- DBClusterNotFoundFault

ListTagsForResource (action)

The AWS CLI name for this API is: `list-tags-for-resource`.

Lists all tags on an Amazon Neptune resource.

Request

- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

This parameter is not currently supported.

- **ResourceName** (in the CLI: `--resource-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Neptune resource with tags to be listed. This value is an Amazon Resource Name (ARN). For information about creating an ARN, see Constructing an Amazon Resource Name (ARN).

Response

- **TagList** – An array of Tag objects.

  List of tags returned by the ListTagsForResource operation.

Errors

- DBInstanceNotFoundFault
- DBSnapshotNotFoundFault
- DBClusterNotFoundFault

RemoveTagsFromResource (action)

The AWS CLI name for this API is: `remove-tags-from-resource`.

Removes metadata tags from an Amazon Neptune resource.

Request

- **ResourceName** (in the CLI: `--resource-name`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Neptune resource that the tags are removed from. This value is an Amazon Resource Name (ARN). For information about creating an ARN, see Constructing an Amazon Resource Name (ARN).

- **TagKeys** (in the CLI: `--tag-keys`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

The tag key (name) of the tag to be removed.

Response

- *No Response parameters.*

Errors

- DBInstanceNotFoundFault
- DBSnapshotNotFoundFault
- DBClusterNotFoundFault

ApplyPendingMaintenanceAction (action)

The AWS CLI name for this API is: `apply-pending-maintenance-action`.

Applies a pending maintenance action to a resource (for example, to a DB instance).

Request

- **ApplyAction** (in the CLI: `--apply-action`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The pending maintenance action to apply to this resource.

  Valid values: `system-update`, `db-upgrade`
- **OptInType** (in the CLI: `--opt-in-type`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A value that specifies the type of opt-in request, or undoes an opt-in request. An opt-in request of type `immediate` can't be undone.

  Valid values:
  - `immediate` - Apply the maintenance action immediately.
  - `next-maintenance` - Apply the maintenance action during the next maintenance window for the resource.
  - `undo-opt-in` - Cancel any existing `next-maintenance` opt-in requests.

- **ResourceIdentifier** (in the CLI: `--resource-identifier`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of the resource that the pending maintenance action applies to. For information about creating an ARN, see Constructing an Amazon Resource Name (ARN).

Response

Describes the pending maintenance actions for a resource.

- **PendingMaintenanceActionDetails** – An array of PendingMaintenanceAction objects.

  A list that provides details about the pending maintenance actions for the resource.
- **ResourceIdentifier** – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of the resource that has pending maintenance actions.

Errors

- ResourceNotFoundFault

DescribePendingMaintenanceActions (action)

The AWS CLI name for this API is: `describe-pending-maintenance-actions`.

Returns a list of resources (for example, DB instances) that have at least one pending maintenance action.

Request

- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  A filter that specifies one or more resources to return pending maintenance actions for.

  Supported filters:

  - `db-cluster-id` - Accepts DB cluster identifiers and DB cluster Amazon Resource Names (ARNs). The results list will only include pending maintenance actions for the DB clusters identified by these ARNs.

- `db-instance-id` - Accepts DB instance identifiers and DB instance ARNs. The results list will only include pending maintenance actions for the DB instances identified by these ARNs.

- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous `DescribePendingMaintenanceActions` request. If this parameter is specified, the response includes only records beyond the marker, up to a number of records specified by `MaxRecords`.

- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more records exist than the specified `MaxRecords` value, a pagination token called a marker is included in the response so that the remaining results can be retrieved.

  Default: 100

  Constraints: Minimum 20, maximum 100.

- **ResourceIdentifier** (in the CLI: `--resource-identifier`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of a resource to return pending maintenance actions for.

Response

- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous `DescribePendingMaintenanceActions` request. If this parameter is specified, the response includes only records beyond the marker, up to a number of records specified by `MaxRecords`.

- **PendingMaintenanceActions** – An array of ResourcePendingMaintenanceActions objects.

  A list of the pending maintenance actions for the resource.

Errors

- ResourceNotFoundFault

DescribeDBEngineVersions (action)

The AWS CLI name for this API is: `describe-db-engine-versions`.

Returns a list of the available DB engines.

Request

- **DBParameterGroupFamily** (in the CLI: `--db-parameter-group-family`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of a specific DB parameter group family to return details for.

  Constraints:

  - If supplied, must match an existing DBParameterGroupFamily.
- **DefaultOnly** (in the CLI: `--default-only`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates that only the default version of the specified engine or engine and major version combination is returned.
- **Engine** (in the CLI: `--engine`) – a String, of type: `string` (a UTF-8 encoded string).

  The database engine to return.
- **EngineVersion** (in the CLI: `--engine-version`) – a String, of type: `string` (a UTF-8 encoded string).

  The database engine version to return.

  Example: `5.1.49`
- **Filters** (in the CLI: `--filters`) – An array of Filter objects.

  Not currently supported.
- **ListSupportedCharacterSets** (in the CLI: `--list-supported-character-sets`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If this parameter is specified and the requested engine supports the `CharacterSetName` parameter for `CreateDBInstance`, the response includes a list of supported character sets for each engine version.

- **ListSupportedTimezones** (in the CLI: `--list-supported-timezones`) – a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  If this parameter is specified and the requested engine supports the `TimeZone` parameter for `CreateDBInstance`, the response includes a list of supported time zones for each engine version.
- **Marker** (in the CLI: `--marker`) – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.
- **MaxRecords** (in the CLI: `--max-records`) – an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The maximum number of records to include in the response. If more than the `MaxRecords` value is available, a pagination token called a marker is included in the response so that the following results can be retrieved.

  Default: 100

  Constraints: Minimum 20, maximum 100.

Response

- **DBEngineVersions** – An array of DBEngineVersion objects.

  A list of `DBEngineVersion` elements.
- **Marker** – a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords`.

Structures:

DBEngineVersion (structure)

This data type is used as a response element in the action the section called "DescribeDBEngineVersions".

Fields

- **DBEngineDescription** – This is a String, of type: `string` (a UTF-8 encoded string).

  The description of the database engine.
- **DBEngineVersionDescription** – This is a String, of type: `string` (a UTF-8 encoded string).

  The description of the database engine version.
- **DBParameterGroupFamily** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB parameter group family for the database engine.
- **Engine** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the database engine.
- **EngineVersion** – This is a String, of type: `string` (a UTF-8 encoded string).

  The version number of the database engine.
- **ExportableLogTypes** – This is a String, of type: `string` (a UTF-8 encoded string).

  The types of logs that the database engine has available for export to CloudWatch Logs.
- **SupportedTimezones** – This is An array of Timezone objects.

  A list of the time zones supported by this engine for the `Timezone` parameter of the `CreateDBInstance` action.
- **SupportsGlobalDatabases** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether you can use Aurora global databases with a specific DB engine version.
- **SupportsLogExportsToCloudwatchLogs** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether the engine version supports exporting the log types specified by ExportableLogTypes to CloudWatch Logs.
- **SupportsReadReplica** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether the database engine version supports read replicas.
- **ValidUpgradeTarget** – This is An array of UpgradeTarget objects.

A list of engine versions that this database engine version can be upgraded to.

EngineDefaults (structure)

Contains the result of a successful invocation of the the section called "DescribeEngineDefaultParameters" action.

Fields

- **DBParameterGroupFamily** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the name of the DB parameter group family that the engine default parameters apply to.

- **Marker** – This is a String, of type: `string` (a UTF-8 encoded string).

  An optional pagination token provided by a previous EngineDefaults request. If this parameter is specified, the response includes only records beyond the marker, up to the value specified by `MaxRecords` .

- **Parameters** – This is An array of Parameter objects.

  Contains a list of engine default parameters.

`EngineDefaults` is used as the response element for:

- DescribeEngineDefaultParameters
- DescribeEngineDefaultClusterParameters

PendingMaintenanceAction (structure)

Provides information about a pending maintenance action for a resource.

Fields

- **Action** – This is a String, of type: `string` (a UTF-8 encoded string).

  The type of pending maintenance action that is available for the resource.

- **AutoAppliedAfterDate** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The date of the maintenance window when the action is applied. The maintenance action is applied to the resource during its first maintenance window after this date. If this date is specified, any `next-maintenance` opt-in requests are ignored.
- **CurrentApplyDate** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The effective date when the pending maintenance action is applied to the resource. This date takes into account opt-in requests received from the the section called "ApplyPendingMaintenanceAction" API, the `AutoAppliedAfterDate`, and the `ForcedApplyDate`. This value is blank if an opt-in request has not been received and nothing has been specified as `AutoAppliedAfterDate` or `ForcedApplyDate`.
- **Description** – This is a String, of type: `string` (a UTF-8 encoded string).

  A description providing more detail about the maintenance action.
- **ForcedApplyDate** – This is a TStamp, of type: `timestamp` (a point in time, generally defined as an offset from midnight 1970-01-01).

  The date when the maintenance action is automatically applied. The maintenance action is applied to the resource on this date regardless of the maintenance window for the resource. If this date is specified, any `immediate` opt-in requests are ignored.
- **OptInStatus** – This is a String, of type: `string` (a UTF-8 encoded string).

  Indicates the type of opt-in request that has been received for the resource.

ResourcePendingMaintenanceActions (structure)

Describes the pending maintenance actions for a resource.

Fields

- **PendingMaintenanceActionDetails** – This is An array of PendingMaintenanceAction objects.

  A list that provides details about the pending maintenance actions for the resource.
- **ResourceIdentifier** – This is a String, of type: `string` (a UTF-8 encoded string).

The ARN of the resource that has pending maintenance actions.

`ResourcePendingMaintenanceActions` is used as the response element for:

- ApplyPendingMaintenanceAction

UpgradeTarget (structure)

The version of the database engine that a DB instance can be upgraded to.

Fields

- **AutoUpgrade** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether the target version is applied to any source DB instances that have AutoMinorVersionUpgrade set to true.
- **Description** – This is a String, of type: `string` (a UTF-8 encoded string).

  The version of the database engine that a DB instance can be upgraded to.
- **Engine** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the upgrade target database engine.
- **EngineVersion** – This is a String, of type: `string` (a UTF-8 encoded string).

  The version number of the upgrade target database engine.
- **IsMajorVersionUpgrade** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether a database engine is upgraded to a major version.
- **SupportsGlobalDatabases** – This is a BooleanOptional, of type: `boolean` (a Boolean (true or false) value).

  A value that indicates whether you can use Neptune global databases with the target engine version.

Tag (structure)

Metadata assigned to an Amazon Neptune resource consisting of a key-value pair.

Fields

- **Key** – This is a String, of type: `string` (a UTF-8 encoded string).

  A key is the required name of the tag. The string value can be from 1 to 128 Unicode characters in length and can't be prefixed with `aws:` or `rds:`. The string can only contain the set of Unicode letters, digits, white-space, '_', '.', '/', '=', '+', '-' (Java regex: "^([\\p{L}\\p{Z}\\p{N}_.:/=+\\-]*)$").
- **Value** – This is a String, of type: `string` (a UTF-8 encoded string).

  A value is the optional value of the tag. The string value can be from 1 to 256 Unicode characters in length and can't be prefixed with `aws:` or `rds:`. The string can only contain the set of Unicode letters, digits, white-space, '_', '.', '/', '=', '+', '-' (Java regex: "^([\\p{L}\\p{Z}\\p{N}_.:/=+\\-]*)$").

Common Neptune Datatypes

Structures:

- AvailabilityZone (structure)
- DBSecurityGroupMembership (structure)
- DomainMembership (structure)
- DoubleRange (structure)
- Endpoint (structure)
- Filter (structure)
- Range (structure)
- ServerlessV2ScalingConfiguration (structure)
- ServerlessV2ScalingConfigurationInfo (structure)
- Timezone (structure)
- VpcSecurityGroupMembership (structure)

AvailabilityZone (structure)

Specifies an Availability Zone.

Fields

- **Name** – This is a String, of type: `string` (a UTF-8 encoded string).

The name of the availability zone.

DBSecurityGroupMembership (structure)

Specifies membership in a designated DB security group.

Fields

- **DBSecurityGroupName** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the DB security group.
- **Status** – This is a String, of type: `string` (a UTF-8 encoded string).

  The status of the DB security group.

DomainMembership (structure)

An Active Directory Domain membership record associated with a DB instance.

Fields

- **Domain** – This is a String, of type: `string` (a UTF-8 encoded string).

  The identifier of the Active Directory Domain.
- **FQDN** – This is a String, of type: `string` (a UTF-8 encoded string).

  The fully qualified domain name of the Active Directory Domain.
- **IAMRoleName** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the IAM role to be used when making API calls to the Directory Service.
- **Status** – This is a String, of type: `string` (a UTF-8 encoded string).

  The status of the DB instance's Active Directory Domain membership, such as joined, pending-join, failed etc).

DoubleRange (structure)

A range of double values.

Fields

- **From** – This is a Double, of type: `double` (a double-precisionn IEEE 754 floating-point number).

  The minimum value in the range.
- **To** – This is a Double, of type: `double` (a double-precisionn IEEE 754 floating-point number).

  The maximum value in the range.

Endpoint (structure)

Specifies a connection endpoint.

For the data structure that represents Amazon Neptune DB cluster endpoints, see `DBClusterEndpoint`.

Fields

- **Address** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the DNS address of the DB instance.
- **HostedZoneId** – This is a String, of type: `string` (a UTF-8 encoded string).

  Specifies the ID that Amazon Route 53 assigns when you create a hosted zone.
- **Port** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  Specifies the port that the database engine is listening on.

Filter (structure)

This type is not currently supported.

Fields

- **Name** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  This parameter is not currently supported.
- **Values** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  This parameter is not currently supported.

Range (structure)

A range of integer values.

Fields

- **From** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  The minimum value in the range.
- **Step** – This is an IntegerOptional, of type: `integer` (a signed 32-bit integer).

  The step value for the range. For example, if you have a range of 5,000 to 10,000, with a step value of 1,000, the valid values start at 5,000 and step up by 1,000. Even though 7,500 is within the range, it isn't a valid value for the range. The valid values are 5,000, 6,000, 7,000, 8,000...
- **To** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  The maximum value in the range.

ServerlessV2ScalingConfiguration (structure)

Contains the scaling configuration of a Neptune Serverless DB cluster.

For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

Fields

- **MaxCapacity** – This is a DoubleOptional, of type: `double` (a double-precisionn IEEE 754 floating-point number).

  The maximum number of Neptune capacity units (NCUs) for a DB instance in a Neptune Serverless cluster. You can specify NCU values in half-step increments, such as 40, 40.5, 41, and so on.
- **MinCapacity** – This is a DoubleOptional, of type: `double` (a double-precisionn IEEE 754 floating-point number).

  The minimum number of Neptune capacity units (NCUs) for a DB instance in a Neptune Serverless cluster. You can specify NCU values in half-step increments, such as 8, 8.5, 9, and so on.

ServerlessV2ScalingConfigurationInfo (structure)

Shows the scaling configuration for a Neptune Serverless DB cluster.

For more information, see Using Amazon Neptune Serverless in the *Amazon Neptune User Guide*.

Fields

- **MaxCapacity** – This is a DoubleOptional, of type: `double` (a double-precisionn IEEE 754 floating-point number).

  The maximum number of Neptune capacity units (NCUs) for a DB instance in a Neptune Serverless cluster. You can specify NCU values in half-step increments, such as 40, 40.5, 41, and so on.
- **MinCapacity** – This is a DoubleOptional, of type: `double` (a double-precisionn IEEE 754 floating-point number).

  The minimum number of Neptune capacity units (NCUs) for a DB instance in a Neptune Serverless cluster. You can specify NCU values in half-step increments, such as 8, 8.5, 9, and so on.

Timezone (structure)

A time zone associated with a the section called “DBInstance”.

Fields

- **TimezoneName** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the time zone.

VpcSecurityGroupMembership (structure)

This data type is used as a response element for queries on VPC security group membership.

Fields

- **Status** – This is a String, of type: `string` (a UTF-8 encoded string).

  The status of the VPC security group.

- **VpcSecurityGroupId** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the VPC security group.

Neptune Exceptions Specific to Individual APIs

Exceptions:

- AuthorizationAlreadyExistsFault (structure)
- AuthorizationNotFoundFault (structure)
- AuthorizationQuotaExceededFault (structure)
- CertificateNotFoundFault (structure)
- DBClusterAlreadyExistsFault (structure)
- DBClusterNotFoundFault (structure)
- DBClusterParameterGroupNotFoundFault (structure)
- DBClusterQuotaExceededFault (structure)
- DBClusterRoleAlreadyExistsFault (structure)
- DBClusterRoleNotFoundFault (structure)
- DBClusterRoleQuotaExceededFault (structure)
- DBClusterSnapshotAlreadyExistsFault (structure)
- DBClusterSnapshotNotFoundFault (structure)
- DBInstanceAlreadyExistsFault (structure)
- DBInstanceNotFoundFault (structure)
- DBLogFileNotFoundFault (structure)
- DBParameterGroupAlreadyExistsFault (structure)
- DBParameterGroupNotFoundFault (structure)
- DBParameterGroupQuotaExceededFault (structure)
- DBSecurityGroupAlreadyExistsFault (structure)
- DBSecurityGroupNotFoundFault (structure)
- DBSecurityGroupNotSupportedFault (structure)
- DBSecurityGroupQuotaExceededFault (structure)

- DBSnapshotAlreadyExistsFault (structure)
- DBSnapshotNotFoundFault (structure)
- DBSubnetGroupAlreadyExistsFault (structure)
- DBSubnetGroupDoesNotCoverEnoughAZs (structure)
- DBSubnetGroupNotAllowedFault (structure)
- DBSubnetGroupNotFoundFault (structure)
- DBSubnetGroupQuotaExceededFault (structure)
- DBSubnetQuotaExceededFault (structure)
- DBUpgradeDependencyFailureFault (structure)
- DomainNotFoundFault (structure)
- EventSubscriptionQuotaExceededFault (structure)
- GlobalClusterAlreadyExistsFault (structure)
- GlobalClusterNotFoundFault (structure)
- GlobalClusterQuotaExceededFault (structure)
- InstanceQuotaExceededFault (structure)
- InsufficientDBClusterCapacityFault (structure)
- InsufficientDBInstanceCapacityFault (structure)
- InsufficientStorageClusterCapacityFault (structure)
- InvalidDBClusterEndpointStateFault (structure)
- InvalidDBClusterSnapshotStateFault (structure)
- InvalidDBClusterStateFault (structure)
- InvalidDBInstanceStateFault (structure)
- InvalidDBParameterGroupStateFault (structure)
- InvalidDBSecurityGroupStateFault (structure)
- InvalidDBSnapshotStateFault (structure)
- InvalidDBSubnetGroupFault (structure)
- InvalidDBSubnetGroupStateFault (structure)
- InvalidDBSubnetStateFault (structure)
- InvalidEventSubscriptionStateFault (structure)
- InvalidGlobalClusterStateFault (structure)

- InvalidOptionGroupStateFault (structure)
- InvalidRestoreFault (structure)
- InvalidSubnet (structure)
- InvalidVPCNetworkStateFault (structure)
- KMSKeyNotAccessibleFault (structure)
- OptionGroupNotFoundFault (structure)
- PointInTimeRestoreNotEnabledFault (structure)
- ProvisionedIopsNotAvailableInAZFault (structure)
- ResourceNotFoundFault (structure)
- SNSInvalidTopicFault (structure)
- SNSNoAuthorizationFault (structure)
- SNSTopicArnNotFoundFault (structure)
- SharedSnapshotQuotaExceededFault (structure)
- SnapshotQuotaExceededFault (structure)
- SourceNotFoundFault (structure)
- StorageQuotaExceededFault (structure)
- StorageTypeNotSupportedFault (structure)
- SubnetAlreadyInUse (structure)
- SubscriptionAlreadyExistFault (structure)
- SubscriptionCategoryNotFoundFault (structure)
- SubscriptionNotFoundFault (structure)

AuthorizationAlreadyExistsFault (structure)

HTTP status code returned: 400.

The specified CIDRIP or EC2 security group is already authorized for the specified DB security group.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

A message describing the details of the problem.

AuthorizationNotFoundFault (structure)

HTTP status code returned: 404.

Specified CIDRIP or EC2 security group is not authorized for the specified DB security group.

Neptune may not also be authorized via IAM to perform necessary actions on your behalf.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

AuthorizationQuotaExceededFault (structure)

HTTP status code returned: 400.

DB security group authorization quota has been reached.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

CertificateNotFoundFault (structure)

HTTP status code returned: 404.

CertificateIdentifier does not refer to an existing certificate.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBClusterAlreadyExistsFault (structure)

HTTP status code returned: 400.

User already has a DB cluster with the given identifier.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBClusterNotFoundFault (structure)

HTTP status code returned: 404.

DBClusterIdentifier does not refer to an existing DB cluster.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBClusterParameterGroupNotFoundFault (structure)

HTTP status code returned: 404.

DBClusterParameterGroupName does not refer to an existing DB Cluster parameter group.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBClusterQuotaExceededFault (structure)

HTTP status code returned: 403.

User attempted to create a new DB cluster and the user has already reached the maximum allowed DB cluster quota.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBClusterRoleAlreadyExistsFault (structure)

HTTP status code returned: 400.

The specified IAM role Amazon Resource Name (ARN) is already associated with the specified DB cluster.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBClusterRoleNotFoundFault (structure)

HTTP status code returned: 404.

The specified IAM role Amazon Resource Name (ARN) is not associated with the specified DB cluster.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBClusterRoleQuotaExceededFault (structure)

HTTP status code returned: 400.

You have exceeded the maximum number of IAM roles that can be associated with the specified DB cluster.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBClusterSnapshotAlreadyExistsFault (structure)

HTTP status code returned: 400.

User already has a DB cluster snapshot with the given identifier.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBClusterSnapshotNotFoundFault (structure)

HTTP status code returned: 404.

DBClusterSnapshotIdentifier does not refer to an existing DB cluster snapshot.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBInstanceAlreadyExistsFault (structure)

HTTP status code returned: 400.

User already has a DB instance with the given identifier.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBInstanceNotFoundFault (structure)

HTTP status code returned: 404.

DBInstanceIdentifier does not refer to an existing DB instance.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBLogFileNotFoundFault (structure)

HTTP status code returned: 404.

LogFileName does not refer to an existing DB log file.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBParameterGroupAlreadyExistsFault (structure)

HTTP status code returned: 400.

A DB parameter group with the same name exists.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBParameterGroupNotFoundFault (structure)

HTTP status code returned: 404.

DBParameterGroupName does not refer to an existing DB parameter group.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBParameterGroupQuotaExceededFault (structure)

HTTP status code returned: 400.

Request would result in user exceeding the allowed number of DB parameter groups.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBSecurityGroupAlreadyExistsFault (structure)

HTTP status code returned: 400.

A DB security group with the name specified in *DBSecurityGroupName* already exists.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBSecurityGroupNotFoundFault (structure)

HTTP status code returned: 404.

DBSecurityGroupName does not refer to an existing DB security group.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBSecurityGroupNotSupportedFault (structure)

HTTP status code returned: 400.

A DB security group is not allowed for this action.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBSecurityGroupQuotaExceededFault (structure)

HTTP status code returned: 400.

Request would result in user exceeding the allowed number of DB security groups.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBSnapshotAlreadyExistsFault (structure)

HTTP status code returned: 400.

DBSnapshotIdentifier is already used by an existing snapshot.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBSnapshotNotFoundFault (structure)

HTTP status code returned: 404.

DBSnapshotIdentifier does not refer to an existing DB snapshot.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBSubnetGroupAlreadyExistsFault (structure)

HTTP status code returned: 400.

DBSubnetGroupName is already used by an existing DB subnet group.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBSubnetGroupDoesNotCoverEnoughAZs (structure)

HTTP status code returned: 400.

Subnets in the DB subnet group should cover at least two Availability Zones unless there is only one Availability Zone.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBSubnetGroupNotAllowedFault (structure)

HTTP status code returned: 400.

Indicates that the DBSubnetGroup should not be specified while creating read replicas that lie in the same region as the source instance.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBSubnetGroupNotFoundFault (structure)

HTTP status code returned: 404.

DBSubnetGroupName does not refer to an existing DB subnet group.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBSubnetGroupQuotaExceededFault (structure)

HTTP status code returned: 400.

Request would result in user exceeding the allowed number of DB subnet groups.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBSubnetQuotaExceededFault (structure)

HTTP status code returned: 400.

Request would result in user exceeding the allowed number of subnets in a DB subnet groups.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DBUpgradeDependencyFailureFault (structure)

HTTP status code returned: 400.

The DB upgrade failed because a resource the DB depends on could not be modified.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

DomainNotFoundFault (structure)

HTTP status code returned: 404.

Domain does not refer to an existing Active Directory Domain.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

EventSubscriptionQuotaExceededFault (structure)

HTTP status code returned: 400.

You have exceeded the number of events you can subscribe to.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

GlobalClusterAlreadyExistsFault (structure)

HTTP status code returned: 400.

The `GlobalClusterIdentifier` already exists. Choose a new global database identifier (unique name) to create a new global database cluster.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

GlobalClusterNotFoundFault (structure)

HTTP status code returned: 404.

The `GlobalClusterIdentifier` doesn't refer to an existing global database cluster.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

GlobalClusterQuotaExceededFault (structure)

HTTP status code returned: 400.

The number of global database clusters for this account is already at the maximum allowed.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InstanceQuotaExceededFault (structure)

HTTP status code returned: 400.

Request would result in user exceeding the allowed number of DB instances.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InsufficientDBClusterCapacityFault (structure)

HTTP status code returned: 403.

The DB cluster does not have enough capacity for the current operation.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InsufficientDBInstanceCapacityFault (structure)

HTTP status code returned: 400.

Specified DB instance class is not available in the specified Availability Zone.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InsufficientStorageClusterCapacityFault (structure)

HTTP status code returned: 400.

There is insufficient storage available for the current action. You may be able to resolve this error by updating your subnet group to use different Availability Zones that have more storage available.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

A message describing the details of the problem.

InvalidDBClusterEndpointStateFault (structure)

HTTP status code returned: 400.

The requested operation cannot be performed on the endpoint while the endpoint is in this state.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidDBClusterSnapshotStateFault (structure)

HTTP status code returned: 400.

The supplied value is not a valid DB cluster snapshot state.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidDBClusterStateFault (structure)

HTTP status code returned: 400.

The DB cluster is not in a valid state.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidDBInstanceStateFault (structure)

HTTP status code returned: 400.

The specified DB instance is not in the *available* state.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidDBParameterGroupStateFault (structure)

HTTP status code returned: 400.

The DB parameter group is in use or is in an invalid state. If you are attempting to delete the parameter group, you cannot delete it when the parameter group is in this state.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidDBSecurityGroupStateFault (structure)

HTTP status code returned: 400.

The state of the DB security group does not allow deletion.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidDBSnapshotStateFault (structure)

HTTP status code returned: 400.

The state of the DB snapshot does not allow deletion.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidDBSubnetGroupFault (structure)

HTTP status code returned: 400.

Indicates the DBSubnetGroup does not belong to the same VPC as that of an existing cross region read replica of the same source instance.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidDBSubnetGroupStateFault (structure)

HTTP status code returned: 400.

The DB subnet group cannot be deleted because it is in use.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidDBSubnetStateFault (structure)

HTTP status code returned: 400.

The DB subnet is not in the *available* state.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidEventSubscriptionStateFault (structure)

HTTP status code returned: 400.

The event subscription is in an invalid state.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidGlobalClusterStateFault (structure)

HTTP status code returned: 400.

The global cluster is in an invalid state and can't perform the requested operation.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidOptionGroupStateFault (structure)

HTTP status code returned: 400.

The option group is not in the *available* state.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidRestoreFault (structure)

HTTP status code returned: 400.

Cannot restore from vpc backup to non-vpc DB instance.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidSubnet (structure)

HTTP status code returned: 400.

The requested subnet is invalid, or multiple subnets were requested that are not all in a common VPC.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

InvalidVPCNetworkStateFault (structure)

HTTP status code returned: 400.

DB subnet group does not cover all Availability Zones after it is created because users' change.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

KMSKeyNotAccessibleFault (structure)

HTTP status code returned: 400.

Error accessing KMS key.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

OptionGroupNotFoundFault (structure)

HTTP status code returned: 404.

The designated option group could not be found.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

PointInTimeRestoreNotEnabledFault (structure)

HTTP status code returned: 400.

SourceDBInstanceIdentifier refers to a DB instance with *BackupRetentionPeriod* equal to 0.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

ProvisionedIopsNotAvailableInAZFault (structure)

HTTP status code returned: 400.

Provisioned IOPS not available in the specified Availability Zone.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

A message describing the details of the problem.

ResourceNotFoundFault (structure)

HTTP status code returned: 404.

The specified resource ID was not found.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

SNSInvalidTopicFault (structure)

HTTP status code returned: 400.

The SNS topic is invalid.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

SNSNoAuthorizationFault (structure)

HTTP status code returned: 400.

There is no SNS authorization.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

SNSTopicArnNotFoundFault (structure)

HTTP status code returned: 404.

The ARN of the SNS topic could not be found.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

SharedSnapshotQuotaExceededFault (structure)

HTTP status code returned: 400.

You have exceeded the maximum number of accounts that you can share a manual DB snapshot with.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

SnapshotQuotaExceededFault (structure)

HTTP status code returned: 400.

Request would result in user exceeding the allowed number of DB snapshots.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

SourceNotFoundFault (structure)

HTTP status code returned: 404.

The source could not be found.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

StorageQuotaExceededFault (structure)

HTTP status code returned: 400.

Request would result in user exceeding the allowed amount of storage available across all DB instances.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

StorageTypeNotSupportedFault (structure)

HTTP status code returned: 400.

StorageType specified cannot be associated with the DB Instance.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

SubnetAlreadyInUse (structure)

HTTP status code returned: 400.

The DB subnet is already in use in the Availability Zone.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

SubscriptionAlreadyExistFault (structure)

HTTP status code returned: 400.

This subscription already exists.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

SubscriptionCategoryNotFoundFault (structure)

HTTP status code returned: 404.

The designated subscription category could not be found.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

SubscriptionNotFoundFault (structure)

HTTP status code returned: 404.

The designated subscription could not be found.

Fields

- **message** – This is an ExceptionMessage, of type: `string` (a UTF-8 encoded string).

  A message describing the details of the problem.

Amazon Neptune Data API Reference

This chapter documents the Neptune data API operations that you can use to load, access, and maintain the data in your Neptune graph.

The Neptune data API provides SDK support for loading data, running queries, getting information about your data, and running machine-learning operations. It supports the Gremlin and openCypher query languages in Neptune, and is available in all SDK languages. It automatically signs API requests and greatly simplifies integrating Neptune into applications.

Contents

- Neptune dataplane engine, fast reset, and general structure APIs
  - GetEngineStatus (action)
  - ExecuteFastReset (action)
  - Engine operation structures:
  - QueryLanguageVersion (structure)
  - FastResetToken (structure)
- Neptune Query APIs
  - ExecuteGremlinQuery (action)
  - ExecuteGremlinExplainQuery (action)
  - ExecuteGremlinProfileQuery (action)
  - ListGremlinQueries (action)
  - GetGremlinQueryStatus (action)
  - CancelGremlinQuery (action)
  - openCypher query actions:
  - ExecuteOpenCypherQuery (action)
  - ExecuteOpenCypherExplainQuery (action)
  - ListOpenCypherQueries (action)
  - GetOpenCypherQueryStatus (action)
  - CancelOpenCypherQuery (action)
  - Query structures:
  - QueryEvalStats (structure)

- GremlinQueryStatus (structure)
- GremlinQueryStatusAttributes (structure)
- Neptune data plane bulk loader APIs
  - StartLoaderJob (action)
  - GetLoaderJobStatus (action)
  - ListLoaderJobs (action)
  - CancelLoaderJob (action)
  - Bulk load structure:
  - LoaderIdResult (structure)
- Neptune streams dataplane API
  - GetPropertygraphStream (action)
  - Stream data structures:
  - PropertygraphRecord (structure)
  - PropertygraphData (structure)
- Neptune dataplane statistics and graph summary APIs
  - GetPropertygraphStatistics (action)
  - ManagePropertygraphStatistics (action)
  - DeletePropertygraphStatistics (action)
  - GetPropertygraphSummary (action)
  - Statistics structures:
  - Statistics (structure)
  - StatisticsSummary (structure)
  - DeleteStatisticsValueMap (structure)
  - RefreshStatisticsIdMap (structure)
  - NodeStructure (structure)
  - EdgeStructure (structure)
  - SubjectStructure (structure)
  - PropertygraphSummaryValueMap (structure)
  - PropertygraphSummary (structure)
- Neptune ML data-processing API

- StartMLDataProcessingJob (action)
- ListMLDataProcessingJobs (action)
- GetMLDataProcessingJob (action)
- CancelMLDataProcessingJob (action)
- ML general-purpose structures:
- MlResourceDefinition (structure)
- MlConfigDefinition (structure)
- Neptune ML model training API
  - StartMLModelTrainingJob (action)
  - ListMLModelTrainingJobs (action)
  - GetMLModelTrainingJob (action)
  - CancelMLModelTrainingJob (action)
  - Model training structures:
  - CustomModelTrainingParameters (structure)
- Neptune ML model transform API
  - StartMLModelTransformJob (action)
  - ListMLModelTransformJobs (action)
  - GetMLModelTransformJob (action)
  - CancelMLModelTransformJob (action)
  - Model transform structures:
  - CustomModelTransformParameters (structure)
- Neptune ML inference endpoint API
  - CreateMLEndpoint (action)
  - ListMLEndpoints (action)
  - GetMLEndpoint (action)
  - DeleteMLEndpoint (action)
- Neptune dataplane API Exceptions
  - AccessDeniedException (structure)
  - BadRequestException (structure)

  - BulkLoadIdNotFoundException (structure)

- CancelledByUserException (structure)
- ClientTimeoutException (structure)
- ConcurrentModificationException (structure)
- ConstraintViolationException (structure)
- ExpiredStreamException (structure)
- FailureByQueryException (structure)
- IllegalArgumentException (structure)
- InternalFailureException (structure)
- InvalidArgumentException (structure)
- InvalidNumericDataException (structure)
- InvalidParameterException (structure)
- LoadUrlAccessDeniedException (structure)
- MalformedQueryException (structure)
- MemoryLimitExceededException (structure)
- MethodNotAllowedException (structure)
- MissingParameterException (structure)
- MLResourceNotFoundException (structure)
- ParsingException (structure)
- PreconditionsFailedException (structure)
- QueryLimitExceededException (structure)
- QueryLimitException (structure)
- QueryTooLargeException (structure)
- ReadOnlyViolationException (structure)
- S3Exception (structure)
- ServerShutdownException (structure)
- StatisticsNotAvailableException (structure)
- StreamRecordsNotFoundException (structure)
- ThrottlingException (structure)
- TimeLimitExceededException (structure)
- TooManyRequestsException (structure)

- UnsupportedOperationException (structure)
- UnloadUrlAccessDeniedException (structure)

Neptune dataplane engine, fast reset, and general structure APIs

Engine operations:

- GetEngineStatus (action)
- ExecuteFastReset (action)

Engine operation structures:

- QueryLanguageVersion (structure)
- FastResetToken (structure)

GetEngineStatus (action)

The AWS CLI name for this API is: `get-engine-status`.

Retrieves the status of the graph database on the host.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:GetEngineStatus IAM action in that cluster.

Request

- *No Request parameters.*

Response

- **dbEngineVersion** – a String, of type: `string` (a UTF-8 encoded string).

  Set to the Neptune engine version running on your DB cluster. If this engine version has been manually patched since it was released, the version number is prefixed by `Patch-`.
- **dfeQueryEngine** – a String, of type: `string` (a UTF-8 encoded string).

Set to `enabled` if the DFE engine is fully enabled, or to `viaQueryHint` (the default) if the DFE engine is only used with queries that have the `useDFE` query hint set to `true`.

- **features** – It is a map array of key-value pairs where:

  Each key is a a String, of type: `string` (a UTF-8 encoded string).

  Each value is a a Document, of type: `document` (a protocol-agnostic open content represented by a JSON-like data model).

  Contains status information about the features enabled on your DB cluster.

- **gremlin** – A QueryLanguageVersion object.

  Contains information about the Gremlin query language available on your cluster. Specifically, it contains a version field that specifies the current TinkerPop version being used by the engine.

- **labMode** – It is a map array of key-value pairs where:

  Each key is a a String, of type: `string` (a UTF-8 encoded string).

  Each value is a a String, of type: `string` (a UTF-8 encoded string).

  Contains Lab Mode settings being used by the engine.

- **opencypher** – A QueryLanguageVersion object.

  Contains information about the openCypher query language available on your cluster. Specifically, it contains a version field that specifies the current operCypher version being used by the engine.

- **role** – a String, of type: `string` (a UTF-8 encoded string).

  Set to `reader` if the instance is a read-replica, or to `writer` if the instance is the primary instance.

- **rollingBackTrxCount** – an Integer, of type: `integer` (a signed 32-bit integer).

  If there are transactions being rolled back, this field is set to the number of such transactions. If there are none, the field doesn't appear at all.

- **rollingBackTrxEarliestStartTime** – a String, of type: `string` (a UTF-8 encoded string).

  Set to the start time of the earliest transaction being rolled back. If no transactions are being rolled back, the field doesn't appear at all.

- **settings** – It is a map array of key-value pairs where:

  Each key is a a String, of type: `string` (a UTF-8 encoded string).

  Each value is a a String, of type: `string` (a UTF-8 encoded string).

  Contains information about the current settings on your DB cluster. For example, contains the current cluster query timeout setting (`clusterQueryTimeoutInMs`).
- **sparql** – A QueryLanguageVersion object.

  Contains information about the SPARQL query language available on your cluster. Specifically, it contains a version field that specifies the current SPARQL version being used by the engine.
- **startTime** – a String, of type: `string` (a UTF-8 encoded string).

  Set to the UTC time at which the current server process started.
- **status** – a String, of type: `string` (a UTF-8 encoded string).

  Set to `healthy` if the instance is not experiencing problems. If the instance is recovering from a crash or from being rebooted and there are active transactions running from the latest server shutdown, status is set to `recovery`.

Errors

- UnsupportedOperationException
- InternalFailureException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- IllegalArgumentException
- TooManyRequestsException

ExecuteFastReset (action)

The AWS CLI name for this API is: `execute-fast-reset`.

The fast reset REST API lets you reset a Neptune graph quicky and easily, removing all of its data.

Neptune fast reset is a two-step process. First you call `ExecuteFastReset` with `action` set to `initiateDatabaseReset`. This returns a UUID token which you then include when calling `ExecuteFastReset` again with `action` set to `performDatabaseReset`. See Empty an Amazon Neptune DB cluster using the fast reset API.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:ResetDatabase IAM action in that cluster.

Request

- **action** (in the CLI: `--action`) – *Required:* an Action, of type: `string` (a UTF-8 encoded string).

  The fast reset action. One of the following values:

  - **`initiateDatabaseReset`** – This action generates a unique token needed to actually perform the fast reset.
  - **`performDatabaseReset`** – This action uses the token generated by the `initiateDatabaseReset` action to actually perform the fast reset.

- **token** (in the CLI: `--token`) – a String, of type: `string` (a UTF-8 encoded string).

  The fast-reset token to initiate the reset.

Response

- **payload** – A FastResetToken object.

  The `payload` is only returned by the `initiateDatabaseReset` action, and contains the unique token to use with the `performDatabaseReset` action to make the reset occur.

- **status** – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The `status` is only returned for the `performDatabaseReset` action, and indicates whether or not the fast reset rquest is accepted.

Errors

- InvalidParameterException
- ClientTimeoutException

- AccessDeniedException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- ServerShutdownException
- PreconditionsFailedException
- MethodNotAllowedException
- ReadOnlyViolationException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException

Engine operation structures:

QueryLanguageVersion (structure)

Structure for expressing the query language version.

Fields

- **version** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The version of the query language.

FastResetToken (structure)

A structure containing the fast reset token used to initiate a fast reset.

Fields

- **token** – This is a String, of type: `string` (a UTF-8 encoded string).

  A UUID generated by the database in the `initiateDatabaseReset` action, and then consumed by the `performDatabaseReset` to reset the database.

Neptune Query APIs

Gremlin query actions:

- ExecuteGremlinQuery (action)
- ExecuteGremlinExplainQuery (action)
- ExecuteGremlinProfileQuery (action)
- ListGremlinQueries (action)
- GetGremlinQueryStatus (action)
- CancelGremlinQuery (action)

openCypher query actions:

- ExecuteOpenCypherQuery (action)
- ExecuteOpenCypherExplainQuery (action)
- ListOpenCypherQueries (action)
- GetOpenCypherQueryStatus (action)
- CancelOpenCypherQuery (action)

Query structures:

- QueryEvalStats (structure)
- GremlinQueryStatus (structure)
- GremlinQueryStatusAttributes (structure)

ExecuteGremlinQuery (action)

The AWS CLI name for this API is: `execute-gremlin-query`.

This commands executes a Gremlin query. Amazon Neptune is compatible with Apache TinkerPop3 and Gremlin, so you can use the Gremlin traversal language to query the graph, as described under The Graph in the Apache TinkerPop3 documentation. More details can also be found in Accessing a Neptune graph with Gremlin.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that enables one of the following IAM actions in that cluster, depending on the query:

- neptune-db:ReadDataViaQuery
- neptune-db:WriteDataViaQuery
- neptune-db:DeleteDataViaQuery

Note that the neptune-db:QueryLanguage:Gremlin IAM condition key can be used in the policy document to restrict the use of Gremlin queries (see Condition keys available in Neptune IAM data-access policy statements).

Request

- **gremlinQuery** (in the CLI: `--gremlin-query`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  Using this API, you can run Gremlin queries in string format much as you can using the HTTP endpoint. The interface is compatible with whatever Gremlin version your DB cluster is using (see the Tinkerpop client section to determine which Gremlin releases your engine version supports).
- **serializer** (in the CLI: `--serializer`) – a String, of type: `string` (a UTF-8 encoded string).

  If non-null, the query results are returned in a serialized response message in the format specified by this parameter. See the GraphSON section in the TinkerPop documentation for a list of the formats that are currently supported.

Response

- **meta** – a Document, of type: `document` (a protocol-agnostic open content represented by a JSON-like data model).

  Metadata about the Gremlin query.
- **requestId** – a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier of the Gremlin query.
- **result** – a Document, of type: `document` (a protocol-agnostic open content represented by a JSON-like data model).

The Gremlin query output from the server.

- **status** – A GremlinQueryStatusAttributes object.

  The status of the Gremlin query.

Errors

- QueryTooLargeException
- BadRequestException
- QueryLimitExceededException
- InvalidParameterException
- QueryLimitException
- ClientTimeoutException
- CancelledByUserException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- FailureByQueryException
- MemoryLimitExceededException
- PreconditionsFailedException
- MalformedQueryException
- ParsingException
- ConstraintViolationException
- TimeLimitExceededException
- InvalidArgumentException
- MissingParameterException
- ConcurrentModificationException

ExecuteGremlinExplainQuery (action)

The AWS CLI name for this API is: `execute-gremlin-explain-query`.

Executes a Gremlin Explain query.

Amazon Neptune has added a Gremlin feature named `explain` that provides is a self-service tool for understanding the execution approach being taken by the Neptune engine for the query. You invoke it by adding an `explain` parameter to an HTTP call that submits a Gremlin query.

The explain feature provides information about the logical structure of query execution plans. You can use this information to identify potential evaluation and execution bottlenecks and to tune your query, as explained in Tuning Gremlin queries. You can also use query hints to improve query execution plans.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows one of the following IAM actions in that cluster, depending on the query:

- neptune-db:ReadDataViaQuery
- neptune-db:WriteDataViaQuery
- neptune-db:DeleteDataViaQuery

Note that the neptune-db:QueryLanguage:Gremlin IAM condition key can be used in the policy document to restrict the use of Gremlin queries (see Condition keys available in Neptune IAM data-access policy statements).

Request

- **gremlinQuery** (in the CLI: `--gremlin-query`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The Gremlin explain query string.

Response

- **output** – a ReportAsText, of type: `blob` (a block of uninterpreted binary data).

  A text blob containing the Gremlin explain result, as described in Tuning Gremlin queries.

Errors

- QueryTooLargeException

- BadRequestException
- QueryLimitExceededException
- InvalidParameterException
- QueryLimitException
- ClientTimeoutException
- CancelledByUserException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- FailureByQueryException
- MemoryLimitExceededException
- PreconditionsFailedException
- MalformedQueryException
- ParsingException
- ConstraintViolationException
- TimeLimitExceededException
- InvalidArgumentException
- MissingParameterException
- ConcurrentModificationException

ExecuteGremlinProfileQuery (action)

The AWS CLI name for this API is: `execute-gremlin-profile-query`.

Executes a Gremlin Profile query, which runs a specified traversal, collects various metrics about the run, and produces a profile report as output. See Gremlin profile API in Neptune for details.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:ReadDataViaQuery IAM action in that cluster.

Note that the neptune-db:QueryLanguage:Gremlin IAM condition key can be used in the policy document to restrict the use of Gremlin queries (see Condition keys available in Neptune IAM data-access policy statements).

Request

- **chop** (in the CLI: `--chop`) – an Integer, of type: `integer` (a signed 32-bit integer).

  If non-zero, causes the results string to be truncated at that number of characters. If set to zero, the string contains all the results.
- **gremlinQuery** (in the CLI: `--gremlin-query`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The Gremlin query string to profile.
- **indexOps** (in the CLI: `--index-ops`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  If this flag is set to `TRUE`, the results include a detailed report of all index operations that took place during query execution and serialization.
- **results** (in the CLI: `--results`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  If this flag is set to `TRUE`, the query results are gathered and displayed as part of the profile report. If `FALSE`, only the result count is displayed.
- **serializer** (in the CLI: `--serializer`) – a String, of type: `string` (a UTF-8 encoded string).

  If non-null, the gathered results are returned in a serialized response message in the format specified by this parameter. See Gremlin profile API in Neptune for more information.

Response

- **output** – a ReportAsText, of type: `blob` (a block of uninterpreted binary data).

  A text blob containing the Gremlin Profile result. See Gremlin profile API in Neptune for details.

Errors

- QueryTooLargeException
- BadRequestException
- QueryLimitExceededException
- InvalidParameterException
- QueryLimitException

- ClientTimeoutException
- CancelledByUserException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- FailureByQueryException
- MemoryLimitExceededException
- PreconditionsFailedException
- MalformedQueryException
- ParsingException
- ConstraintViolationException
- TimeLimitExceededException
- InvalidArgumentException
- MissingParameterException
- ConcurrentModificationException

ListGremlinQueries (action)

The AWS CLI name for this API is: `list-gremlin-queries`.

Lists active Gremlin queries. See Gremlin query status API for details about the output.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:GetQueryStatus IAM action in that cluster.

Note that the neptune-db:QueryLanguage:Gremlin IAM condition key can be used in the policy document to restrict the use of Gremlin queries (see Condition keys available in Neptune IAM data-access policy statements).

Request

- **includeWaiting** (in the CLI: `--include-waiting`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

If set to TRUE, the list returned includes waiting queries. The default is FALSE;

Response

- **acceptedQueryCount** – an Integer, of type: `integer` (a signed 32-bit integer).

  The number of queries that have been accepted but not yet completed, including queries in the queue.
- **queries** – An array of GremlinQueryStatus objects.

  A list of the current queries.
- **runningQueryCount** – an Integer, of type: `integer` (a signed 32-bit integer).

  The number of Gremlin queries currently running.

Errors

- BadRequestException
- InvalidParameterException
- ClientTimeoutException
- AccessDeniedException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- FailureByQueryException
- PreconditionsFailedException
- ParsingException
- ReadOnlyViolationException
- ConstraintViolationException
- TimeLimitExceededException
- InvalidArgumentException
- MissingParameterException

- ConcurrentModificationException

GetGremlinQueryStatus (action)

The AWS CLI name for this API is: `get-gremlin-query-status`.

Gets the status of a specified Gremlin query.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:GetQueryStatus IAM action in that cluster.

Note that the neptune-db:QueryLanguage:Gremlin IAM condition key can be used in the policy document to restrict the use of Gremlin queries (see Condition keys available in Neptune IAM data-access policy statements).

Request

- **queryId** (in the CLI: `--query-id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier that identifies the Gremlin query.

Response

- **queryEvalStats** – A QueryEvalStats object.

  The evaluation status of the Gremlin query.
- **queryId** – a String, of type: `string` (a UTF-8 encoded string).

  The ID of the query for which status is being returned.
- **queryString** – a String, of type: `string` (a UTF-8 encoded string).

  The Gremlin query string.

Errors

- BadRequestException
- InvalidParameterException

- ClientTimeoutException
- AccessDeniedException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- FailureByQueryException
- PreconditionsFailedException
- ParsingException
- ReadOnlyViolationException
- ConstraintViolationException
- TimeLimitExceededException
- InvalidArgumentException
- MissingParameterException
- ConcurrentModificationException

CancelGremlinQuery (action)

The AWS CLI name for this API is: `cancel-gremlin-query`.

Cancels a Gremlin query. See Gremlin query cancellation for more information.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:CancelQuery IAM action in that cluster.

Request

- **queryId** (in the CLI: `--query-id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier that identifies the query to be canceled.

Response

- **status** – a String, of type: `string` (a UTF-8 encoded string).

The status of the cancelation

Errors

- BadRequestException
- InvalidParameterException
- ClientTimeoutException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- FailureByQueryException
- PreconditionsFailedException
- ParsingException
- ConstraintViolationException
- TimeLimitExceededException
- InvalidArgumentException
- MissingParameterException
- ConcurrentModificationException

openCypher query actions:

ExecuteOpenCypherQuery (action)

The AWS CLI name for this API is: `execute-open-cypher-query`.

Executes an openCypher query. See Accessing the Neptune Graph with openCypher for more information.

Neptune supports building graph applications using openCypher, which is currently one of the most popular query languages among developers working with graph databases. Developers, business analysts, and data scientists like openCypher's declarative, SQL-inspired syntax because it provides a familiar structure in which to querying property graphs.

The openCypher language was originally developed by Neo4j, then open-sourced in 2015 and contributed to the openCypher project under an Apache 2 open-source license.

Note that when invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows one of the following IAM actions in that cluster, depending on the query:

- neptune-db:ReadDataViaQuery
- neptune-db:WriteDataViaQuery
- neptune-db:DeleteDataViaQuery

Note also that the neptune-db:QueryLanguage:OpenCypher IAM condition key can be used in the policy document to restrict the use of openCypher queries (see Condition keys available in Neptune IAM data-access policy statements).

Request

- **openCypherQuery** (in the CLI: `--open-cypher-query`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The openCypher query string to be executed.
- **parameters** (in the CLI: `--parameters`) – a String, of type: `string` (a UTF-8 encoded string).

  The openCypher query parameters for query execution. See Examples of openCypher parameterized queries for more information.

Response

- **results** – *Required:* a Document, of type: `document` (a protocol-agnostic open content represented by a JSON-like data model).

  The openCypherquery results.

Errors

- QueryTooLargeException
- InvalidNumericDataException
- BadRequestException

- QueryLimitExceededException
- InvalidParameterException
- QueryLimitException
- ClientTimeoutException
- CancelledByUserException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- FailureByQueryException
- MemoryLimitExceededException
- PreconditionsFailedException
- MalformedQueryException
- ParsingException
- ConstraintViolationException
- TimeLimitExceededException
- InvalidArgumentException
- MissingParameterException
- ConcurrentModificationException

ExecuteOpenCypherExplainQuery (action)

The AWS CLI name for this API is: `execute-open-cypher-explain-query`.

Executes an openCypher `explain` request. See The openCypher explain feature for more information.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:ReadDataViaQuery IAM action in that cluster.

Note that the neptune-db:QueryLanguage:OpenCypher IAM condition key can be used in the policy document to restrict the use of openCypher queries (see Condition keys available in Neptune IAM data-access policy statements).

Request

- **explainMode** (in the CLI: `--explain-mode`) – *Required:* an OpenCypherExplainMode, of type: `string` (a UTF-8 encoded string).

  The openCypher `explain` mode. Can be one of: `static`, `dynamic`, or `details`.
- **openCypherQuery** (in the CLI: `--open-cypher-query`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The openCypher query string.
- **parameters** (in the CLI: `--parameters`) – a String, of type: `string` (a UTF-8 encoded string).

  The openCypher query parameters.

Response

- **results** – *Required:* a Blob, of type: `blob` (a block of uninterpreted binary data).

  A text blob containing the openCypher `explain` results.

Errors

- QueryTooLargeException
- InvalidNumericDataException
- BadRequestException
- QueryLimitExceededException
- InvalidParameterException
- QueryLimitException
- ClientTimeoutException
- CancelledByUserException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- FailureByQueryException
- MemoryLimitExceededException

- PreconditionsFailedException
- MalformedQueryException
- ParsingException
- ConstraintViolationException
- TimeLimitExceededException
- InvalidArgumentException
- MissingParameterException
- ConcurrentModificationException

ListOpenCypherQueries (action)

The AWS CLI name for this API is: `list-open-cypher-queries`.

Lists active openCypher queries. See Neptune openCypher status endpoint for more information.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:GetQueryStatus IAM action in that cluster.

Note that the neptune-db:QueryLanguage:OpenCypher IAM condition key can be used in the policy document to restrict the use of openCypher queries (see Condition keys available in Neptune IAM data-access policy statements).

Request

- **includeWaiting** (in the CLI: `--include-waiting`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  When set to TRUE and other parameters are not present, causes status information to be returned for waiting queries as well as for running queries.

Response

- **acceptedQueryCount** – an Integer, of type: `integer` (a signed 32-bit integer).

  The number of queries that have been accepted but not yet completed, including queries in the queue.

- **queries** – An array of GremlinQueryStatus objects.

  A list of current openCypher queries.

- **runningQueryCount** – an Integer, of type: `integer` (a signed 32-bit integer).

  The number of currently running openCypher queries.

Errors

- InvalidNumericDataException
- BadRequestException
- InvalidParameterException
- ClientTimeoutException
- AccessDeniedException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- FailureByQueryException
- PreconditionsFailedException
- ParsingException
- ReadOnlyViolationException
- ConstraintViolationException
- TimeLimitExceededException
- InvalidArgumentException
- MissingParameterException
- ConcurrentModificationException

GetOpenCypherQueryStatus (action)

The AWS CLI name for this API is: `get-open-cypher-query-status`.

Retrieves the status of a specified openCypher query.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:GetQueryStatus IAM action in that cluster.

Note that the neptune-db:QueryLanguage:OpenCypher IAM condition key can be used in the policy document to restrict the use of openCypher queries (see Condition keys available in Neptune IAM data-access policy statements).

Request

- **queryId** (in the CLI: `--query-id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The unique ID of the openCypher query for which to retrieve the query status.

Response

- **queryEvalStats** – A QueryEvalStats object.

  The openCypher query evaluation status.
- **queryId** – a String, of type: `string` (a UTF-8 encoded string).

  The unique ID of the query for which status is being returned.
- **queryString** – a String, of type: `string` (a UTF-8 encoded string).

  The openCypher query string.

Errors

- InvalidNumericDataException
- BadRequestException
- InvalidParameterException
- ClientTimeoutException
- AccessDeniedException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException

- FailureByQueryException
- PreconditionsFailedException
- ParsingException
- ReadOnlyViolationException
- ConstraintViolationException
- TimeLimitExceededException
- InvalidArgumentException
- MissingParameterException
- ConcurrentModificationException

CancelOpenCypherQuery (action)

The AWS CLI name for this API is: `cancel-open-cypher-query`.

Cancels a specified openCypher query. See Neptune openCypher status endpoint for more information.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:CancelQuery IAM action in that cluster.

Request

- **queryId** (in the CLI: `--query-id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The unique ID of the openCypher query to cancel.
- **silent** (in the CLI: `--silent`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  If set to TRUE, causes the cancelation of the openCypher query to happen silently.

Response

- **payload** – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  The cancelation payload for the openCypher query.
- **status** – a String, of type: `string` (a UTF-8 encoded string).

The cancellation status of the openCypher query.

Errors

- InvalidNumericDataException
- BadRequestException
- InvalidParameterException
- ClientTimeoutException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- FailureByQueryException
- PreconditionsFailedException
- ParsingException
- ConstraintViolationException
- TimeLimitExceededException
- InvalidArgumentException
- MissingParameterException
- ConcurrentModificationException

Query structures:

QueryEvalStats (structure)

Structure to capture query statistics such as how many queries are running, accepted or waiting and their details.

Fields

- **cancelled** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Set to TRUE if the query was cancelled, or FALSE otherwise.
- **elapsed** – This is an Integer, of type: `integer` (a signed 32-bit integer).

The number of milliseconds the query has been running so far.

- **subqueries** – This is a Document, of type: `document` (a protocol-agnostic open content represented by a JSON-like data model).

  The number of subqueries in this query.

- **waited** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  Indicates how long the query waited, in milliseconds.

GremlinQueryStatus (structure)

Captures the status of a Gremlin query (see the Gremlin query status API page).

Fields

- **queryEvalStats** – This is A QueryEvalStats object.

  The query statistics of the Gremlin query.

- **queryId** – This is a String, of type: `string` (a UTF-8 encoded string).

  The ID of the Gremlin query.

- **queryString** – This is a String, of type: `string` (a UTF-8 encoded string).

  The query string of the Gremlin query.

GremlinQueryStatusAttributes (structure)

Contains status components of a Gremlin query.

Fields

- **attributes** – This is a Document, of type: `document` (a protocol-agnostic open content represented by a JSON-like data model).

  Attributes of the Gremlin query status.

- **code** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  The HTTP response code returned fro the Gremlin query request..

- **message** – This is a String, of type: `string` (a UTF-8 encoded string).

  The status message.

Neptune data plane bulk loader APIs

Bulk-load actions:

- StartLoaderJob (action)
- GetLoaderJobStatus (action)
- ListLoaderJobs (action)
- CancelLoaderJob (action)

Bulk load structure:

- LoaderIdResult (structure)

StartLoaderJob (action)

The AWS CLI name for this API is: `start-loader-job`.

Starts a Neptune bulk loader job to load data from an Amazon S3 bucket into a Neptune DB instance. See Using the Amazon Neptune Bulk Loader to Ingest Data.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:StartLoaderJob IAM action in that cluster.

Request

- **dependencies** (in the CLI: `--dependencies`) – a String, of type: `string` (a UTF-8 encoded string).

  This is an optional parameter that can make a queued load request contingent on the successful completion of one or more previous jobs in the queue.

  Neptune can queue up as many as 64 load requests at a time, if their `queueRequest` parameters are set to `"TRUE"`. The `dependencies` parameter lets you make execution of such a queued

request dependent on the successful completion of one or more specified previous requests in the queue.

For example, if load `Job-A` and `Job-B` are independent of each other, but load `Job-C` needs `Job-A` and `Job-B` to be finished before it begins, proceed as follows:

1. Submit `load-job-A` and `load-job-B` one after another in any order, and save their load-ids.
2. Submit `load-job-C` with the load-ids of the two jobs in its `dependencies` field:

Example

```
"dependencies" : ["(job_A_load_id)", "(job_B_load_id)"]
```

Because of the `dependencies` parameter, the bulk loader will not start `Job-C` until `Job-A` and `Job-B` have completed successfully. If either one of them fails, Job-C will not be executed, and its status will be set to `LOAD_FAILED_BECAUSE_DEPENDENCY_NOT_SATISFIED`.

You can set up multiple levels of dependency in this way, so that the failure of one job will cause all requests that are directly or indirectly dependent on it to be cancelled.

- **failOnError** (in the CLI: `--fail-on-error`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  `failOnError` – A flag to toggle a complete stop on an error.

  Allowed values: `"TRUE"`, `"FALSE"`.

  Default value: `"TRUE"`.

  When this parameter is set to `"FALSE"`, the loader tries to load all the data in the location specified, skipping any entries with errors.

  When this parameter is set to `"TRUE"`, the loader stops as soon as it encounters an error. Data loaded up to that point persists.

- **format** (in the CLI: `--format`) – *Required:* a Format, of type: `string` (a UTF-8 encoded string).

  The format of the data. For more information about data formats for the Neptune `Loader` command, see Load Data Formats.

Allowed values

- **`csv`** for the Gremlin CSV data format.
- **`opencypher`** for the openCypher CSV data format.
- **`ntriples`** for the N-Triples RDF data format.
- **`nquads`** for the N-Quads RDF data format.
- **`rdfxml`** for the RDF\XML RDF data format.
- **`turtle`** for the Turtle RDF data format.

- **iamRoleArn** (in the CLI: `--iam-role-arn`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) for an IAM role to be assumed by the Neptune DB instance for access to the S3 bucket. The IAM role ARN provided here should be attached to the DB cluster (see Adding the IAM Role to an Amazon Neptune Cluster.

- **mode** (in the CLI: `--mode`) – a Mode, of type: `string` (a UTF-8 encoded string).

  The load job mode.

  Allowed values: `RESUME`, `NEW`, `AUTO`.

  Default value: `AUTO`.

  - `RESUME` – In RESUME mode, the loader looks for a previous load from this source, and if it finds one, resumes that load job. If no previous load job is found, the loader stops.

    The loader avoids reloading files that were successfully loaded in a previous job. It only tries to process failed files. If you dropped previously loaded data from your Neptune cluster, that data is not reloaded in this mode. If a previous load job loaded all files from the same source successfully, nothing is reloaded, and the loader returns success.
  - `NEW` – In NEW mode, the creates a new load request regardless of any previous loads. You can use this mode to reload all the data from a source after dropping previously loaded data from your Neptune cluster, or to load new data available at the same source.
  - `AUTO` – In AUTO mode, the loader looks for a previous load job from the same source, and if it finds one, resumes that job, just as in `RESUME` mode.

If the loader doesn't find a previous load job from the same source, it loads all data from the source, just as in `NEW` mode.

- **parallelism** (in the CLI: `--parallelism`) – a Parallelism, of type: `string` (a UTF-8 encoded string).

  The optional `parallelism` parameter can be set to reduce the number of threads used by the bulk load process.

  Allowed values:

  - `LOW` – The number of threads used is the number of available vCPUs divided by 8.
  - `MEDIUM` – The number of threads used is the number of available vCPUs divided by 2.
  - `HIGH` – The number of threads used is the same as the number of available vCPUs.
  - `OVERSUBSCRIBE` – The number of threads used is the number of available vCPUs multiplied by 2. If this value is used, the bulk loader takes up all available resources.

    This does not mean, however, that the `OVERSUBSCRIBE` setting results in 100% CPU utilization. Because the load operation is I/O bound, the highest CPU utilization to expect is in the 60% to 70% range.

  Default value: `HIGH`

  The `parallelism` setting can sometimes result in a deadlock between threads when loading openCypher data. When this happens, Neptune returns the `LOAD_DATA_DEADLOCK` error. You can generally fix the issue by setting `parallelism` to a lower setting and retrying the load command.

- **parserConfiguration** (in the CLI: `--parser-configuration`) – It is a map array of key-value pairs where:

  Each key is a a String, of type: `string` (a UTF-8 encoded string).

  Each value is a a String, of type: `string` (a UTF-8 encoded string).

  `parserConfiguration` – An optional object with additional parser configuration values. Each of the child parameters is also optional:

  - **`namedGraphUri`** – The default graph for all RDF formats when no graph is specified (for non-quads formats and NQUAD entries with no graph).

The default is `neptune/vocab/v01/DefaultNamedGraph`.

- **`baseUri`** – The base URI for RDF/XML and Turtle formats.

  The default is `neptune/default`.

- **`allowEmptyStrings`** – Gremlin users need to be able to pass empty string values("") as node and edge properties when loading CSV data. If `allowEmptyStrings` is set to `false` (the default), such empty strings are treated as nulls and are not loaded.

  If `allowEmptyStrings` is set to `true`, the loader treats empty strings as valid property values and loads them accordingly.

- **queueRequest** (in the CLI: `--queue-request`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  This is an optional flag parameter that indicates whether the load request can be queued up or not.

  You don't have to wait for one load job to complete before issuing the next one, because Neptune can queue up as many as 64 jobs at a time, provided that their `queueRequest` parameters are all set to `"TRUE"`. The queue order of the jobs will be first-in-first-out (FIFO).

  If the `queueRequest` parameter is omitted or set to `"FALSE"`, the load request will fail if another load job is already running.

  Allowed values: `"TRUE"`, `"FALSE"`.

  Default value: `"FALSE"`.

- **s3BucketRegion** (in the CLI: `--s-3-bucket-region`) – *Required:* a S3BucketRegion, of type: `string` (a UTF-8 encoded string).

  The Amazon region of the S3 bucket. This must match the Amazon Region of the DB cluster.

- **source** (in the CLI: `--source`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The `source` parameter accepts an S3 URI that identifies a single file, multiple files, a folder, or multiple folders. Neptune loads every data file in any folder that is specified.

  The URI can be in any of the following formats.

  - `s3://(bucket_name)/(object-key-name)`
  - `https://s3.amazonaws.com/(bucket_name)/(object-key-name)`

- `https://s3.us-east-1.amazonaws.com/(bucket_name)/(object-key-name)`

  The `object-key-name` element of the URI is equivalent to the prefix parameter in an S3 ListObjects API call. It identifies all the objects in the specified S3 bucket whose names begin with that prefix. That can be a single file or folder, or multiple files and/or folders.

  The specified folder or folders can contain multiple vertex files and multiple edge files.

- **updateSingleCardinalityProperties** (in the CLI: `--update-single-cardinality-properties`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  `updateSingleCardinalityProperties` is an optional parameter that controls how the bulk loader treats a new value for single-cardinality vertex or edge properties.

  Allowed values: `"TRUE"`, `"FALSE"`.

  Default value: `"FALSE"`.

  By default, or when `updateSingleCardinalityProperties` is explicitly set to `"FALSE"`, the loader treats a new value as an error, because it violates single cardinality.

  When `updateSingleCardinalityProperties` is set to `"TRUE"`, on the other hand, the bulk loader replaces the existing value with the new one. If multiple edge or single-cardinality vertex property values are provided in the source file(s) being loaded, the final value at the end of the bulk load could be any one of those new values. The loader only guarantees that the existing value has been replaced by one of the new ones.

- **userProvidedEdgeIds** (in the CLI: `--user-provided-edge-ids`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  This parameter is required only when loading openCypher data that contains relationship IDs. It must be included and set to `True` when openCypher relationship IDs are explicitly provided in the load data (recommended).

  When `userProvidedEdgeIds` is absent or set to `True`, an `:ID` column must be present in every relationship file in the load.

  When `userProvidedEdgeIds` is present and set to `False`, relationship files in the load **must not** contain an `:ID` column. Instead, the Neptune loader automatically generates an ID for each relationship.

It's useful to provide relationship IDs explicitly so that the loader can resume loading after error in the CSV data have been fixed, without having to reload any relationships that have already been loaded. If relationship IDs have not been explicitly assigned, the loader cannot resume a failed load if any relationship file has had to be corrected, and must instead reload all the relationships.

Response

- **payload** – *Required:* It is a map array of key-value pairs where:

  Each key is a a String, of type: `string` (a UTF-8 encoded string).

  Each value is a a String, of type: `string` (a UTF-8 encoded string).

  Contains a `loadId` name-value pair that provides an identifier for the load operation.
- **status** – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP return code indicating the status of the load job.

Errors

- BadRequestException
- InvalidParameterException
- BulkLoadIdNotFoundException
- ClientTimeoutException
- LoadUrlAccessDeniedException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- InternalFailureException
- S3Exception
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException

- MissingParameterException

GetLoaderJobStatus (action)

The AWS CLI name for this API is: `get-loader-job-status`.

Gets status information about a specified load job. Neptune keeps track of the most recent 1,024 bulk load jobs, and stores the last 10,000 error details per job.

See Neptune Loader Get-Status API for more information.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:GetLoaderJobStatus IAM action in that cluster..

Request

- **details** (in the CLI: `--details`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Flag indicating whether or not to include details beyond the overall status (`TRUE` or `FALSE`; the default is `FALSE`).
- **errors** (in the CLI: `--errors`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Flag indicating whether or not to include a list of errors encountered (`TRUE` or `FALSE`; the default is `FALSE`).

  The list of errors is paged. The `page` and `errorsPerPage` parameters allow you to page through all the errors.
- **errorsPerPage** (in the CLI: `--errors-per-page`) – a PositiveInteger, of type: `integer` (a signed 32-bit integer), at least 1 ?st?.

  The number of errors returned in each page (a positive integer; the default is `10`). Only valid when the `errors` parameter set to `TRUE`.
- **loadId** (in the CLI: `--load-id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The load ID of the load job to get the status of.
- **page** (in the CLI: `--page`) – a PositiveInteger, of type: `integer` (a signed 32-bit integer), at least 1 ?st?.

The error page number (a positive integer; the default is 1). Only valid when the `errors` parameter is set to TRUE.

Response

- **payload** – *Required:* a Document, of type: `document` (a protocol-agnostic open content represented by a JSON-like data model).

  Status information about the load job, in a layout that could look like this:

  Example

```
{
          "status" : "200 OK",
          "payload" : {
            "feedCount" : [
              {
                "LOAD_FAILED" : (number)
              }
            ],
            "overallStatus" : {
              "fullUri" : "s3://(bucket)/(key)",
              "runNumber" : (number),
              "retryNumber" : (number),
              "status" : "(string)",
              "totalTimeSpent" : (number),
              "startTime" : (number),
              "totalRecords" : (number),
              "totalDuplicates" : (number),
              "parsingErrors" : (number),
              "datatypeMismatchErrors" : (number),
              "insertErrors" : (number),
            },
            "failedFeeds" : [
              {
                "fullUri" : "s3://(bucket)/(key)",
                "runNumber" : (number),
                "retryNumber" : (number),
                "status" : "(string)",
                "totalTimeSpent" : (number),
                "startTime" : (number),
```

```
                "totalRecords" : (number),
                "totalDuplicates" : (number),
                "parsingErrors" : (number),
                "datatypeMismatchErrors" : (number),
                "insertErrors" : (number),
              }
            ],
            "errors" : {
              "startIndex" : (number),
              "endIndex" : (number),
              "loadId" : "(string),
              "errorLogs" : [ ]
            }
          }
        }
```

- **status** – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP response code for the request.

Errors

- BadRequestException
- InvalidParameterException
- BulkLoadIdNotFoundException
- ClientTimeoutException
- LoadUrlAccessDeniedException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- InternalFailureException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException

ListLoaderJobs (action)

The AWS CLI name for this API is: `list-loader-jobs`.

Retrieves a list of the `loadIds` for all active loader jobs.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:ListLoaderJobs IAM action in that cluster..

Request

- **includeQueuedLoads** (in the CLI: `--include-queued-loads`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  An optional parameter that can be used to exclude the load IDs of queued load requests when requesting a list of load IDs by setting the parameter to `FALSE`. The default value is `TRUE`.
- **limit** (in the CLI: `--limit`) – a ListLoaderJobsInputLimitInteger, of type: `integer` (a signed 32-bit integer), not less than 1 or more than 100 ?st?s.

  The number of load IDs to list. Must be a positive integer greater than zero and not more than `100` (which is the default).

Response

- **payload** – *Required:* A LoaderIdResult object.

  The requested list of job IDs.
- **status** – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  Returns the status of the job list request.

Errors

- UnsupportedOperationException
- BadRequestException
- InvalidParameterException
- BulkLoadIdNotFoundException

- InternalFailureException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- LoadUrlAccessDeniedException
- IllegalArgumentException
- TooManyRequestsException

CancelLoaderJob (action)

The AWS CLI name for this API is: `cancel-loader-job`.

Cancels a specified load job. This is an HTTP `DELETE` request. See Neptune Loader Get-Status API for more information.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:CancelLoaderJob IAM action in that cluster..

Request

- **loadId** (in the CLI: `--load-id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the load job to be deleted.

Response

- **status** – a String, of type: `string` (a UTF-8 encoded string).

  The cancellation status.

Errors

- BadRequestException
- InvalidParameterException

- BulkLoadIdNotFoundException
- ClientTimeoutException
- LoadUrlAccessDeniedException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- InternalFailureException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException

Bulk load structure:

LoaderIdResult (structure)

Contains a list of load IDs.

Fields

- **loadIds** – This is a String, of type: `string` (a UTF-8 encoded string).

  A list of load IDs.

Neptune streams dataplane API

Stream access actions:

- GetPropertygraphStream (action)

Stream data structures:

- PropertygraphRecord (structure)
- PropertygraphData (structure)

GetPropertygraphStream (action)

The AWS CLI name for this API is: `get-propertygraph-stream`.

Gets a stream for a property graph.

With the Neptune Streams feature, you can generate a complete sequence of change-log entries that record every change made to your graph data as it happens. `GetPropertygraphStream` lets you collect these change-log entries for a property graph.

The Neptune streams feature needs to be enabled on your Neptune DBcluster. To enable streams, set the neptune_streams DB cluster parameter to 1.

See Capturing graph changes in real time using Neptune streams.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:GetStreamRecords IAM action in that cluster.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that enables one of the following IAM actions, depending on the query:

Note that you can restrict property-graph queries using the following IAM context keys:

- neptune-db:QueryLanguage:Gremlin
- neptune-db:QueryLanguage:OpenCypher

See Condition keys available in Neptune IAM data-access policy statements).

Request

- **commitNum** (in the CLI: `--commit-num`) – a Long, of type: `long` (a signed 64-bit integer).

  The commit number of the starting record to read from the change-log stream. This parameter is required when `iteratorType` is`AT_SEQUENCE_NUMBER` or `AFTER_SEQUENCE_NUMBER`, and ignored when `iteratorType` is `TRIM_HORIZON` or `LATEST`.
- **encoding** (in the CLI: `--encoding`) – an Encoding, of type: `string` (a UTF-8 encoded string).

  If set to TRUE, Neptune compresses the response using gzip encoding.

- **iteratorType** (in the CLI: `--iterator-type`) – an IteratorType, of type: `string` (a UTF-8 encoded string).

  Can be one of:

  - `AT_SEQUENCE_NUMBER` – Indicates that reading should start from the event sequence number specified jointly by the `commitNum` and `opNum` parameters.
  - `AFTER_SEQUENCE_NUMBER` – Indicates that reading should start right after the event sequence number specified jointly by the `commitNum` and `opNum` parameters.
  - `TRIM_HORIZON` – Indicates that reading should start at the last untrimmed record in the system, which is the oldest unexpired (not yet deleted) record in the change-log stream.
  - `LATEST` – Indicates that reading should start at the most recent record in the system, which is the latest unexpired (not yet deleted) record in the change-log stream.
- **limit** (in the CLI: `--limit`) – a GetPropertygraphStreamInputLimitLong, of type: `long` (a signed 64-bit integer), not less than 1 or more than 100000 ?st?s.

  Specifies the maximum number of records to return. There is also a size limit of 10 MB on the response that can't be modified and that takes precedence over the number of records specified in the `limit` parameter. The response does include a threshold-breaching record if the 10 MB limit was reached.

  The range for `limit` is 1 to 100,000, with a default of 10.
- **opNum** (in the CLI: `--op-num`) – a Long, of type: `long` (a signed 64-bit integer).

  The operation sequence number within the specified commit to start reading from in the change-log stream data. The default is 1.

Response

- **format** – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  Serialization format for the change records being returned. Currently, the only supported value is `PG_JSON`.
- **lastEventId** – *Required:* It is a map array of key-value pairs where:

  Each key is a a String, of type: `string` (a UTF-8 encoded string).

  Each value is a a String, of type: `string` (a UTF-8 encoded string).

Sequence identifier of the last change in the stream response.

An event ID is composed of two fields: a `commitNum`, which identifies a transaction that changed the graph, and an `opNum`, which identifies a specific operation within that transaction:

Example

```
"eventId": {
          "commitNum": 12,
          "opNum": 1
        }
```

- **lastTrxTimestampInMillis** – *Required:* a Long, of type: `long` (a signed 64-bit integer).

  The time at which the commit for the transaction was requested, in milliseconds from the Unix epoch.
- **records** – *Required:* An array of PropertygraphRecord objects.

  An array of serialized change-log stream records included in the response.
- **totalRecords** – *Required:* an Integer, of type: `integer` (a signed 32-bit integer).

  The total number of records in the response.

Errors

- UnsupportedOperationException
- ExpiredStreamException
- InvalidParameterException
- MemoryLimitExceededException
- StreamRecordsNotFoundException
- ClientTimeoutException
- PreconditionsFailedException
- ThrottlingException
- ConstraintViolationException
- InvalidArgumentException
- IllegalArgumentException

- TooManyRequestsException

Stream data structures:

PropertygraphRecord (structure)

Structure of a property graph record.

Fields

- **commitTimestampInMillis** – This is *Required:* a Long, of type: `long` (a signed 64-bit integer).

  The time at which the commit for the transaction was requested, in milliseconds from the Unix epoch.
- **data** – This is *Required:* A PropertygraphData object.

  The serialized Gremlin or openCypher change record.
- **eventId** – This is *Required:* It is a map array of key-value pairs where:

  Each key is a a String, of type: `string` (a UTF-8 encoded string).

  Each value is a a String, of type: `string` (a UTF-8 encoded string).

  The sequence identifier of the stream change record.
- **isLastOp** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Only present if this operation is the last one in its transaction. If present, it is set to true. It is useful for ensuring that an entire transaction is consumed.
- **op** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The operation that created the change.

PropertygraphData (structure)

A Gremlin or openCypher change record.

Fields

- **from** – This is a String, of type: `string` (a UTF-8 encoded string).

If this is an edge (type = e), the ID of the corresponding `from` vertex or source node.

- **id** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the Gremlin or openCypher element.

- **key** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The property name. For element labels, this is `label`.

- **to** – This is a String, of type: `string` (a UTF-8 encoded string).

  If this is an edge (type = e), the ID of the corresponding `to` vertex or target node.

- **type** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The type of this Gremlin or openCypher element. Must be one of:

  - **v1** - Vertex label for Gremlin, or node label for openCypher.
  - **vp** - Vertex properties for Gremlin, or node properties for openCypher.
  - **e** - Edge and edge label for Gremlin, or relationship and relationship type for openCypher.
  - **ep** - Edge properties for Gremlin, or relationship properties for openCypher.

- **value** – This is *Required:* a Document, of type: `document` (a protocol-agnostic open content represented by a JSON-like data model).

  This is a JSON object that contains a value field for the value itself, and a datatype field for the JSON data type of that value:

  Example

```
"value": {
          "value": "(the new value"),
          "dataType": "(the JSON datatypenew value")
        }
```

Neptune dataplane statistics and graph summary APIs

Property graph statistics actions:

- GetPropertygraphStatistics (action)
- ManagePropertygraphStatistics (action)

- DeletePropertygraphStatistics (action)
- GetPropertygraphSummary (action)

Statistics structures:

- Statistics (structure)
- StatisticsSummary (structure)
- DeleteStatisticsValueMap (structure)
- RefreshStatisticsIdMap (structure)
- NodeStructure (structure)
- EdgeStructure (structure)
- SubjectStructure (structure)
- PropertygraphSummaryValueMap (structure)
- PropertygraphSummary (structure)

GetPropertygraphStatistics (action)

The AWS CLI name for this API is: `get-propertygraph-statistics`.

Gets property graph statistics (Gremlin and openCypher).

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:GetStatisticsStatus IAM action in that cluster.

Request

- *No Request parameters.*

Response

- **payload** – *Required:* A Statistics object.

  Statistics for property-graph data.
- **status** – *Required:* a String, of type: `string` (a UTF-8 encoded string).

The HTTP return code of the request. If the request succeeded, the code is 200. See Common error codes for DFE statistics request for a list of common errors.

Errors

- BadRequestException
- InvalidParameterException
- StatisticsNotAvailableException
- ClientTimeoutException
- AccessDeniedException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- PreconditionsFailedException
- ReadOnlyViolationException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException

ManagePropertygraphStatistics (action)

The AWS CLI name for this API is: `manage-propertygraph-statistics`.

Manages the generation and use of property graph statistics.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:ManageStatistics IAM action in that cluster.

Request

- **mode** (in the CLI: `--mode`) – a StatisticsAutoGenerationMode, of type: `string` (a UTF-8 encoded string).

The statistics generation mode. One of: `DISABLE_AUTOCOMPUTE`, `ENABLE_AUTOCOMPUTE`, or `REFRESH`, the last of which manually triggers DFE statistics generation.

Response

- **payload** – A RefreshStatisticsIdMap object.

  This is only returned for refresh mode.
- **status** – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP return code of the request. If the request succeeded, the code is 200.

Errors

- BadRequestException
- InvalidParameterException
- StatisticsNotAvailableException
- ClientTimeoutException
- AccessDeniedException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- PreconditionsFailedException
- ReadOnlyViolationException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException

DeletePropertygraphStatistics (action)

The AWS CLI name for this API is: `delete-propertygraph-statistics`.

Deletes statistics for Gremlin and openCypher (property graph) data.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:DeleteStatistics IAM action in that cluster.

Request

- *No Request parameters.*

Response

- **payload** – A DeleteStatisticsValueMap object.

  The deletion payload.
- **status** – a String, of type: `string` (a UTF-8 encoded string).

  The cancel status.
- **statusCode** – an Integer, of type: `integer` (a signed 32-bit integer).

  The HTTP response code: 200 if the delete was successful, or 204 if there were no statistics to delete.

Errors

- BadRequestException
- InvalidParameterException
- StatisticsNotAvailableException
- ClientTimeoutException
- AccessDeniedException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException
- PreconditionsFailedException
- ReadOnlyViolationException
- ConstraintViolationException
- InvalidArgumentException

- MissingParameterException

GetPropertygraphSummary (action)

The AWS CLI name for this API is: `get-propertygraph-summary`.

Gets a graph summary for a property graph.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:GetGraphSummary IAM action in that cluster.

Request

- **mode** (in the CLI: `--mode`) – a GraphSummaryType, of type: `string` (a UTF-8 encoded string).

  Mode can take one of two values: `BASIC` (the default), and `DETAILED`.

Response

- **payload** – A PropertygraphSummaryValueMap object.

  Payload containing the property graph summary response.
- **statusCode** – an Integer, of type: `integer` (a signed 32-bit integer).

  The HTTP return code of the request. If the request succeeded, the code is 200.

Errors

- BadRequestException
- InvalidParameterException
- StatisticsNotAvailableException
- ClientTimeoutException
- AccessDeniedException
- IllegalArgumentException
- TooManyRequestsException
- UnsupportedOperationException

- PreconditionsFailedException
- ReadOnlyViolationException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException

Statistics structures:

Statistics (structure)

Contains statistics information. The DFE engine uses information about the data in your Neptune graph to make effective trade-offs when planning query execution. This information takes the form of statistics that include so-called characteristic sets and predicate statistics that can guide query planning. See Managing statistics for the Neptune DFE to use.

Fields

- **active** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not DFE statistics generation is enabled at all.
- **autoCompute** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Indicates whether or not automatic statistics generation is enabled.
- **date** – This is a SyntheticTimestamp_date_time, of type: `string` (a UTF-8 encoded string).

  The UTC time at which DFE statistics have most recently been generated.
- **note** – This is a String, of type: `string` (a UTF-8 encoded string).

  A note about problems in the case where statistics are invalid.
- **signatureInfo** – This is A StatisticsSummary object.

  A StatisticsSummary structure that contains:
  - `signatureCount` - The total number of signatures across all characteristic sets.
  - `instanceCount` - The total number of characteristic-set instances.
  - `predicateCount` - The total number of unique predicates.
- **statisticsId** – This is a String, of type: `string` (a UTF-8 encoded string).

Reports the ID of the current statistics generation run. A value of -1 indicates that no statistics have been generated.

StatisticsSummary (structure)

Information about the characteristic sets generated in the statistics.

Fields

- **instanceCount** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  The total number of characteristic-set instances.
- **predicateCount** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  The total number of unique predicates.
- **signatureCount** – This is an Integer, of type: `integer` (a signed 32-bit integer).

  The total number of signatures across all characteristic sets.

DeleteStatisticsValueMap (structure)

The payload for DeleteStatistics.

Fields

- **active** – This is a Boolean, of type: `boolean` (a Boolean (true or false) value).

  The current status of the statistics.
- **statisticsId** – This is a String, of type: `string` (a UTF-8 encoded string).

  The ID of the statistics generation run that is currently occurring.

RefreshStatisticsIdMap (structure)

Statistics for `REFRESH` mode.

Fields

- **statisticsId** – This is a String, of type: `string` (a UTF-8 encoded string).

The ID of the statistics generation run that is currently occurring.

NodeStructure (structure)

A node structure.

Fields

- **count** – This is a Long, of type: `long` (a signed 64-bit integer).

  Number of nodes that have this specific structure.
- **distinctOutgoingEdgeLabels** – This is a String, of type: `string` (a UTF-8 encoded string).

  A list of distinct outgoing edge labels present in this specific structure.
- **nodeProperties** – This is a String, of type: `string` (a UTF-8 encoded string).

  A list of the node properties present in this specific structure.

EdgeStructure (structure)

An edge structure.

Fields

- **count** – This is a Long, of type: `long` (a signed 64-bit integer).

  The number of edges that have this specific structure.
- **edgeProperties** – This is a String, of type: `string` (a UTF-8 encoded string).

  A list of edge properties present in this specific structure.

SubjectStructure (structure)

A subject structure.

Fields

- **count** – This is a Long, of type: `long` (a signed 64-bit integer).

Number of occurrences of this specific structure.

- **predicates** – This is a String, of type: `string` (a UTF-8 encoded string).

  A list of predicates present in this specific structure.

PropertygraphSummaryValueMap (structure)

Payload for the property graph summary response.

Fields

- **graphSummary** – This is A PropertygraphSummary object.

  The graph summary.

- **lastStatisticsComputationTime** – This is a SyntheticTimestamp_date_time, of type: `string` (a UTF-8 encoded string).

  The timestamp, in ISO 8601 format, of the time at which Neptune last computed statistics.

- **version** – This is a String, of type: `string` (a UTF-8 encoded string).

  The version of this graph summary response.

PropertygraphSummary (structure)

The graph summary API returns a read-only list of node and edge labels and property keys, along with counts of nodes, edges, and properties. See Graph summary response for a property graph (PG).

Fields

- **edgeLabels** – This is a String, of type: `string` (a UTF-8 encoded string).

  A list of the distinct edge labels in the graph.

- **edgeProperties** – This is LongValuedMap objects It is a map array of key-value pairs where:

  Each key is a a String, of type: `string` (a UTF-8 encoded string).

  Each value is a a Long, of type: `long` (a signed 64-bit integer).

A list of the distinct edge properties in the graph, along with the count of edges where each property is used.

- **edgeStructures** – This is An array of EdgeStructure objects.

  This field is only present when the requested mode is `DETAILED`. It contains a list of edge structures.

- **nodeLabels** – This is a String, of type: `string` (a UTF-8 encoded string).

  A list of the distinct node labels in the graph.

- **nodeProperties** – This is LongValuedMap objects It is a map array of key-value pairs where:

  Each key is a a String, of type: `string` (a UTF-8 encoded string).

  Each value is a a Long, of type: `long` (a signed 64-bit integer).

  The number of distinct node properties in the graph.

- **nodeStructures** – This is An array of NodeStructure objects.

  This field is only present when the requested mode is `DETAILED`. It contains a list of node structures.

- **numEdgeLabels** – This is a Long, of type: `long` (a signed 64-bit integer).

  The number of distinct edge labels in the graph.

- **numEdgeProperties** – This is a Long, of type: `long` (a signed 64-bit integer).

  The number of distinct edge properties in the graph.

- **numEdges** – This is a Long, of type: `long` (a signed 64-bit integer).

  The number of edges in the graph.

- **numNodeLabels** – This is a Long, of type: `long` (a signed 64-bit integer).

  The number of distinct node labels in the graph.

- **numNodeProperties** – This is a Long, of type: `long` (a signed 64-bit integer).

  A list of the distinct node properties in the graph, along with the count of nodes where each property is used.

- **numNodes** – This is a Long, of type: `long` (a signed 64-bit integer).

The number of nodes in the graph.

- **totalEdgePropertyValues** – This is a Long, of type: `long` (a signed 64-bit integer).

  The total number of usages of all edge properties.

- **totalNodePropertyValues** – This is a Long, of type: `long` (a signed 64-bit integer).

  The total number of usages of all node properties.

Neptune ML data-processing API

Data-processing actions:

- StartMLDataProcessingJob (action)
- ListMLDataProcessingJobs (action)
- GetMLDataProcessingJob (action)
- CancelMLDataProcessingJob (action)

ML general-purpose structures:

- MlResourceDefinition (structure)
- MlConfigDefinition (structure)

StartMLDataProcessingJob (action)

The AWS CLI name for this API is: `start-ml-data-processing-job`.

Creates a new Neptune ML data processing job for processing the graph data exported from Neptune for training. See The `dataprocessing` command.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:StartMLModelDataProcessingJob IAM action in that cluster.

Request

- **configFileName** (in the CLI: `--config-file-name`) – a String, of type: `string` (a UTF-8 encoded string).

A data specification file that describes how to load the exported graph data for training. The file is automatically generated by the Neptune export toolkit. The default is `training-data-configuration.json`.

- **id** (in the CLI: `--id`) – a String, of type: `string` (a UTF-8 encoded string).

  A unique identifier for the new job. The default is an autogenerated UUID.

- **inputDataS3Location** (in the CLI: `--input-data-s3-location`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The URI of the Amazon S3 location where you want SageMaker to download the data needed to run the data processing job.

- **modelType** (in the CLI: `--model-type`) – a String, of type: `string` (a UTF-8 encoded string).

  One of the two model types that Neptune ML currently supports: heterogeneous graph models (`heterogeneous`), and knowledge graph (`kge`). The default is none. If not specified, Neptune ML chooses the model type automatically based on the data.

- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Resource Name (ARN) of an IAM role that SageMaker can assume to perform tasks on your behalf. This must be listed in your DB cluster parameter group or an error will occur.

- **previousDataProcessingJobId** (in the CLI: `--previous-data-processing-job-id`) – a String, of type: `string` (a UTF-8 encoded string).

  The job ID of a completed data processing job run on an earlier version of the data.

- **processedDataS3Location** (in the CLI: `--processed-data-s3-location`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The URI of the Amazon S3 location where you want SageMaker to save the results of a data processing job.

- **processingInstanceType** (in the CLI: `--processing-instance-type`) – a String, of type: `string` (a UTF-8 encoded string).

  The type of ML instance used during data processing. Its memory should be large enough to hold the processed dataset. The default is the smallest ml.r5 type whose memory is ten times larger than the size of the exported graph data on disk.

- **processingInstanceVolumeSizeInGB** (in the CLI: `--processing-instance-volume-size-in-gb`) – an Integer, of type: `integer` (a signed 32-bit integer).

  The disk volume size of the processing instance. Both input data and processed data are stored on disk, so the volume size must be large enough to hold both data sets. The default is 0. If not specified or 0, Neptune ML chooses the volume size automatically based on the data size.
- **processingTimeOutInSeconds** (in the CLI: `--processing-time-out-in-seconds`) – an Integer, of type: `integer` (a signed 32-bit integer).

  Timeout in seconds for the data processing job. The default is 86,400 (1 day).
- **s3OutputEncryptionKMSKey** (in the CLI: `--s-3-output-encryption-kms-key`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Key Management Service (Amazon KMS) key that SageMaker uses to encrypt the output of the processing job. The default is none.
- **sagemakerIamRoleArn** (in the CLI: `--sagemaker-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role for SageMaker execution. This must be listed in your DB cluster parameter group or an error will occur.
- **securityGroupIds** (in the CLI: `--security-group-ids`) – a String, of type: `string` (a UTF-8 encoded string).

  The VPC security group IDs. The default is None.
- **subnets** (in the CLI: `--subnets`) – a String, of type: `string` (a UTF-8 encoded string).

  The IDs of the subnets in the Neptune VPC. The default is None.
- **volumeEncryptionKMSKey** (in the CLI: `--volume-encryption-kms-key`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Key Management Service (Amazon KMS) key that SageMaker uses to encrypt data on the storage volume attached to the ML compute instances that run the training job. The default is None.

Response

- **arn** – a String, of type: `string` (a UTF-8 encoded string).

The ARN of the data processing job.

- **creationTimeInMillis** – a Long, of type: `long` (a signed 64-bit integer).

  The time it took to create the new processing job, in milliseconds.

- **id** – a String, of type: `string` (a UTF-8 encoded string).

  The unique ID of the new data processing job.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

ListMLDataProcessingJobs (action)

The AWS CLI name for this API is: `list-ml-data-processing-jobs`.

Returns a list of Neptune ML data processing jobs. See Listing active data-processing jobs using the Neptune ML dataprocessing command.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:ListMLDataProcessingJobs IAM action in that cluster.

Request

- **maxItems** (in the CLI: `--max-items`) – a ListMLDataProcessingJobsInputMaxItemsInteger, of type: `integer` (a signed 32-bit integer), not less than 1 or more than 1024 ?st?s.

  The maximum number of items to return (from 1 to 1024; the default is 10).
- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.

Response

- **ids** – a String, of type: `string` (a UTF-8 encoded string).

  A page listing data processing job IDs.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

GetMLDataProcessingJob (action)

The AWS CLI name for this API is: `get-ml-data-processing-job`.

Retrieves information about a specified data processing job. See The `dataprocessing` command.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:neptune-db:GetMLDataProcessingJobStatus IAM action in that cluster.

Request

- **id** (in the CLI: `--id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier of the data-processing job to be retrieved.
- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.

Response

- **id** – a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier of this data-processing job.
- **processingJob** – A MlResourceDefinition object.

  Definition of the data processing job.
- **status** – a String, of type: `string` (a UTF-8 encoded string).

  Status of the data processing job.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException

- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

CancelMLDataProcessingJob (action)

The AWS CLI name for this API is: `cancel-ml-data-processing-job`.

Cancels a Neptune ML data processing job. See The `dataprocessing` command.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:CancelMLDataProcessingJob IAM action in that cluster.

Request

- **clean** (in the CLI: `--clean`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  If set to TRUE, this flag specifies that all Neptune ML S3 artifacts should be deleted when the job is stopped. The default is FALSE.
- **id** (in the CLI: `--id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier of the data-processing job.
- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.

Response

- **status** – a String, of type: `string` (a UTF-8 encoded string).

  The status of the cancellation request.

Errors

- UnsupportedOperationException

- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

ML general-purpose structures:

MlResourceDefinition (structure)

Defines a Neptune ML resource.

Fields

- **arn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The resource ARN.
- **cloudwatchLogUrl** – This is a String, of type: `string` (a UTF-8 encoded string).

  The CloudWatch log URL for the resource.
- **failureReason** – This is a String, of type: `string` (a UTF-8 encoded string).

  The failure reason, in case of a failure.
- **name** – This is a String, of type: `string` (a UTF-8 encoded string).

  The resource name.
- **outputLocation** – This is a String, of type: `string` (a UTF-8 encoded string).

  The output location.
- **status** – This is a String, of type: `string` (a UTF-8 encoded string).

The resource status.

MlConfigDefinition (structure)

Contains a Neptune ML configuration.

Fields

- **arn** – This is a String, of type: `string` (a UTF-8 encoded string).

  The ARN for the configuration.
- **name** – This is a String, of type: `string` (a UTF-8 encoded string).

  The configuration name.

Neptune ML model training API

Model training actions:

- StartMLModelTrainingJob (action)
- ListMLModelTrainingJobs (action)
- GetMLModelTrainingJob (action)
- CancelMLModelTrainingJob (action)

Model training structures:

- CustomModelTrainingParameters (structure)

StartMLModelTrainingJob (action)

> The AWS CLI name for this API is: `start-ml-model-training-job`.

Creates a new Neptune ML model training job. See Model training using the `modeltraining` command.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:StartMLModelTrainingJob IAM action in that cluster.

Request

- **baseProcessingInstanceType** (in the CLI: `--base-processing-instance-type`) – a String, of type: `string` (a UTF-8 encoded string).

  The type of ML instance used in preparing and managing training of ML models. This is a CPU instance chosen based on memory requirements for processing the training data and model.
- **customModelTrainingParameters** (in the CLI: `--custom-model-training-parameters`) – A CustomModelTrainingParameters object.

  The configuration for custom model training. This is a JSON object.
- **dataProcessingJobId** (in the CLI: `--data-processing-job-id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The job ID of the completed data-processing job that has created the data that the training will work with.
- **enableManagedSpotTraining** (in the CLI: `--enable-managed-spot-training`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  Optimizes the cost of training machine-learning models by using Amazon Elastic Compute Cloud spot instances. The default is `False`.
- **id** (in the CLI: `--id`) – a String, of type: `string` (a UTF-8 encoded string).

  A unique identifier for the new job. The default is An autogenerated UUID.
- **maxHPONumberOfTrainingJobs** (in the CLI: `--max-hpo-number-of-training-jobs`) – an Integer, of type: `integer` (a signed 32-bit integer).

  Maximum total number of training jobs to start for the hyperparameter tuning job. The default is 2. Neptune ML automatically tunes the hyperparameters of the machine learning model. To obtain a model that performs well, use at least 10 jobs (in other words, set `maxHPONumberOfTrainingJobs` to 10). In general, the more tuning runs, the better the results.
- **maxHPOParallelTrainingJobs** (in the CLI: `--max-hpo-parallel-training-jobs`) – an Integer, of type: `integer` (a signed 32-bit integer).

Maximum number of parallel training jobs to start for the hyperparameter tuning job. The default is 2. The number of parallel jobs you can run is limited by the available resources on your training instance.

- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.

- **previousModelTrainingJobId** (in the CLI: `--previous-model-training-job-id`) – a String, of type: `string` (a UTF-8 encoded string).

  The job ID of a completed model-training job that you want to update incrementally based on updated data.

- **s3OutputEncryptionKMSKey** (in the CLI: `--s-3-output-encryption-kms-key`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Key Management Service (KMS) key that SageMaker uses to encrypt the output of the processing job. The default is none.

- **sagemakerIamRoleArn** (in the CLI: `--sagemaker-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role for SageMaker execution.This must be listed in your DB cluster parameter group or an error will occur.

- **securityGroupIds** (in the CLI: `--security-group-ids`) – a String, of type: `string` (a UTF-8 encoded string).

  The VPC security group IDs. The default is None.

- **subnets** (in the CLI: `--subnets`) – a String, of type: `string` (a UTF-8 encoded string).

  The IDs of the subnets in the Neptune VPC. The default is None.

- **trainingInstanceType** (in the CLI: `--training-instance-type`) – a String, of type: `string` (a UTF-8 encoded string).

  The type of ML instance used for model training. All Neptune ML models support CPU, GPU, and multiGPU training. The default is `ml.p3.2xlarge`. Choosing the right instance type for training depends on the task type, graph size, and your budget.

- **trainingInstanceVolumeSizeInGB** (in the CLI: `--training-instance-volume-size-in-gb`) – an Integer, of type: `integer` (a signed 32-bit integer).

  The disk volume size of the training instance. Both input data and the output model are stored on disk, so the volume size must be large enough to hold both data sets. The default is 0. If not specified or 0, Neptune ML selects a disk volume size based on the recommendation generated in the data processing step.
- **trainingTimeOutInSeconds** (in the CLI: `--training-time-out-in-seconds`) – an Integer, of type: `integer` (a signed 32-bit integer).

  Timeout in seconds for the training job. The default is 86,400 (1 day).
- **trainModelS3Location** (in the CLI: `--train-model-s3-location`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The location in Amazon S3 where the model artifacts are to be stored.
- **volumeEncryptionKMSKey** (in the CLI: `--volume-encryption-kms-key`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Key Management Service (KMS) key that SageMaker uses to encrypt data on the storage volume attached to the ML compute instances that run the training job. The default is None.

Response

- **arn** – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of the new model training job.
- **creationTimeInMillis** – a Long, of type: `long` (a signed 64-bit integer).

  The model training job creation time, in milliseconds.
- **id** – a String, of type: `string` (a UTF-8 encoded string).

  The unique ID of the new model training job.

Errors

- UnsupportedOperationException

- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

ListMLModelTrainingJobs (action)

The AWS CLI name for this API is: `list-ml-model-training-jobs`.

Lists Neptune ML model-training jobs. See Model training using the `modeltraining` command.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:neptune-db:ListMLModelTrainingJobs IAM action in that cluster.

Request

- **maxItems** (in the CLI: `--max-items`) – a ListMLModelTrainingJobsInputMaxItemsInteger, of type: `integer` (a signed 32-bit integer), not less than 1 or more than 1024 ?st?s.

  The maximum number of items to return (from 1 to 1024; the default is 10).
- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.

Response

- **ids** – a String, of type: `string` (a UTF-8 encoded string).

A page of the list of model training job IDs.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

GetMLModelTrainingJob (action)

The AWS CLI name for this API is: `get-ml-model-training-job`.

Retrieves information about a Neptune ML model training job. See Model training using the `modeltraining` command.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:GetMLModelTrainingJobStatus IAM action in that cluster.

Request

- **id** (in the CLI: `--id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier of the model-training job to retrieve.
- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.

Response

- **hpoJob** – A MlResourceDefinition object.

  The HPO job.
- **id** – a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier of this model-training job.
- **mlModels** – An array of MlConfigDefinition objects.

  A list of the configurations of the ML models being used.
- **modelTransformJob** – A MlResourceDefinition object.

  The model transform job.
- **processingJob** – A MlResourceDefinition object.

  The data processing job.
- **status** – a String, of type: `string` (a UTF-8 encoded string).

  The status of the model training job.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException

- IllegalArgumentException
- TooManyRequestsException

CancelMLModelTrainingJob (action)

The AWS CLI name for this API is: `cancel-ml-model-training-job`.

Cancels a Neptune ML model training job. See Model training using the `modeltraining` command.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:CancelMLModelTrainingJob IAM action in that cluster.

Request

- **clean** (in the CLI: `--clean`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  If set to `TRUE`, this flag specifies that all Amazon S3 artifacts should be deleted when the job is stopped. The default is `FALSE`.
- **id** (in the CLI: `--id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier of the model-training job to be canceled.
- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.

Response

- **status** – a String, of type: `string` (a UTF-8 encoded string).

  The status of the cancellation.

Errors

- UnsupportedOperationException

- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

Model training structures:

CustomModelTrainingParameters (structure)

Contains custom model training parameters. See Custom models in Neptune ML.

Fields

- **sourceS3DirectoryPath** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The path to the Amazon S3 location where the Python module implementing your model is located. This must point to a valid existing Amazon S3 location that contains, at a minimum, a training script, a transform script, and a `model-hpo-configuration.json` file.
- **trainingEntryPointScript** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the entry point in your module of a script that performs model training and takes hyperparameters as command-line arguments, including fixed hyperparameters. The default is `training.py`.
- **transformEntryPointScript** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the entry point in your module of a script that should be run after the best model from the hyperparameter search has been identified, to compute the model artifacts necessary for model deployment. It should be able to run with no command-line arguments.The default is `transform.py`.

Neptune ML model transform API

Model transform actions:

- StartMLModelTransformJob (action)
- ListMLModelTransformJobs (action)
- GetMLModelTransformJob (action)
- CancelMLModelTransformJob (action)

Model transform structures:

- CustomModelTransformParameters (structure)

StartMLModelTransformJob (action)

The AWS CLI name for this API is: `start-ml-model-transform-job`.

Creates a new model transform job. See Use a trained model to generate new model artifacts.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:StartMLModelTransformJob IAM action in that cluster.

Request

- **baseProcessingInstanceType** (in the CLI: `--base-processing-instance-type`) – a String, of type: `string` (a UTF-8 encoded string).

  The type of ML instance used in preparing and managing training of ML models. This is an ML compute instance chosen based on memory requirements for processing the training data and model.
- **baseProcessingInstanceVolumeSizeInGB** (in the CLI: `--base-processing-instance-volume-size-in-gb`) – an Integer, of type: `integer` (a signed 32-bit integer).

  The disk volume size of the training instance in gigabytes. The default is 0. Both input data and the output model are stored on disk, so the volume size must be large enough to hold both data sets. If not specified or 0, Neptune ML selects a disk volume size based on the recommendation generated in the data processing step.

- **customModelTransformParameters** (in the CLI: `--custom-model-transform-parameters`) – A CustomModelTransformParameters object.

  Configuration information for a model transform using a custom model. The `customModelTransformParameters` object contains the following fields, which must have values compatible with the saved model parameters from the training job:
- **dataProcessingJobId** (in the CLI: `--data-processing-job-id`) – a String, of type: `string` (a UTF-8 encoded string).

  The job ID of a completed data-processing job. You must include either `dataProcessingJobId` and a `mlModelTrainingJobId`, or a `trainingJobName`.
- **id** (in the CLI: `--id`) – a String, of type: `string` (a UTF-8 encoded string).

  A unique identifier for the new job. The default is an autogenerated UUID.
- **mlModelTrainingJobId** (in the CLI: `--ml-model-training-job-id`) – a String, of type: `string` (a UTF-8 encoded string).

  The job ID of a completed model-training job. You must include either `dataProcessingJobId` and a `mlModelTrainingJobId`, or a `trainingJobName`.
- **modelTransformOutputS3Location** (in the CLI: `--model-transform-output-s3-location`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The location in Amazon S3 where the model artifacts are to be stored.
- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.
- **s3OutputEncryptionKMSKey** (in the CLI: `--s-3-output-encryption-kms-key`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Key Management Service (KMS) key that SageMaker uses to encrypt the output of the processing job. The default is none.
- **sagemakerIamRoleArn** (in the CLI: `--sagemaker-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role for SageMaker execution. This must be listed in your DB cluster parameter group or an error will occur.

- **securityGroupIds** (in the CLI: `--security-group-ids`) – a String, of type: `string` (a UTF-8 encoded string).

  The VPC security group IDs. The default is None.
- **subnets** (in the CLI: `--subnets`) – a String, of type: `string` (a UTF-8 encoded string).

  The IDs of the subnets in the Neptune VPC. The default is None.
- **trainingJobName** (in the CLI: `--training-job-name`) – a String, of type: `string` (a UTF-8 encoded string).

  The name of a completed SageMaker training job. You must include either `dataProcessingJobId` and a `mlModelTrainingJobId`, or a `trainingJobName`.
- **volumeEncryptionKMSKey** (in the CLI: `--volume-encryption-kms-key`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Key Management Service (KMS) key that SageMaker uses to encrypt data on the storage volume attached to the ML compute instances that run the training job. The default is None.

Response

- **arn** – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of the model transform job.
- **creationTimeInMillis** – a Long, of type: `long` (a signed 64-bit integer).

  The creation time of the model transform job, in milliseconds.
- **id** – a String, of type: `string` (a UTF-8 encoded string).

  The unique ID of the new model transform job.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException

- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

ListMLModelTransformJobs (action)

The AWS CLI name for this API is: `list-ml-model-transform-jobs`.

Returns a list of model transform job IDs. See Use a trained model to generate new model artifacts.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:ListMLModelTransformJobs IAM action in that cluster.

Request

- **maxItems** (in the CLI: `--max-items`) – a ListMLModelTransformJobsInputMaxItemsInteger, of type: `integer` (a signed 32-bit integer), not less than 1 or more than 1024 ?st?s.

  The maximum number of items to return (from 1 to 1024; the default is 10).
- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.

Response

- **ids** – a String, of type: `string` (a UTF-8 encoded string).

  A page from the list of model transform IDs.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

GetMLModelTransformJob (action)

The AWS CLI name for this API is: `get-ml-model-transform-job`.

Gets information about a specified model transform job. See Use a trained model to generate new model artifacts.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:GetMLModelTransformJobStatus IAM action in that cluster.

Request

- **id** (in the CLI: `--id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier of the model-transform job to be reetrieved.
- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.

Response

- **baseProcessingJob** – A MlResourceDefinition object.

  The base data processing job.
- **id** – a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier of the model-transform job to be retrieved.
- **models** – An array of MlConfigDefinition objects.

  A list of the configuration information for the models being used.
- **remoteModelTransformJob** – A MlResourceDefinition object.

  The remote model transform job.
- **status** – a String, of type: `string` (a UTF-8 encoded string).

  The status of the model-transform job.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

CancelMLModelTransformJob (action)

The AWS CLI name for this API is: `cancel-ml-model-transform-job`.

Cancels a specified model transform job. See Use a trained model to generate new model artifacts.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:CancelMLModelTransformJob IAM action in that cluster.

Request

- **clean** (in the CLI: `--clean`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  If this flag is set to `TRUE`, all Neptune ML S3 artifacts should be deleted when the job is stopped. The default is `FALSE`.
- **id** (in the CLI: `--id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The unique ID of the model transform job to be canceled.
- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.

Response

- **status** – a String, of type: `string` (a UTF-8 encoded string).

  the status of the cancelation.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException

- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

Model transform structures:

CustomModelTransformParameters (structure)

Contains custom model transform parameters. See Use a trained model to generate new model artifacts.

Fields

- **sourceS3DirectoryPath** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The path to the Amazon S3 location where the Python module implementing your model is located. This must point to a valid existing Amazon S3 location that contains, at a minimum, a training script, a transform script, and a `model-hpo-configuration.json` file.
- **transformEntryPointScript** – This is a String, of type: `string` (a UTF-8 encoded string).

  The name of the entry point in your module of a script that should be run after the best model from the hyperparameter search has been identified, to compute the model artifacts necessary for model deployment. It should be able to run with no command-line arguments. The default is `transform.py`.

Neptune ML inference endpoint API

Inference endpoint actions:

- CreateMLEndpoint (action)
- ListMLEndpoints (action)
- GetMLEndpoint (action)
- DeleteMLEndpoint (action)

CreateMLEndpoint (action)

The AWS CLI name for this API is: `create-ml-endpoint`.

Creates a new Neptune ML inference endpoint that lets you query one specific model that the model-training process constructed. See Managing inference endpoints using the endpoints command.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:CreateMLEndpoint IAM action in that cluster.

Request

- **id** (in the CLI: `--id`) – a String, of type: `string` (a UTF-8 encoded string).

  A unique identifier for the new inference endpoint. The default is an autogenerated timestamped name.

- **instanceCount** (in the CLI: `--instance-count`) – an Integer, of type: `integer` (a signed 32-bit integer).

  The minimum number of Amazon EC2 instances to deploy to an endpoint for prediction. The default is 1

- **instanceType** (in the CLI: `--instance-type`) – a String, of type: `string` (a UTF-8 encoded string).

  The type of Neptune ML instance to use for online servicing. The default is `ml.m5.xlarge`. Choosing the ML instance for an inference endpoint depends on the task type, the graph size, and your budget.

- **mlModelTrainingJobId** (in the CLI: `--ml-model-training-job-id`) – a String, of type: `string` (a UTF-8 encoded string).

  The job Id of the completed model-training job that has created the model that the inference endpoint will point to. You must supply either the `mlModelTrainingJobId` or the `mlModelTransformJobId`.

- **mlModelTransformJobId** (in the CLI: `--ml-model-transform-job-id`) – a String, of type: `string` (a UTF-8 encoded string).

The job Id of the completed model-transform job. You must supply either the `mlModelTrainingJobId` or the `mlModelTransformJobId`.

- **modelName** (in the CLI: `--model-name`) – a String, of type: `string` (a UTF-8 encoded string).

  Model type for training. By default the Neptune ML model is automatically based on the `modelType` used in data processing, but you can specify a different model type here. The default is `rgcn` for heterogeneous graphs and `kge` for knowledge graphs. The only valid value for heterogeneous graphs is `rgcn`. Valid values for knowledge graphs are: `kge`, `transe`, `distmult`, and `rotate`.

- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role providing Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will be thrown.

- **update** (in the CLI: `--update`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  If set to `true`, `update` indicates that this is an update request. The default is `false`. You must supply either the `mlModelTrainingJobId` or the `mlModelTransformJobId`.

- **volumeEncryptionKMSKey** (in the CLI: `--volume-encryption-kms-key`) – a String, of type: `string` (a UTF-8 encoded string).

  The Amazon Key Management Service (Amazon KMS) key that SageMaker uses to encrypt data on the storage volume attached to the ML compute instances that run the training job. The default is None.

Response

- **arn** – a String, of type: `string` (a UTF-8 encoded string).

  The ARN for the new inference endpoint.

- **creationTimeInMillis** – a Long, of type: `long` (a signed 64-bit integer).

  The endpoint creation time, in milliseconds.

- **id** – a String, of type: `string` (a UTF-8 encoded string).

  The unique ID of the new inference endpoint.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

ListMLEndpoints (action)

The AWS CLI name for this API is: `list-ml-endpoints`.

Lists existing inference endpoints. See Managing inference endpoints using the endpoints command.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:ListMLEndpoints IAM action in that cluster.

Request

- **maxItems** (in the CLI: `--max-items`) – a ListMLEndpointsInputMaxItemsInteger, of type: `integer` (a signed 32-bit integer), not less than 1 or more than 1024 ?st?s.

  The maximum number of items to return (from 1 to 1024; the default is 10.
- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.

Response

- **ids** – a String, of type: `string` (a UTF-8 encoded string).

  A page from the list of inference endpoint IDs.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

GetMLEndpoint (action)

The AWS CLI name for this API is: `get-ml-endpoint`.

Retrieves details about an inference endpoint. See Managing inference endpoints using the endpoints command.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:GetMLEndpointStatus IAM action in that cluster.

Request

- **id** (in the CLI: `--id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier of the inference endpoint.

- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role that provides Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will occur.

Response

- **endpoint** – A MlResourceDefinition object.

  The endpoint definition.
- **endpointConfig** – A MlConfigDefinition object.

  The endpoint configuration
- **id** – a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier of the inference endpoint.
- **status** – a String, of type: `string` (a UTF-8 encoded string).

  The status of the inference endpoint.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException
- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

DeleteMLEndpoint (action)

The AWS CLI name for this API is: `delete-ml-endpoint`.

Cancels the creation of a Neptune ML inference endpoint. See Managing inference endpoints using the endpoints command.

When invoking this operation in a Neptune cluster that has IAM authentication enabled, the IAM user or role making the request must have a policy attached that allows the neptune-db:DeleteMLEndpoint IAM action in that cluster.

Request

- **clean** (in the CLI: `--clean`) – a Boolean, of type: `boolean` (a Boolean (true or false) value).

  If this flag is set to TRUE, all Neptune ML S3 artifacts should be deleted when the job is stopped. The default is FALSE.
- **id** (in the CLI: `--id`) – *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The unique identifier of the inference endpoint.
- **neptuneIamRoleArn** (in the CLI: `--neptune-iam-role-arn`) – a String, of type: `string` (a UTF-8 encoded string).

  The ARN of an IAM role providing Neptune access to SageMaker and Amazon S3 resources. This must be listed in your DB cluster parameter group or an error will be thrown.

Response

- **status** – a String, of type: `string` (a UTF-8 encoded string).

  The status of the cancellation.

Errors

- UnsupportedOperationException
- BadRequestException
- MLResourceNotFoundException
- InvalidParameterException

- ClientTimeoutException
- PreconditionsFailedException
- ConstraintViolationException
- InvalidArgumentException
- MissingParameterException
- IllegalArgumentException
- TooManyRequestsException

Neptune dataplane API Exceptions

Exceptions:

- AccessDeniedException (structure)
- BadRequestException (structure)
- BulkLoadIdNotFoundException (structure)
- CancelledByUserException (structure)
- ClientTimeoutException (structure)
- ConcurrentModificationException (structure)
- ConstraintViolationException (structure)
- ExpiredStreamException (structure)
- FailureByQueryException (structure)
- IllegalArgumentException (structure)
- InternalFailureException (structure)
- InvalidArgumentException (structure)
- InvalidNumericDataException (structure)
- InvalidParameterException (structure)
- LoadUrlAccessDeniedException (structure)
- MalformedQueryException (structure)
- MemoryLimitExceededException (structure)
- MethodNotAllowedException (structure)

- MissingParameterException (structure)
- MLResourceNotFoundException (structure)
- ParsingException (structure)
- PreconditionsFailedException (structure)
- QueryLimitExceededException (structure)
- QueryLimitException (structure)
- QueryTooLargeException (structure)
- ReadOnlyViolationException (structure)
- S3Exception (structure)
- ServerShutdownException (structure)
- StatisticsNotAvailableException (structure)
- StreamRecordsNotFoundException (structure)
- ThrottlingException (structure)
- TimeLimitExceededException (structure)
- TooManyRequestsException (structure)
- UnsupportedOperationException (structure)
- UnloadUrlAccessDeniedException (structure)

AccessDeniedException (structure)

Raised in case of an authentication or authorization failure.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

BadRequestException (structure)

Raised when a request is submitted that cannot be processed.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the bad request.

BulkLoadIdNotFoundException (structure)

Raised when a specified bulk-load job ID cannot be found.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The bulk-load job ID that could not be found.

CancelledByUserException (structure)

Raised when a user cancelled a request.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

The HTTP status code returned with the exception.

- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

ClientTimeoutException (structure)

Raised when a request timed out in the client.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

ConcurrentModificationException (structure)

Raised when a request attempts to modify data that is concurrently being modified by another process.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

The ID of the request in question.

ConstraintViolationException (structure)

Raised when a value in a request field did not satisfy required constraints.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

ExpiredStreamException (structure)

Raised when a request attempts to access an stream that has expired.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

FailureByQueryException (structure)

Raised when a request fails.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

IllegalArgumentException (structure)

Raised when an argument in a request is not supported.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

InternalFailureException (structure)

Raised when the processing of the request failed unexpectedly.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.

- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

InvalidArgumentException (structure)

Raised when an argument in a request has an invalid value.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

InvalidNumericDataException (structure)

Raised when invalid numerical data is encountered when servicing a request.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

InvalidParameterException (structure)

Raised when a parameter value is not valid.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request that includes an invalid parameter.

LoadUrlAccessDeniedException (structure)

Raised when access is denied to a specified load URL.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

MalformedQueryException (structure)

Raised when a query is submitted that is syntactically incorrect or does not pass additional validation.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

A detailed message describing the problem.

- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the malformed query request.

MemoryLimitExceededException (structure)

Raised when a request fails because of insufficient memory resources. The request can be retried.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request that failed.

MethodNotAllowedException (structure)

Raised when the HTTP method used by a request is not supported by the endpoint being used.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

MissingParameterException (structure)

Raised when a required parameter is missing.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in which the parameter is missing.

MLResourceNotFoundException (structure)

Raised when a specified machine-learning resource could not be found.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

ParsingException (structure)

Raised when a parsing issue is encountered.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

The HTTP status code returned with the exception.

- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

PreconditionsFailedException (structure)

Raised when a precondition for processing a request is not satisfied.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

QueryLimitExceededException (structure)

Raised when the number of active queries exceeds what the server can process. The query in question can be retried when the system is less busy.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

The ID of the request which exceeded the limit.

QueryLimitException (structure)

Raised when the size of a query exceeds the system limit.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request that exceeded the limit.

QueryTooLargeException (structure)

Raised when the body of a query is too large.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request that is too large.

ReadOnlyViolationException (structure)

Raised when a request attempts to write to a read-only resource.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in which the parameter is missing.

S3Exception (structure)

Raised when there is a problem accessing Amazon S3.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

ServerShutdownException (structure)

Raised when the server shuts down while processing a request.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.

- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

StatisticsNotAvailableException (structure)

Raised when statistics needed to satisfy a request are not available.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

StreamRecordsNotFoundException (structure)

Raised when stream records requested by a query cannot be found.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

ThrottlingException (structure)

Raised when the rate of requests exceeds the maximum throughput. Requests can be retried after encountering this exception.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request that could not be processed for this reason.

TimeLimitExceededException (structure)

Raised when the an operation exceeds the time limit allowed for it.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.
- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.
- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request that could not be processed for this reason.

TooManyRequestsException (structure)

Raised when the number of requests being processed exceeds the limit.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

The HTTP status code returned with the exception.

- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.

- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request that could not be processed for this reason.

UnsupportedOperationException (structure)

Raised when a request attempts to initiate an operation that is not supported.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.

- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.

- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The ID of the request in question.

UnloadUrlAccessDeniedException (structure)

Raised when access is denied to a URL that is an unload target.

Fields

- **code** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  The HTTP status code returned with the exception.

- **detailedMessage** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

  A detailed message describing the problem.

- **requestId** – This is *Required:* a String, of type: `string` (a UTF-8 encoded string).

The ID of the request in question.