Neptune Analytics User Guide

Neptune Analytics

Neptune Analytics: Neptune Analytics User Guide

Table of Contents

What is Neptune Analytics?

Neptune Analytics is a memory-optimized graph database engine for analytics. With Neptune Analytics, you can get insights and find trends by processing large amounts of graph data in seconds. To analyze graph data quickly and easily, Neptune Analytics stores large graph datasets in memory. It supports a library of optimized graph analytic algorithms, low-latency graph queries, and vector search capabilities within graph traversals.

Neptune Analytics is an ideal choice for investigatory, exploratory, or data-science workloads that require fast iteration for data, analytical and algorithmic processing, or vector search on graph data. It complements Amazon Neptune Database, a popular managed graph database. To perform intensive analysis, you can load the data from a Neptune Database graph or snapshot into Neptune Analytics. You can also load graph data that's stored in Amazon S3.

You can get started by creating a new Neptune Analytics graph and loading data into it in a number of ways.

Topics

Features

Neptune Analytics operates in a managed environment that can load data extremely fast into memory and run graph algorithms natively. With Neptune Analytics, you can perform in-database analytics on large graphs.

This functionality lets you perform business intelligence and custom analytical queries, and use pre-built graph algorithms with the openCypher language. For example, with Neptune Analytics you can ingest text from cybersecurity reports to analyze relationships within security environments and calculate vulnerability mitigations using graph algorithms or openCypher queries.

Neptune Analytics offers a graph as a service experience by managing graphs instead of infrastructure, so you can focus on queries and workflows to solve problems. It automatically provisions the compute resources necessary to run analytics workloads based on the size of the graph.

You can load graph data into Neptune Analytics from Amazon S3 or from a Neptune Database endpoint. You can then run graph analytics queries using pre-built or custom graph queries.

When to use Neptune Analytics and when to use Neptune Database

Amazon Neptune makes it easy to work with graph data in the AWS Cloud. Amazon Neptune includes both Neptune Database and Neptune Analytics.

Neptune Database is a serverless graph database designed for optimal scalability and availability. It provides a solution for graph database workloads that need to scale to 100,000 queries per second, Multi-AZ high availability, and multi-Region deployments. You can use Neptune Database for social networking, fraud alerting, and Customer 360 applications.

Neptune Analytics is an analytics database engine 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.

You can use Neptune Analytics to analyze and query graphs in data science workflows that build targeted content recommendations, assist with fraud investigations, and detect network threats.

By providing a simple API for loading, querying, and analyzing graph data, Neptune Analytics also removes the overhead of building and managing complex data-analytics pipelines.

Neptune Analytics makes it easy to apply powerful algorithms both to the data in your Neptune Database and to graph data that's stored externally. Because Neptune Analytics can load a large dataset very quickly into memory, it becomes possible to analyze graphs with tens of billions of relationships and to process thousands of analytic queries per second using popular graph analytics algorithms.

Changes and updates to Neptune Analytics

The following table lists important releases relating to Neptune Analytics.

Change	Description	Date
Byte handling	Previously, when data was loaded using the `Byte` data type it was erroneously stored as an unsigned byte, this has been updated to store the data a signed byte. Previously loaded data and queries will not be impacted.	July 29, 2024
Updated float serialization	Float serialization changed to use a lesser number of digits for higher precision. This will change the float values that are returned from the server.	June 20, 2024
UNWIND performance improvement	Improved the UNWIND operations (e.g. transform a list of values within a property into individual vertices or edges) to help prevent out of memory (OOM) situations.	June 20, 2024
Query plan cache update	Fixed an issue in query plan cache when `skip` or `limit` is used in an inner `WITH` clause and parameterized. This fix ensures that parameter values for `SKIP` and `LIMIT` are now properly handled, providing accurate results	June 20, 2024

	for every execution. If you disabled the query plan cache, you can now remove the query hint `QUERY:PLANCACHE "disabled"` when submitting a query.	
Increased efficiency when handling special characters	Fixed an issue with handling of node and relationship labels, node identifiers, and property names containing the “Ā” character (Latin uppercase letter A with Macron, unicode code point U+0100).	June 20, 2024
Datetime formats	Extended support for datetime formats.	June 20, 2024
Condition key - neptune-graph:PublicConnectivity	neptune-graph:PublicConnectivity filters access by the value of the public connectivity parameter provided in the request or its default value, if unspecified. All access to graphs is IAM authenticated.	April 29, 2024

StartImportTask available	Neptune Analytics now allows you to efficiently import large datasets into an already provisioned graph database using the `StartImportTask` API. This API facilitates the direct loading of data from an Amazon S3 bucket into an **empty** Neptune Analytics graph. This is designed for loading data into existing empty clusters.	March 30, 2024
Neptune Analytics available in Europe (London) region	Neptune Analytics is now available in the `eu-west-2` Europe (London) region.	March 14, 2024
Column delimiter `;` is now supported in values	When a column delimiter presents in the values, e.g. for `'two;words'` , two values {'two','words'} will be inserted. However, if the column delimiter is escaped by a backslash `'\'`, the value will be inserted as a whole with the escape character removed, e.g. `'one\;word'` will be inserted as 'one;word'.	February 27, 2024
Graph provisioning time reduced to five minutes or less	Neptune Analytics graphs are now provisioned and ready to be used in five minutes or less.	February 19, 2024

Query improvements - Data Plane SDK	The Neptune Analytics data API provides support for data operations including query execution, query status checking, query cancellation, and graph summarizing via the HTTPS endpoint, the AWS CLI, and the SDK.	February 2, 2024
Initial release	Initial release of Neptune Analytics.	November 29, 2023

Getting started

To get started using Neptune Analytics, you need to create a graph using the AWS console, the AWS CLI, or AWS CloudFormation. You can load data into a graph from another Neptune database, Neptune database cluster snapshot, or from files located in Amazon S3.

Topics

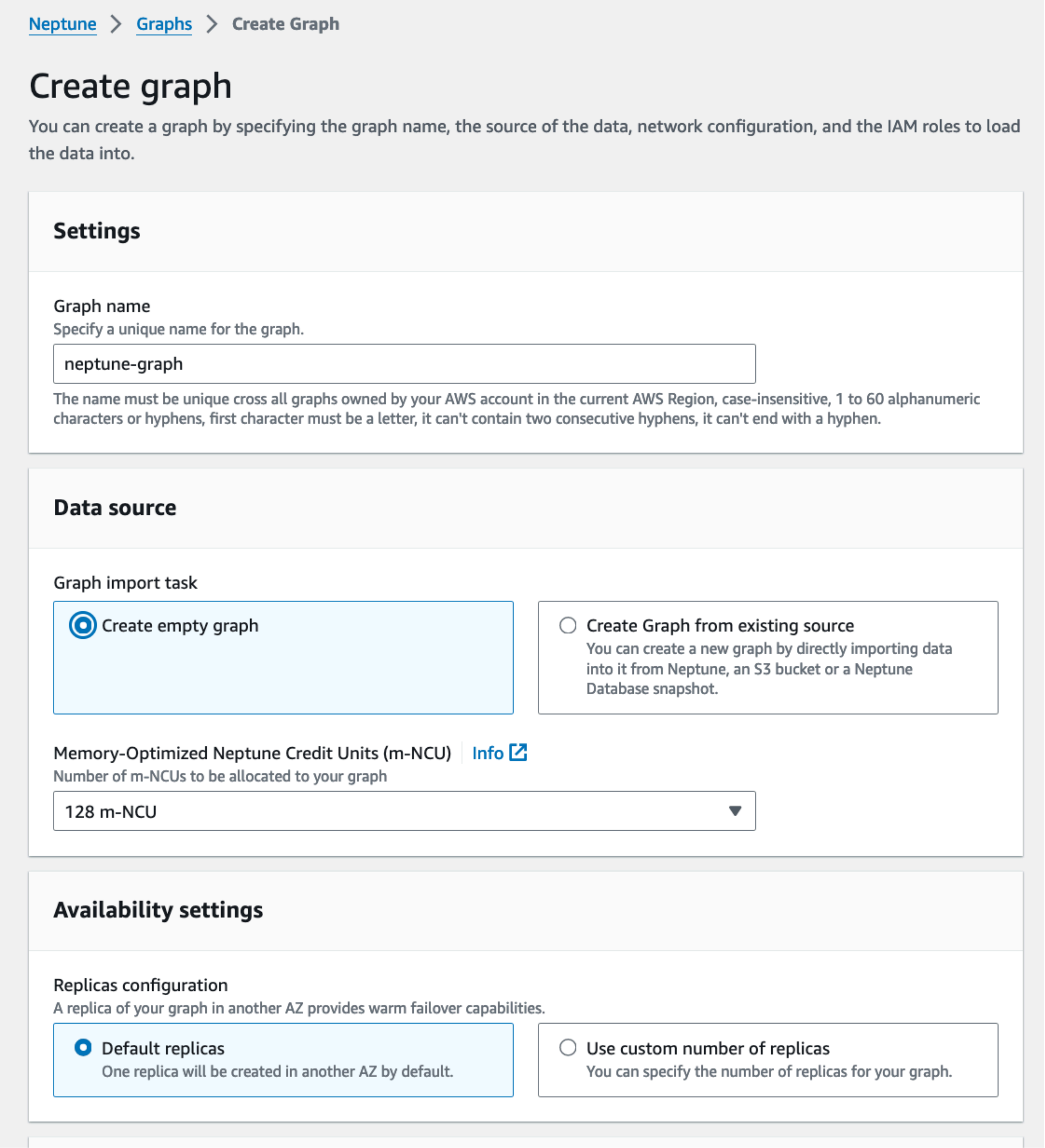
- Create an empty Neptune graph
- Create a Neptune graph from existing sources
- Connecting to a graph

Create an empty Neptune graph

AWS console

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/.
2. In the upper right corner of the console, choose the AWS region in which you want to create the graph.
3. In the navigation pane, choose **Graphs** in the **Analytics** section.
4. Choose the **Create graph** button.
5. In **settings**, input the **graph name**, size, and replica configuration.
6. In the **data source** section, choose the **empty graph** option.

> **Note**
>
> Additional charges equivalent to the m-NCUs selected for the graph apply for each replica.



Neptune > Graphs > Create Graph

Create graph

You can create a graph by specifying the graph name, the source of the data, network configuration, and the IAM roles to load the data into.

Settings

Graph name

Specify a unique name for the graph.

neptune-graph

The name must be unique cross all graphs owned by your AWS account in the current AWS Region, case-insensitive, 1 to 60 alphanumeric characters or hyphens, first character must be a letter, it can't contain two consecutive hyphens, it can't end with a hyphen.

Data source

Graph import task

● Create empty graph

○ Create Graph from existing source
You can create a new graph by directly importing data into it from Neptune, an S3 bucket or a Neptune Database snapshot.

Memory-Optimized Neptune Credit Units (m-NCU) Info

Number of m-NCUs to be allocated to your graph

128 m-NCU ▼

Availability settings

Replicas configuration

A replica of your graph in another AZ provides warm failover capabilities.

● Default replicas
One replica will be created in another AZ by default.

○ Use custom number of replicas
You can specify the number of replicas for your graph.

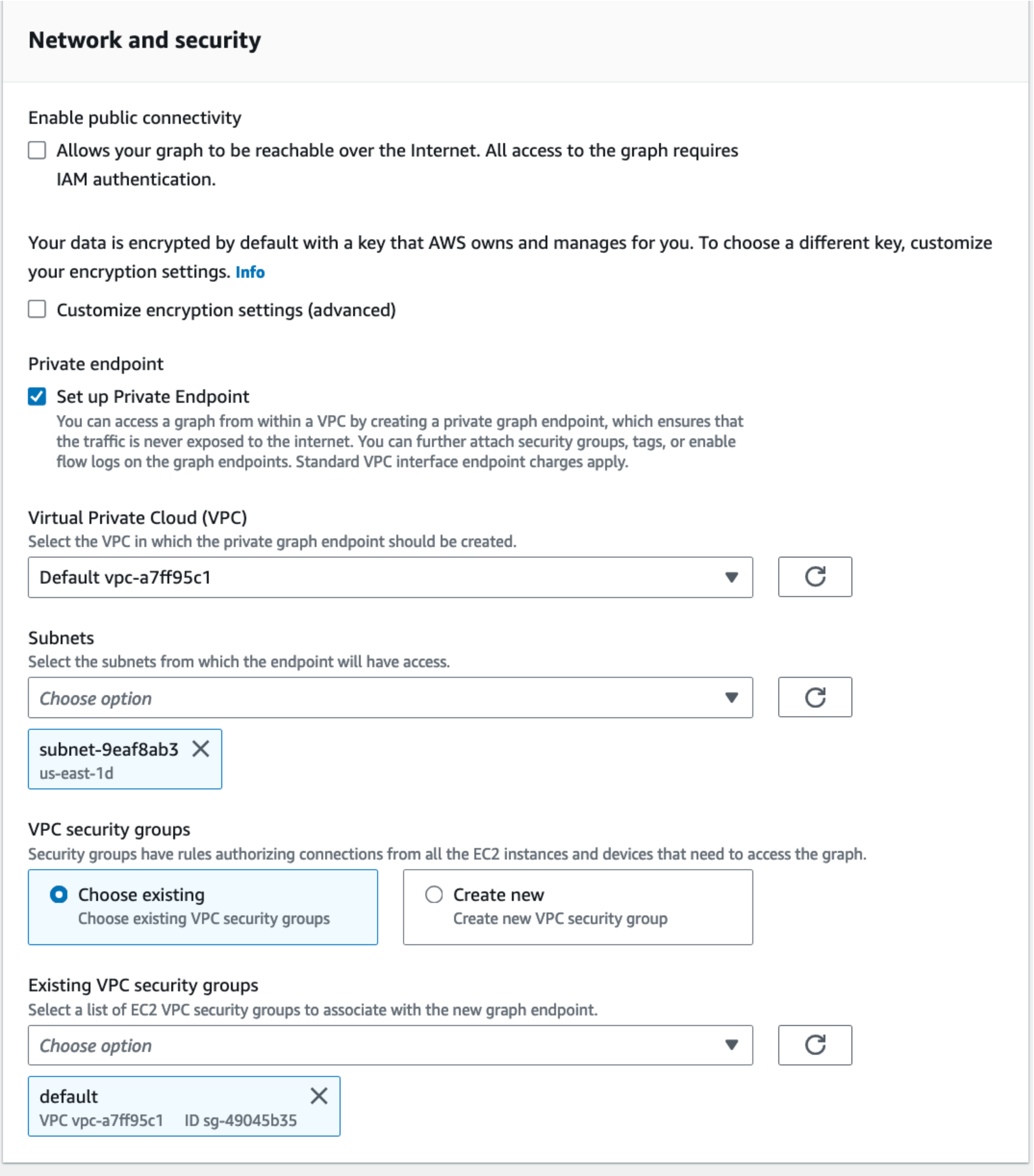
7. You can connect to a Neptune graph from a public endpoint or a private endpoint. Select your network configuration accordingly.

Note

If you're creating a private graph endpoint, the following permissions are required:

- ec2:CreateVpcEndpoint
- ec2:DescribeAvailabilityZones
- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets
- ec2:DescribeVpcAttribute
- ec2:DescribeVpcEndpoints
- ec2:DescribeVpcs
- ec2:ModifyVpcEndpoint
- route53:AssociateVPCWithHostedZone

For more information about required permissions, see Actions defined by Neptune Analytics.



Network and security

Enable public connectivity

□ Allows your graph to be reachable over the Internet. All access to the graph requires IAM authentication.

Your data is encrypted by default with a key that AWS owns and manages for you. To choose a different key, customize your encryption settings. Info

□ Customize encryption settings (advanced)

Private endpoint

☑ Set up Private Endpoint

You can access a graph from within a VPC by creating a private graph endpoint, which ensures that the traffic is never exposed to the internet. You can further attach security groups, tags, or enable flow logs on the graph endpoints. Standard VPC interface endpoint charges apply.

Virtual Private Cloud (VPC)

Select the VPC in which the private graph endpoint should be created.

Default vpc-a7ff95c1

Subnets

Select the subnets from which the endpoint will have access.

Choose option

subnet-9eaf8ab3 ✕
us-east-1d

VPC security groups

Security groups have rules authorizing connections from all the EC2 instances and devices that need to access the graph.

● Choose existing
Choose existing VPC security groups

○ Create new
Create new VPC security group

Existing VPC security groups

Select a list of EC2 VPC security groups to associate with the new graph endpoint.

Choose option

default ✕
VPC vpc-a7ff95c1 ID sg-49045b35

8. Additionally, you can select vector search configuration for the graph. For more information on vector search configuration, see Vector indexing.

9. Choose **Create graph**.

AWS CLI

Create a Neptune graph using the AWS CLI.

Create a public graph endpoint:

```
aws neptune-graph create-graph --graph-name 'test-neptune-graph' \
--region us-east-1 --provisioned-memory 128  --public-connectivity \
--replica-count 0 --vector-search '{"dimension": 384}'
```

Create a private graph endpoint:

```
aws neptune-graph create-private-graph-endpoint –vpc-id vpc-0a9b7a5b15 \
--subnet-ids subnet-06a4b41a6221b subnet-0840a4b327ab77 subnet-0353627ab123 \
--vpc-security-group-ids sg-0ab7abab56ab \
--graph-identifier g-146a51b7a151ba –region us-east-1
```

Check the status of graph creation:

```
aws neptune-graph get-graph --graph-identifier <graph-id>
```

List all graphs in the default region:

```
aws neptune-graph list-graphs
```

AWS CloudFormation

Instead of using the console to create your Neptune graph, you can use AWS CloudFormation to provision AWS resources by treating infrastructure as code. To help you organize your AWS resources into smaller and more manageable units, you can use the AWS CloudFormation nested stack functionality. For more information, see Creating a stack on the AWS CloudFormation console and working with nested stacks.

AWS CloudFormation is free, but the resources that CloudFormation creates are live. You incur the standard usage fees for these resources until you terminate them. The total charges will be minimal. For information about how you might minimize any charges, see AWS free tier.

To create your resources using the AWS CloudFormation console, complete the following steps:

1. Create the CloudFormation template.
2. Configure your resources using CloudFormation.

The following sample template will create a Neptune graph with a public endpoint.

```
AWSTemplateFormatVersion: 2010-09-09
Description: NeptuneGraph Graph Create Demo using CloudFormation
Resources:
  NeptuneGraph:
    Type: AWS::NeptuneGraph::Graph
    DeletionPolicy: Delete
    Properties:
      DeletionProtection: false
      GraphName: neptune-graph-demo
      ProvisionedMemory: 128
      ReplicaCount: 1
      PublicConnectivity: true
      Tags:
        - Key: stage
          Value: test
```

The following sample template will create a Neptune graph with a public endpoint.

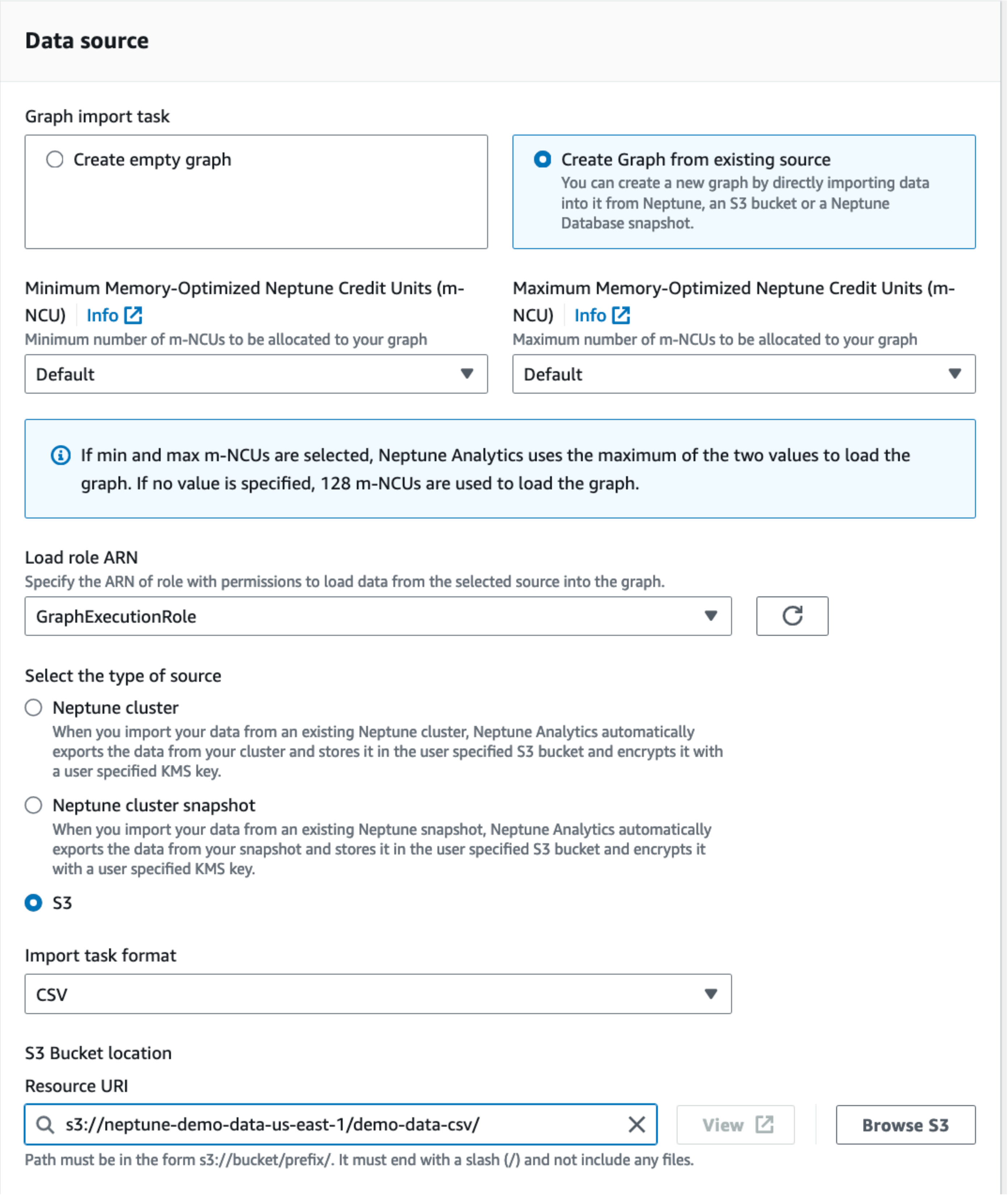
```
AWSTemplateFormatVersion: 2010-09-09
Description: NeptuneGraph Graph Create Demo using CloudFormation
Resources:
  NeptuneGraph:
    Type: AWS::NeptuneGraph::Graph
    DeletionPolicy: Delete
    Properties:
      DeletionProtection: false
      GraphName: neptune-graph-demo
      ProvisionedMemory: 128
      ReplicaCount: 1
      PublicConnectivity: false
      Tags:
        - Key: stage
          Value: test
  NeptuneGraphPrivateEndpoint:
    Type: AWS::NeptuneGraph::PrivateGraphEndpoint
    DeletionPolicy: Delete
    Properties:
      GraphIdentifier: NeptuneGraph
      VpcId: myVpc
```

> ⚠ **Important**
>
> You can't change the graph name, VPC, subnet ids and vector search configuration. After starting the graph creation, the graph has a status of **Creating** until the graph is ready to use. When the status of the graph changes to **Available**, you can connect to the DB cluster at that time. Depending on the configuration, it can take up to 20 minutes before the new graph is available.

Create a Neptune graph from existing sources

You can load data into a Neptune graph from another Neptune database, Neptune database cluster snapshot, or from Amazon S3 files. Select the data sources and an IAM role for the data import accordingly. For more information about loading data, see Create a graph from Amazon S3, a Neptune cluster, or a snapshot.

AWS console



Data source

Graph import task

○ Create empty graph

● Create Graph from existing source
You can create a new graph by directly importing data into it from Neptune, an S3 bucket or a Neptune Database snapshot.

Minimum Memory-Optimized Neptune Credit Units (m-NCU) Info
Minimum number of m-NCUs to be allocated to your graph
Default ▼

Maximum Memory-Optimized Neptune Credit Units (m-NCU) Info
Maximum number of m-NCUs to be allocated to your graph
Default ▼

ⓘ If min and max m-NCUs are selected, Neptune Analytics uses the maximum of the two values to load the graph. If no value is specified, 128 m-NCUs are used to load the graph.

Load role ARN
Specify the ARN of role with permissions to load data from the selected source into the graph.
GraphExecutionRole ▼

Select the type of source

○ Neptune cluster
When you import your data from an existing Neptune cluster, Neptune Analytics automatically exports the data from your cluster and stores it in the user specified S3 bucket and encrypts it with a user specified KMS key.

○ Neptune cluster snapshot
When you import your data from an existing Neptune snapshot, Neptune Analytics automatically exports the data from your snapshot and stores it in the user specified S3 bucket and encrypts it with a user specified KMS key.

● S3

Import task format
CSV ▼

S3 Bucket location

Resource URI
s3://neptune-demo-data-us-east-1/demo-data-csv/
Path must be in the form s3://bucket/prefix/. It must end with a slash (/) and not include any files.

AWS CLI

The following example creates a graph and loads data from Amazon S3.

```
aws neptune-graph create-graph-using-import-task \
--graph-name "neptune-graph-from-s3-source" \
--region "us-east-1" \
--format "CSV" \
--role-arn "arn:aws:iam::1234567890124:role/GraphExecutionRole" \
--source "s3://neptune-demo-test-us-east-1/test-data-csv/" \
--public-connectivity \
--min-provisioned-memory 256 \
--max-provisioned-memory 256
```

Connecting to a graph

In Neptune Analytics, you can provision your graph to be accessed publicly over the internet or have a private endpoint to access the graph within a VPC. If your graph is not configured for public connectivity, then you must create a private endpoint for your Neptune Analytics graph that allows access to the graph only from within the same Amazon Virtual Private Cloud (VPC) and availability zones associated with the subnet associated with the graph’s private endpoint (You must ensure the subnets belong to all the availability zones in the VPC). This means that applications using Neptune Analytics must be deployed in the same VPC; or For applications which are deployed in different VPC but uses techniques like VPC peering, AWS Site-to-Site VPN connections, or AWS Direct Connect connections might face issues with DNS resolution to connect to private graph endpoint.

If your graph is configured for public connectivity, you can connect to your graph from multiple VPCs and from the internet. This allows you to access a Neptune Analytics graph without also setting up additional supporting AWS services. The simplicity of setting up public connectivity-enabled graphs makes it useful for initial exploration of the service.

Graphs are created with public connectivity disabled by default. However, this can be configured by enabling public connectivity at graph creation or by updating the graph configuration post-creation.

Note

All Neptune Analytics graphs are configured to use AWS Identity and Access Management (IAM) for authentication and authorization. This means that all requests to the graph should be signed using AWS Signature Version 4 (SIGV4). If you are using the AWS CLI or SDK to connect, then the signing of the requests is handled by the client library. The library requires the user to provide the credentials to sign using one of the known methods. You can also make HTTP requests to the APIs by using AWSCurl, which provides a curl like interface to make HTTP requests and supports SIGV4. For Neptune Analytics specific IAM documentation please refer to the Neptune Analytics user guide Security IAM section.

Topics

Connecting to a private endpoint from within the same VPC

For graphs using private endpoints, you can connect to your graph from any resource that has access to the private VPC, such as AWS Lambda, an Amazon SageMaker notebook instance, an Amazon EC2 instance, etc. The instance must be in the same VPC and subnet as the private endpoint for your graph. Ensure that the security group attached to the VPC endpoint of your private graph's endpoint allows ingress on port 443, and optionally port 8182.

For details on how to use notebooks and how to create one capable of connecting to the private endpoint of your graph, see the Neptune Analytics user guide section on notebooks, making sure to supply the necessary VPC and subnet identifier when setting up your network options. If the VPC CIDR is 172.17.0.0/16, notebooks will have some difficult connecting the graph endpoints.

You can also create an Amazon EC2 instance to connect to the private endpoint of your graph. You will need to select the correct VPC and availability zone to match your graph’s private endpoint. When prompted for a security group to associate with the instance, create or choose one that has inbound TCP rules allowing ingress traffic over ports 22 (for SSH), and egress traffic over port 443 if custom egress rules are needed. For the detailed prerequisites and steps to create and connect to an Amazon EC2 instance, see the Amazon EC2 user guide.

Note

For troubleshooting connectivity issues refer to the reachability analyzer guide. You can get destination VPC endpointId by using the GetPrivateGraphEndpoint API.

Connecting to a private endpoint from a different VPC (including cross-account)

In some cases, you may be required to connect to your graph from a different VPC without enabling public connectivity. For example, applications that segregate AWS services using different VPCs or different accounts. In this case, connectivity can be achieved through the use of private graph endpoints and Amazon Route 53 private hosted zones. The steps in the following procedure refer to a client in VPC B, wanting to access a Neptune Analytics graph in VPC A.

1. **Establish network connectivity between VPC A and VPC B**

   You can use any method that allows traffic to move between VPCs. For example, VPC peering or AWS Transit Gateway. In addition to establishing the network connection, make sure your security groups and network ACLs allow traffic between the two VPCs. You can verify network connectivity with the reachability analyzer.

2. **Create a private graph endpoint in VPC A**

   If you haven't already, create a private graph endpoint in VPC A. This can be done through the console or the CreatePrivateGraphEndpoint API. Once created, collect the DNS name for the VPC endpoint that was deployed.

   1. Find the VPC endpoint ID from the value of **vpcEndpointId** when calling the ListPrivateGraphEndpoints API.
   2. From the console or using the DescribeVpcEndpoints API, collect the DNS name of the VPC endpoint. This should have the format of `vpce-<alphanumeric>.vpce-svc-<alphanumeric>.<region>.vpce.amazonaws.com`.

3. Use Amazon Route 53 to create a private hosted zone for VPC B.

   1. From the Route 53 console, choose **Create hosted zone**.

2. Set the **domain name** of the private hosted zone to the graph endpoint of the Neptune Analytics graph. The graph endpoint should have the format of `g-<alphanumeric>.<region>.neptune-graph.amazonaws.com`.
3. Set the **Type** to **Private hosted zone**.
4. Associate VPC B with the hosted zone.
5. Choose **Create hosted zone**.

Add a record to route traffic destined for the graph endpoint to the VPC endpoint directly.

1. When the hosted zone is created, choose **Create record**.
2. From the creation wizard, choose **Simple routing** for the routing policy.
3. Choose **Define simple record**. Set the **Record type** to **A**, which routes traffic to an IPv4 address and some AWS resources. Set **Value/Route** traffic to to the DNS hostname of the VPC endpoint from Step 2. This should have the format of `vpce-<alphanumeric>.vpce-svc-<alphanumeric>.<region>.vpce.amazonaws.com`.

To use private hosted zones, `enableDnsHostnames` and `enableDnsSupport` should be set to `true` for both VPCs. Depending on your networking configuration, other considerations may apply when using private hosted zones. See Route 53 private hosted zone considerations documentation to validate your setup.

4. **Establish cross-account IAM permissions (only required for cross-account access)**

In addition to the network connectivity established in prior steps, if the client in VPC B is in a different account (Account B), they will also need appropriate credentials to access the Neptune Analytics graph in VPC A (in Account A). You can use cross-account IAM roles to give permissions to the client.

1. Create the IAM role and policy that the client in Account B will be using (IAM role B).
2. Create an IAM role and policy in Account A that grants the desired permissions to the Neptune Analytics graph (IAM role A). Make sure that there are also permissions for IAM role B to assume this role.
3. Add permissions to IAM role B to assume the IAM role A.
4. When making a cross-account call to the Neptune Analytics graph, use the AWS Security Token Service AssumeRole API to have IAM role B assume IAM role A. Use the returned

credentials when making requests to the Neptune Analytics graph, e.g. via AWS SDK, awscurl, etc.

Accessing the graph

Once you’ve created a graph and set up the prerequisites for connecting to that graph, you can proceed with accessing your graph to load and query data. This section contains an explanation of ways you can communicate with your graph along with some example queries. For details on how to load data, see the loading data section in the Neptune Analytics user guide.

Using a notebook

You can access to your Neptune Analytics graph through a Neptune workbench, which provides visualization tools on top of Neptune Analytics which can help with interpreting query results. For more information on how to set up and use a graph notebook, see the notebooks section in the Neptune Analytics user guide.

Example

The cell magic below submits an openCypher query that returns a single node.

```
%%oc
MATCH (n) RETURN n LIMIT 1
```

Using the AWS SDK

With the AWS SDK, you can access your graph using a programming language of your choice, which provides clean integration between Neptune Analytics and your applications. With the Neptune Analytics SDK (service name Neptune graph), you can perform data operations like querying and summarization in addition to control plane operations such as graph creation, deletion, and modification. For a list of the supported programming languages and directions for setting up the SDK in each language, see the AWS developer tools documentation.

Direct links to the API reference documentation for the Neptune Analytics service in each SDK language can be found below:

Programming language	Neptune graph API reference
C++	https://sdk.amazonaws.com/cpp/api/LATEST/aws-cpp-sdk-neptune-graph/html/annotated.html
Go	https://pkg.go.dev/github.com/aws/aws-sdk-go-v2/service/neptunegraph
Java	https://sdk.amazonaws.com/java/api/latest/software/amazon/awssdk/services/neptunegraph/package-summary.html
JavaScript	https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/Package/-aws-sdk-client-neptune-graph/
Kotlin	https://sdk.amazonaws.com/kotlin/api/latest/neptunegraph/index.html
.NET	https://docs.aws.amazon.com/sdkfornet/v3/apidocs/items/NeptuneGraph/NNeptuneGraph.html
PHP	https://docs.aws.amazon.com/aws-sdk-php/v3/api/namespace-Aws.NeptuneGraph.html
Python	https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/neptune-graph.html
CLI	https://docs.aws.amazon.com/cli/latest/reference/neptune-graph/
Ruby	https://docs.aws.amazon.com/sdk-for-ruby/v3/api/Aws/NeptuneGraph.html
Rust	https://crates.io/crates/aws-sdk-neptunegraph

Programming language	Neptune graph API reference
Swift	https://sdk.amazonaws.com/swift/api/awsneptunegraph/0.37.0/documentation/awsneptunegraph

Examples

Python

The code sample below uses the Python SDK to submit a query that returns a single node and prints the result.

1. Follow the installation instructions to install Boto3. If you are using a SageMaker hosted Jupyter notebook, Boto3 will be pre-installed, but you may need to update it.
2. Configure your Boto3 credentials by following the configuring credentials guide.
3. Create a file named `queryExample.py`.
4. In that file, paste the following code. It will set up a Neptune graph client, execute an openCypher query request, and print the result. Replace the graph identifier and query string as needed.

```
import boto3

// Set up the Neptune Graph client.
client = boto3.client('neptune-graph')

// Execute a query.
response = client.execute_query(
    graphIdentifier='g-0123456789',
    queryString='MATCH (n) RETURN n LIMIT 1',
    language='OPEN_CYPHER'
)

// Print the response.
print(response['payload'].read().decode('utf-8'))
```

5. Run the sample code by entering `python queryExample.py`.

Go

The code sample below uses the Go SDK to submit a query that returns a single node and prints the result.

1. Follow the installation instructions to install Go and the AWS SDK for Go.
2. Create a file named `queryExample.go`.
3. In that file, paste the following code. It will set up a Neptune graph client, execute an openCypher query request, and print the result. Replace the graph identifier and query string as needed.

```
package main

import (
        "context"
        "log"
        "github.com/aws/aws-sdk-go-v2/aws"
        "github.com/aws/aws-sdk-go-v2/config"
        "github.com/aws/aws-sdk-go-v2/service/neptunegraph"
        "io"
)

func main() {
        // Load the Shared AWS Configuration (~/.aws/config)
        cfg, err := config.LoadDefaultConfig(context.TODO())
        if err != nil {
                log.Fatal(err)
        }

        // Create an Amazon Neptune Analytics service client
        client := neptunegraph.NewFromConfig(cfg)

        // Execute a query
        output, err := client.ExecuteQuery(context.TODO(),
 &neptunegraph.ExecuteQueryInput{
                GraphIdentifier: aws.String("g-0123456789"),
                Language: "OPEN_CYPHER",
                QueryString: aws.String("MATCH (n) RETURN n LIMIT 1"),
        })
        if err != nil {
                log.Fatal(err)
        }
        // Print the results
        bytes, err := io.ReadAll(output.Payload)
        log.Println(string(bytes))
}
```

4. Run the sample code by entering `go run queryExample.go`.

Node.js

The Node.js code sample below uses the JavaScript SDK to submit a query that returns a single node and prints the result.

1. Follow the installation instructions to install Node.js and set up your package structure. For this example, install the Neptune graph client package instead of the Amazon S3 client package: `npm install @aws-sdk/client-neptune-graph`.
2. Create a file in that directory structure named `queryExample.js`.
3. In that file, paste the following code. It will set up a Neptune graph client, execute an openCypher query request, and print the result. Replace the graph identifier and query string as needed.

```
import { NeptuneGraphClient, ExecuteQueryCommand } from "@aws-sdk/client-neptune-
graph";

// Set up the client.
const neptuneGraphClient = new NeptuneGraphClient({});

// Send the query request.
const output = await neptuneGraphClient.send(
    new ExecuteQueryCommand({
        graphIdentifier: "g-0123456789",
        language: "OPEN_CYPHER",
        queryString: "MATCH (n) RETURN n LIMIT 1"
    })
);

// Print the result.
console.log(await output.payload.transformToString('utf-8'));
```

4. Run the sample code by entering `node queryExample.js`.

Java

The tutorial below sets up a project that uses the Java SDK to submit a query that returns a single node and prints the result.

1. To get started with the Java SDK, follow the installation instructions to install Java and set up a build tool that supports Maven central. This example will use Apache Maven.
2. Follow the steps to create a project using Maven based on the quickstart template. When executing these steps, please make the following modifications:

   > **Note**
   >
   > When generating the project from the template, specify a version of the Java SDK that includes the Neptune graph service APIs. For this example, use 2.25.7 as your archetype version.

   ```
   mvn archetype:generate \
     -DarchetypeGroupId=software.amazon.awssdk \
     -DarchetypeArtifactId=archetype-app-quickstart \
     -DarchetypeVersion=2.25.7
   ```

   Running the command above will present you with several prompts. When asked to provide a 'service' (i.e., the service whose client and APIs you plan to use for this tutorial), please enter `neptunegraph` as the service name. An updated table of prompts and values can be found below:

Prompt	Value to enter
Define value for property 'service':	neptunegraph
Define value for property 'httpClient':	apache-client
Define value for property 'nativeImage':	false
Define value for property 'credentialProvider':	identity-center

Prompt	Value to enter
Define value for property 'groupId':	org.example
Define value for property 'artifactId':	getstarted
Define value for property 'version' 1.0-SNAPSHOT:	<Enter>
Define value for property 'package' org.example:	<Enter>

3. After generating the project structure, you should see three Maven-generated classes defined in the `getstarted/src/main/java/org/example/` directory: `App.java`, `DependencyFactory.java`, and `Handler.java`. For details on each of these classes, see step 3 in the [SDK for Java] guide. Since this example uses the `neptunegraph` service, the Maven-generated code in the DependencyFactor and Handler classes will be using a different client than the code samples provided there. Refer to the Neptune graph-specific equivalents of the auto-generated classes below:

   a. Maven-generated `App.java` - this file is the same regardless of the service used.

   ```
   package org.example;
   import org.slf4j.Logger;
   import org.slf4j.LoggerFactory;

   public class App {
       private static final Logger logger = LoggerFactory.getLogger(App.class);

       public static void main(String... args) {
           logger.info("Application starts");

           Handler handler = new Handler();
           handler.sendRequest();

           logger.info("Application ends");
       }
   }
   ```

   b. Maven-generated `DependencyFactory.java` - This file uses the client class `NeptuneGraphClient` because `neptunegraph` was chosen as the service during the project setup.

```
package org.example;

import software.amazon.awssdk.http.apache.ApacheHttpClient;
import software.amazon.awssdk.services.neptunegraph.NeptuneGraphClient;

/**
 * The module containing all dependencies required by the {@link Handler}.
 */
public class DependencyFactory {

    private DependencyFactory() {}

    /**
     * @return an instance of NeptuneGraphClient
     */
    public static NeptuneGraphClient neptuneGraphClient() {
        return NeptuneGraphClient.builder()
                       .httpClientBuilder(ApacheHttpClient.builder())
                       .build();
    }
}
```

c. Maven-generated `Handler.java` - This file uses the client class `NeptuneGraphClient` because `neptunegraph` was chosen as the service during the project setup.

```
package org.example;

import software.amazon.awssdk.services.neptunegraph.NeptuneGraphClient;


public class Handler {
    private final NeptuneGraphClient neptuneGraphClient;

    public Handler() {
        neptuneGraphClient = DependencyFactory.neptuneGraphClient();
    }

    public void sendRequest() {
        // TODO: invoking the api calls using neptuneGraphClient.
    }
}
```

4. Make changes to the Maven-generated handler class to fill in the missing logic in `sendRequest` to use the `NeptuneGraphClient` to execute a query, then print the result. Copy the completed code below, which replaces the TODO with code and adds the necessary imports. Replace the graph identifier and query string as needed.

```
package org.example;

import software.amazon.awssdk.core.ResponseInputStream;
import software.amazon.awssdk.services.neptunegraph.NeptuneGraphClient;
import software.amazon.awssdk.services.neptunegraph.model.ExecuteQueryRequest;
import software.amazon.awssdk.services.neptunegraph.model.ExecuteQueryResponse;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;


public class Handler {
    private final NeptuneGraphClient neptuneGraphClient;

    public Handler() {
        neptuneGraphClient = DependencyFactory.neptuneGraphClient();
    }

    public void sendRequest() {
        String graphIdentifier = "g-0123456789";
        String queryString = "MATCH (n) RETURN n LIMIT 1";

        ExecuteQueryRequest request = ExecuteQueryRequest.builder()
                .graphIdentifier(graphIdentifier)
                .queryString(queryString)
                .build();

        System.out.println("Executing query: " + queryString);

        ResponseInputStream<ExecuteQueryResponse> response =
 neptuneGraphClient.executeQuery(request);
        BufferedReader reader = new BufferedReader(new
 InputStreamReader(response));

        try {
                System.out.println("Printing query result:");
                String line;
                while ((line = reader.readLine()) != null) {
                        System.out.println(line);
                }
        } catch (IOException e) {
                System.out.println("Error occurred while printing result.");
        }
    }
}
```

5. Navigate to your project directory `getStarted`.
6. Build your project: `mvn clean package`
7. Run the application: `mvn exec:java -Dexec.mainClass="org.example.App"`.

Using the AWS CLI

You can connect to your graph using the AWS command-line interface. Specify the neptune-graph service name to use Neptune Analytics APIs. For information on AWS CLI installation and usage, see the AWS CLI documentation. To set up credentials, refer to the AWS CLI user guide.

The example command below uses the AWS CLI to submit a query that returns a single node.

```
aws neptune-graph execute-query \
--graph-identifier g-0123456789 \
--region us-east-2 \
--query-string "MATCH (n) RETURN n LIMIT 1" \
--language open_cypher \
out.txt
```

Using AWSCURL

You can connect to your graph using the `awscurl` command-line tool. This allows you to directly make requests using HTTPS against a graph endpoint. You can find the correct endpoint to use in the AWS console (under the “Connectivity & Security” section of a Neptune Analytics graph page) and in the response of any `GetGraph` API request. To set up credentials, refer to the AWS CLI user guide.

For `awscurl` installation and setup instructions, see AWSCURL github repository.

The following command uses `awscurl` to submit a query that returns a single node.

```
awscurl -X POST "https://g-0123456789.us-east-2.neptune-graph.amazonaws.com/queries" \
-H "Content-Type: application/x-www-form-urlencoded" \
--region us-east-2 \
--service neptune-graph \
-d "query=MATCH (n) RETURN n LIMIT 1"
```

Best practices

Ensure the streams have been consumed and closed to be able to re-use client connections in the SDK. See the SDK for Java developer guide for more information.

CLI and SDK

Using the default settings, any CLI or SDK request will timeout in 60 seconds and attempt a retry. For the cases where you are running queries that can take longer than 60 seconds, it is recommended to set the CLI/SDK timeout to 0 (no timeout), or a much larger value to avoid unnecesssary retries. It is also recommended to set MAX_ATTEMPTS for CLI/SDK to 1 for `execute_query` to avoid any retries by the CLI/SDK. For the Boto client, set the `read_timeout` to None, and the `total_max_attempts` to 1.

```
import boto3
from botocore.config import Config
n = boto3.client('neptune-graph',
                 config=(Config(retries={"total_max_attempts": 1, "mode": "standard"},
 read_timeout=None)))
```

For the CLI, set the `--cli-read-timeout` parameter to `0` for no timeout, and set the environment variable AWS_MAX_ATTEMPTS to 1 to prevent retries.

```
export AWS_MAX_ATTEMPTS=1
aws neptune-graph execute-query \
--graph-identifier <graph-id> \
--region <region> \
--query-string "MATCH (p:Person)-[r:KNOWS]->(p1) RETURN *;" \
--cli-read-timeout 0
--language open_cypher /tmp/out.txt
```

Using notebooks with Neptune Analytics

The Neptune managed open-source graph-notebook project provides a plethora of Jupyter extensions and sample notebooks that make it easy to interact with and learn to use a Neptune Analytics graph.

These graph notebooks support a suite of intuitive Jupyter line- and cell-magic commands. The magic commands abstract away much of the initial setup typically required for using Neptune Analytics, and take care of SigV4 signing of requests. They can create graph connections, load data, run openCypher queries, and interact with various Neptune Analytics APIs.

You can find a list of the full set of Neptune graph-notebook magics and their options in the Neptune Userguide. However, only the following magics are compatible with Neptune Analytics graphs:

- %seed (adds sample data to a graph).
- %load (uses the `neptune-load()` openCypher integration to let you batch-load data).
- %status or %get_graph (gets status information about the graph).
- %%opencypher or %%oc (issues an openCypher query).
- %opencypher_status, or %oc_status (retrieves query status for, or cancels, an openCypher query).
- %%graph_notebook_config (displays a JSON object containing the configuration that the notebook is using).
- %graph_notebook_host (sets the line input as the notebook's host).
- %graph_notebook_version (returns the Neptune workbench notebook release number).
- %graph_notebook_service (sets the line input as the Neptune service name to use).
- %%graph_notebook_vis_options (lets you set visualization options for the notebook).
- %summary (retrieves graph summary information).
- %graph_reset (empties the data from a graph).

You can use a Neptune graph notebook to generate an interactive visualization of the results returned from an openCypher query, and use options to customize the appearance of the visualized graph (see Graph visualization in the Neptune workbench).

Take advantage of all the sample notebooks

A wide variety of sample Jupyter notebooks are available in the Neptune graph-notebook project. Some of these are purpose-built for learning how to get the most of a Neptune Analytics graph and its powerful built-in algorithms in the context of common real-world applications.

After installing the graph-notebook project either locally or on SageMaker, you should be able to find sample notebooks under the notebook directory, `../Neptune/02-Neptune-Analytics`.

Creating a new Neptune Analytics notebook using a AWS CloudFormation template

Amazon SageMaker Notebook instances provide a fully managed Jupyter environment for running graph notebooks that are connected to a Neptune Analytics graph. SageMaker Notebooks run natively on Amazon Linux 2, and support use of the Jupyter Classic Notebook or JupyterLab 3 interface on the same instance.

You can use one of the following AWS CloudFormation templates to set up a new Neptune Analytics notebook to use with your Neptune Analytics graph:

To use an AWS CloudFormation stack to create a new Neptune Analytics notebook

1. Choose one of the **Launch Stack** buttons in the following table to launch the AWS CloudFormation stack on the AWS CloudFormation console.

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 (Oregon)	View	View in Designer	Launch Stack
Europe (Ireland)	View	View in Designer	Launch Stack

Region	View	View in Designer	Launch
Europe (Frankfurt)	View	View in Designer	Launch Stack
Asia Pacific (Tokyo)	View	View in Designer	Launch Stack
Asia Pacific (Singapore)	View	View in Designer	Launch Stack

2. On the **Select Template** page, choose **Next**.
3. In the **Stack Details** page, under **GraphEndpoint**, enter the public or private endpoint of your Neptune Analytics graph.
4. Under **Notebook Name** enter a name for the new notebook that is unique for your account and region in SageMaker.
5. On the **Options** page, choose **Next**.
6. If you're using a private endpoint for your Neptune Analytics graph, enter the following under **Network Options**:

    a. Under **GraphVPC** enter the ID of a VPC associated with the private graph endpoint.

    b. Under **GraphSubnetId** enter the ID of any subnet associated with your private graph endpoint.

    c. Under **GraphSecurityGroup** enter the ID of a security group associated with the VPC. This is optional; if not provided, a new security group is automatically created for this purpose.
7. Click through the rest of the stack creation steps, leaving everything as default, and submit for creation.

In around 5 minutes, you should see the new Neptune Analytics notebook appear in the SageMaker and Neptune consoles.

Creating a new Neptune Analytics notebook using the AWS Management Console

You can create a new notebook manually using the AWS Management Console if you aren't able to use AWS CloudFormation. The first thing you need is an IAM role to use for the notebook. If you already have one, you can skip the following section.

Create an IAM role for a Neptune Analytics notebook

To create an IAM role for a Neptune Analytics notebook

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, expand **Access management**, then choose **Roles**.
3. Select **Create role**.
4. Under **Trusted entity type**, select **Custom trust policy** and copy in the following trust policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "sagemaker.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

5. Choose **Next**, and then **Next** again.
6. Enter a name and description for the role, and select **Create role**.
7. Go back to the **Roles** page, search for the name of the role you just created, and open it.
8. On the **Permissions** tab Under **Permissions policies**, select **Add permissions** and choose **Create inline policy**.
9. In the **Policy editor**, switch to the **JSON** option, and copy in the following policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::aws-neptune-notebook-(AWS region)",
        "arn:aws:s3:::aws-neptune-notebook-(AWS region)/*",
        "arn:aws:s3:::aws-neptune-customer-samples-(AWS region)",
        "arn:aws:s3:::aws-neptune-customer-samples-(AWS region)/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "neptune-graph:*",
      "Resource": [
        "arn:aws:neptune-graph:(AWS region):(AWS account ID):graph/(Neptune Graph resource ID)"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": [
        "arn:aws:logs:*:*:log-group:/aws/sagemaker/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "sagemaker:DescribeNotebookInstance",
      "Resource": [
        "arn:aws:sagemaker:(AWS region):(AWS account ID):notebook-instance/*"
      ]
    }
  ]
}
```

10. Choose **Next**.
11. Give a name to the inline policy.
12. Select **Create policy**. Make note of the name of the policy you just created.

Next, create the Neptune Analytics notebook in SageMaker

1. Open the Amazon SageMaker console at https://console.aws.amazon.com/sagemaker/.
2. In the navigation pane, expand **Notebook**, then choose **Notebook instances**.
3. Choose **Create notebook instance**.
4. In **Notebook instance settings**, under **Notebook instance name**, give the notebook a name prefixed by `aws-neptune-` (for example, `aws-neptune-my-test-notebook`).
5. Under **Platform identifier**, select **Amazon Linux 2, JupyterLab 3**.
6. Select **Additional configuration**.
7. Under **Lifecycle configuration**, choose **Create a new lifecycle configuration**.
8. In **Configuration**, under **Name** enter the notebook instance name from step 4.
9. In **Scripts**, under **Start notebook**, replace the existing script with this:

```
#!/bin/bash

sudo -u ec2-user -i <<'EOF'

echo "export GRAPH_NOTEBOOK_AUTH_MODE=IAM" >> ~/.bashrc
echo "export GRAPH_NOTEBOOK_SSL=True" >> ~/.bashrc
echo "export GRAPH_NOTEBOOK_SERVICE=neptune-graph" >> ~/.bashrc
echo "export GRAPH_NOTEBOOK_HOST=(Neptune Analytics graph endpoint, public or
 private)" >> ~/.bashrc
echo "export GRAPH_NOTEBOOK_PORT=8182" >> ~/.bashrc
echo "export NEPTUNE_LOAD_FROM_S3_ROLE_ARN=" >> ~/.bashrc
echo "export AWS_REGION=(AWS region)" >> ~/.bashrc

aws s3 cp s3://aws-neptune-notebook-(AWS region)/graph_notebook.tar.gz /tmp/
graph_notebook.tar.gz
rm -rf /tmp/graph_notebook
tar -zxvf /tmp/graph_notebook.tar.gz -C /tmp
/tmp/graph_notebook/install.sh
```

```
EOF
```

10. Select **Create configuration**.
11. In **Permissions and encryption**, under **IAM Role**, select the role you created above.
12. In **Network**, if you are using a private graph endpoint:

    a. Under **VPC**, select the VPC where the Neptune Analytics graph resides.

    b. Under **Subnet**, select a subnet associated with the Neptune Analytics graph.

    c. Under **Security Group(s)**, select all the security groups associated with the Neptune Analytics graph.
13. Choose **Create notebook instance**.
14. After 5 or 10 minutes, when your new notebook reaches `Ready` status, select it. Choose **Open Jupyter** or **Open JupyterLab**.

Hosting a Neptune Analytics graph-notebook on your local machine

It is also possible to install and run a Neptune Analytics graph notebook on your local machine. You can find instructions in the GitHub graph-notebook repository:

- Prerequisites
- Jupyter Classic Notebook or https://github.com/aws/graph-notebook/#jupyterlab-3x installation
- Connecting to Neptune

When setting up for Neptune Analytics:

- When setting the connection using %%graph_notebook_config, make sure to set the `neptune_service` field to the value `neptune-graph`.
- If you're connecting to a private graph endpoint, you need to enable access to the VPC where the Neptune Analytics instance resides. The easiest way to set this is up is using an SSH tunnel to a proxy EC2 instance in the VPC. For more information, see Connecting graph notebook locally to Amazon Neptune in GitHub.
- If you're using a public graph endpoint, no additional connectivity setup is required.

Creating a new Neptune Analytics graph using the AWS Management Console

You can use the Neptune console to create a new Neptune Analytics graph.

> **Note**
>
> If you are working with a large dataset (on the order of 50 GiB or larger) that you intend to load at the same time the new graph is created, be sure to create an IAM role that grants permissions to load the dataset from the location where it resides.

To use the Neptune console to create a graph

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/.
2. In the navigation pane, select **Graphs** under **Analytics**.
3. Select **Create graph**.
4. Enter a name for the new graph.
5. The next steps depend on whether you are creating an empty graph or one preloaded with data.
   - If you choose **Create empty graph**, choose the number of memory-optimized Neptune Capacity Units (m-NCUs) to allocate to the new Neptune Analytics graph, between 128 and 1024. Each m-NCU has around one GiB of memory capacity and corresponding compute and networking.
   - If you choose **Create Graph from existing source**, Neptune Analytics will bulk-load data for you when the graph is created. Choose this option if you want to import a large dataset, on the order of 50 GiB or larger. See Bulk import for details.

     a. Set values for the minimum and maximum m-NCUs, or just leave them at their default values (128 m-NCUs). The units are memory-optimized Neptune Capacity Units (m-NCUs), each of which is roughly equivalent to 1 GiB of memory and corresponding compute and networking. Neptune Analytics evaluates the data that you want to load and estimates the resources needed to handle it, within the range of m-NCUs that you specify.

   b. Under **Load role ARN**, select an IAM role that you have created to provide the necessary permissions for the data import. See Create an IAM role with permissions to export from Neptune to Neptune Analytics for instructions about how to create the role.

   c. The next steps depend on what source you're loading data from:

      - If you choose **Neptune cluster** as the type of source, select the DB cluster that you want to load from under **Neptune cluster**.
      - If you choose **Neptune cluster snapshot** as the type of source, select one of your manual DB snapshots that you want to load from under **Neptune DB snapshot**.
      - If you choose **S3** as the type of source, enter the URL of the Amazon S3 location where the data file(s) to be loaded are located, under **Resource URI**. The path to the folder location must end in a slash rather than specify to a particular file.

6. Under **Availability settings**, choose how many failover replicas you want to create for the new graph. The default is one, but if you select Use custom number of replicas you can choose from zero to two failover replicas.

   > ⚠ **Important**
   >
   > Additional charges equivalent to the m-NCUs selected for the graph apply for each replica.

7. Under **Network and security**, check **Allow from public** to create a public endpoint for your new Neptune Analytics graph to make it accessible over the internet. If you want to use your own KMS key to encrypt your data, check **Customize encryption settings** a specify a KMS key of your choosing.
8. Under **Vector search settings**, if you want to set up a vector index for the graph, choose **Use vector dimension** and then specify the number of dimensions for the vectors in the index.
9. Under **Advanced settings**, you can make it easier to delete your new graph by selecting **Turn off deletion protection**. Deletion protection is turned on by default.
10. Finally, under **Tags**, you can associate tags with your new Neptune Analytics graph.
11. When everything is configured as you want it to be, choose **Create Graph**.

Loading data into a Neptune Analytics graph

Neptune Analytics provides several options for loading data into a graph, supporting both RDF (Resource Description Framework) and LPG (Labeled Property Graph) models.

- **Bulk import** – Designed to handle large scale data ingestion and is the fastest way to load large volumes of data. Bulk import runs a task to load data from files in Amazon S3. This option **must** be done on an empty graph, either at creation time using the CreateGraphUsingImportTask action, or on an existing graph using the StartImportTask action.
- **Batch load** – Designed to handle incremental data ingestion to existing graphs using files in Amazon S3. This can be used to add more data or update single cardinality property values in existing graph data. The volume of data that can be ingested in a single request is lower than what bulk import can support.
- **openCypher queries** – Add more data through queries, if data is not available from files in Amazon S3 or the data volume is small. This is also a more generic approach for conditional inserts based on data already in the graph, and updating contents of the graph.

> ⚠ **Warning**
>
> Be cautious while loading a file of edges. If the same edge file is loaded twice, duplicate edges will be inserted into the graph which can lead to unintended results.
> Also, while using the SDK/CLI command execute-query to run neptune.load(), it is recommended to increase the timeout window and disable the retries for the SDK/CLI.
> For more information about increasing the timeout and disabling retries, see ExecuteQuery.

Topics

Data format for loading from Amazon S3 into Neptune Analytics

Neptune Analytics, just like Neptune Database, supports three formats for loading data:

- RDF (ntriples), which is a line-based format for triples. See Using RDF data for more information on how this data is handled.
- csv and opencypher, which are csv-based formats with schema restrictions. A csv file must contain a header row and the column values. The remainder of the files are interpreted based on the corresponding header column. The header could contain predefined system column names and user-defined column names annotated with predefined datatypes and cardinality.

It's possible to combine RDF data with csv data in the same graph. For example, you could first load csv data and later enrich it with RDF data.

Differences with Neptune csv (opencypher) format

Edge files:

- The `~id` (`:ID`) column in `edge` (`relationship`) files in `CSV` (`opencypher`) format are not supported. They are ignored if provided in any of the `edge` (`relationship`) files.

Vertex files:

- Only explicitly provided labels are associated with the vertices. If the label provided is empty, the vertex would be added without a label. If a row contains just the vertex id without any labels or properties then the row is ignored, and no vertex is added. For more information about vertices, see vertices.
- A new column type `Vector` is supported for associating embeddings with vertices. Since Neptune Analytics only supports one index type at this moment, the property name for embeddings is currently fixed to `Embeddings`. If the element type of the embedding are not floating point (FP32), there will be typecasted to FP32. The embeddings in the `csv` files are optional when the vector index is enabled. This means that not every node needs to be associated with an embedding. If you want to set up a vector index for the graph, choose use `vector dimension` and then specify the number of dimensions for the vectors in the index. Note that the `dimension` must match the dimension of the embeddings in the vertex files. For more details of loading embeddings, refer to vector-index.

Edge or vertex files:

- Unlike Neptune Database, a vertex identifier could appear just in edge files. Neptune Analytics allows loading just the edge data from files in Amazon S3, and running an algorithm over the data without needing to provide any additional vertex information. The edges are created between vertices with the given identifiers, and the vertices have no labels or properties unless any are provided in the vertex files. For more information on vertices and what they are, see vertices.
- `Date` column type is not supported. The `datetime` column type is supported and could be used instead. Pure date values in the `datetime` column are implicitly assigned `00:00:00z` time.
- The datetime values can either be provided in 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`
- Float and double values in scientific notation are currently not supported. Also, `Infinity`, `INF`, `-Infinity`, `-INF`, and `NaN` (`Not-a-number`) values are not recognized.
- `Gzip` files are not supported.
- The maximum length of the strings supported is smaller, and limited to 1,048,062 bytes. The limit would be lower for strings with unicode characters since some unicode characters are represented using multiple bytes.
- Multi-line string values are not supported. Imports behavior is undefined if the dataset contains multi-line string values.
- Quoted string values must not have leading space between the delimiter and quotes. For example, if a line is `abc, “def”` then that is interpreted as line with two fields with string values `abc` and `“def”`. `“def"` is a non-quoted string field and quotes are stored as is in the value, with a size of 6 characters. If the line is `abc,“def”` then it is interpreted as a line with two fields with string values `abc` and `def`.

Using RDF data

Neptune Analytics supports importing RDF data using the n-triples format. With this support it is possible to load CSV and n-triples data files into the same graph. The handling of RDF values is described below, including how RDF data is interpreted as LPG concepts and can be queried using openCypher.

Handling of RDF values

The handling of RDF specific values, that don‘t have a direct equivalent in LPG, is described here.

IRIs

Values of type IRI, like `<http://example.com/Alice>` , are stored as such. IRIs and Strings are distinct data types.

Calling openCypher function `TOSTRING()` on an IRI returns a string containing the IRI wrapped inside <>. For example, if `x` is the IRI `<http://example.com/Alice>`, then `TOSTRING(x)` returns `"<http://example.com/Alice>"`. When serializing openCypher query results in json format, IRI values are included as strings in this same format.

Language-tagged literals

Values like `"Hallo"@de` are treated as follows:

- When used as input for openCypher string functions, like `trim()`, a language-tagged string is treated as a simple string; so `trim("Hallo"@de)` is equivalent to `trim("Hallo")`.
- When used in comparison operations, like `x = y` or `x <> y` or `x < y` or `ORDER BY`, a language-tagged literal is “greater than” (and thus “not equal to”) the corresponding simple string: `"Hallo" < "Hallo"@de`.

Calling a function, such as `TOSTRING()` on a language-tagged literal, returns that literal as a string without language tag. For example, if `x` is the value `"Hallo"@de`, then `TOSTRING(x)` returns `"Hallo"`. When serializing openCypher query results in JSON format, language-tagged literals are also serialized as strings without an associated language tag.

Blank nodes

Blank nodes in n-triples data files are replaced with globally unique IRIs at import time.

Loading RDF datasets that contains blank nodes is supported; but those blank nodes are represented as IRIs in the graph. When loading ntriples files the parameter `blankNodeHandling` needs to be specified, with the value `convertToIri`.

The generated IRI for a blank node has the format: `<http://aws.amazon.com/neptune/vocab/v01/BNode/scope#id>`

In these IRIs, `scope` is a unique identifier for the blank node scope, and `id` is the blank node identifier in the file. For example for a blank node `_:b123` the generated IRI could be `<http://aws.amazon.com/neptune/vocab/v01/BNode/737c0b5386448f78#b123>`.

The **blank node scope** (e.g. 737c0b5386448f78) is generated by Neptune Analytics and designates one file within one load operation. This means that when two different ntriples files reference the same blank node identifier, like `_:b123`, there will be two IRIs generated, namely one for each file. All references to `_:b123` within the first file will end up as references to the first IRI, like `<http://aws.amazon.com/neptune/vocab/v01/BNode/1001#b123>`, and all references within the second file will end up referring to another IRI, like `<http://aws.amazon.com/neptune/vocab/v01/BNode/1002#b123>`.

Referencing IRIs in queries

There are two ways to reference an IRI in an openCypher query:

- Wrap the full IRI inside < and > . Depending on where in the query this IRI is referenced, the IRI is then provided as a String, such as `"<http://example.com/Alice>"` (when the IRI is the value of property `~id`), or in backticks such as `` `<http://example.com/Alice>` `` (when the IRI is a label, or property key).

```
CREATE (:`<http://xmlns.com/foaf/0.1/Person>` {`~id`: "<http://example.com/Alice>"})
```

- Define a PREFIX at the start of the query, and inside the query reference an IRI using `prefix::suffix`. For example, after `PREFIX ex: <http://example.com/>` the reference `ex::Alice` also references the full IRI `<http://example.com/Alice>`.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.com/>
CREATE (: foaf::Person {`~id`: ex::Alice})
```

Additional query examples below show the use of both full IRIs and the prefix syntax.

Mapping RDF triples to LPG concepts

There are three rules that define how RDF triples correspond to LPG concepts:

```
Case     RDF triple                  #    LPG concept
-----------------------------------------------------------------
Case #1  { <iri> rdf:type <iri> }    #    vertex with id + label
Case #2  { <iri> <iri> "literal"}    #    vertex property
Case #3  { <iri> <iri> <iri> }       #    edge with label
```

Case #1: Vertex with id and label

A triple like:

```
<http://example.com/Alice> rdf:type <http://xmlns.com/foaf/0.1/Person>
```

is equivalent to creating the vertex in openCypher like:

```
CREATE (:`<http://xmlns.com/foaf/0.1/Person>` {`~id`: "<http://example.com/Alice>"})
```

In this example, the vertex label `<http://xmlns.com/foaf/0.1/Person>` is interpreted and stored as an IRI.

> **Note**
>
> The back quote syntax `` ` ` `` is part of openCypher which allows inserting characters that normally cannot be used in labels. Using this mechanism, it's possible to include complete IRIs in a query.

Using `PREFIX`, the same `CREATE` query could look like:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.com/>
CREATE (: foaf::Person {`~id`: ex::Alice})
```

To match the newly created vertex based on its id:

```
MATCH (v {`~id`: "<http://example.com/Alice>"}) RETURN v
```

or equivalently:

```
PREFIX ex: <http://example.com/>
MATCH (v {`~id`: ex::Alice}) RETURN v
```

To find vertices with that RDF Class/LPG Label:

```
MATCH (v:`<http://xmlns.com/foaf/0.1/Person>`) RETURN v
```

or equivalently:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
MATCH (v : foaf::Person) RETURN v
```

Case #2: Vertex property

A triple like:

```
<http://example.com/Alice> <http://xmlns.com/foaf/0.1/name> "Alice Smith"
```

is equivalent to defining with openCypher node with a given `~id` and property, where both the `~id` and the property key are IRIs:

```
CREATE ({`~id`: "<http://example.com/Alice>",
        `<http://xmlns.com/foaf/0.1/name>`: "Alice Smith" })
```

or equivalently:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.com/>
CREATE ({`~id`: ex::Alice, foaf::name: "Alice Smith" })
```

To match the vertex with that property:

```
MATCH (v {`<http://xmlns.com/foaf/0.1/name>`: "Alice Smith"}) RETURN v
```

or equivalently:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

```
MATCH (v { foaf::name : "Alice Smith"}) RETURN v
```

Case #3: Edge

A triple like:

```
<http://example.com/Alice> <http://example.com/knows> <http://example.com/Bob>
```

is equivalent to defining with OpenCypher an edge like this, where the edge label and vertices ids are all IRIs:

```
CREATE ({`~id`: "<http://example.com/Alice>"})
          -[:`<http://example.com/knows>`]->({`~id`: "<http://example.com/Bob>"})
```

or equivalently:

```
PREFIX ex: <http://example.com/>
CREATE ({`~id`: ex::Alice })-[: ex::knows ]->({`~id`: ex::Bob })
```

To match the edges with that label:

```
MATCH (v)-[:`<http://example.com/knows>`]->(w) RETURN v, w
```

or equivalently:

```
PREFIX ex: <http://example.com/>
MATCH (v)-[: ex::knows ]->(w) RETURN v, w
```

Query Examples

Matching language-tagged literals

If this triple was loaded from a dataset:

```
<http://example.com/German> <http://example.com/greeting> "Hallo"@de
```

then it will **not** be matched by this query:

```
MATCH (n) WHERE n.`<http://example.com/greeting>` = "Hallo"
```

because the language-tagged literal "Hallo"@de and the string “Hallo” are not equal. For more information, see Language-tagged literals. The query can use TOSTRING( ) in order to find the match:

```
MATCH (n) WHERE TOSTRING(n.`<http://example.com/greeting>`) = "Hallo"
```

Batch load

Neptune Analytics supports a CALL procedure neptune.load to load data from Amazon S3, to insert new vertices, edges, and properties, or to update single cardinality vertex property values. It executes as a mutation query and does atomic writes. It uses the IAM credentials of the caller to access the data in Amazon S3. See Create your IAM role for Amazon S3 access to set up the permissions.

Request syntax

The signature of the CALL procedure is shown below:

```
CALL neptune.load(
  {
    source: "string",
    region: "us-east-1",
    format: "csv",
    failOnError: true,
    concurrency: 1
  }
)
```

- **source** (required) – An Amazon S3 URI prefix. All object names with matching prefixes are loaded. See Neptune Database loader reference for Amazon S3 URI prefix examples. The IAM user who signs the openCypher request must have permissions to list and download these objects, and must be authorized for WriteDataViaQuery and DeleteDataViaQuery actions. See IAM role mapping for more IAM authentication related details.
- **region** (required) – The AWS region where the Amazon S3 bucket is hosted. Currently, cross-region loads are not supported.
- **format** (required) – The data format of the Amazon S3 data to be loaded, either csv, opencypher, or ntriples. For more information, see Data format for loading from Amazon S3 into Neptune Analytics.

- **blankNodeHanding**(must be provided when format is `ntriples`) - The method to handle blank nodes in the dataset. Currently, only `convertToIri` is supported, meaning blank nodes are converted to unique IRIs at load time. For more information, see [Handling RDF values](#).
- **failOnError** (optional) default: true – If set to `true` (the default), the load process halts whenever there is an error parsing or inserting data. If set to `false`, the load process continues and commits whatever data was successfully inserted.

  The edge or relationship data should be loaded with `failOnError` set to `true`, to avoid duplication of partially committed edges or relationships in subsequent loads.
- *concurrency* (optional) default: 1 – This value controls the number of threads used to run the load process, up to the maximum available.

> **Note**
>
> Unlike bulk import, there is no need to pass the `role-arn` for batch load since the IAM credentials of the signer of the openCypher query are used to download data from Amazon S3. The signer must have permissions to download data from Amazon S3 with the trust relationship set up to assume the role, so that Neptune Analytics can assume the role to load the data into the graph from files in Amazon S3.

Response syntax

A sample response is shown below.

```
{
    "results": [
        {
            "totalRecords": 108070,
            "totalDuplicates": 46521,
            "totalTimeSpentMillis": 558,
            "numThreads": 16,
            "insertErrors": 0,
            "throughputRecordsPerSec": 193673,
            "loadId": "13a60c3b-754d-c49b-4c23-06b9dd5b346b"
        }
    ]
}
```

- `totalRecords`: The number of graph elements - vertex labels, edges, and properties - attempted for insertion.
- `totalDuplicates`: The count of duplicate graph elements - vertex labels or properties - encountered. These elements may have pre-existed before the load request or were duplicates within the input CSV files. Each edge is treated as new, so edges are excluded from this count.
- `totalTimeSpentMillis`: The total time taken for downloading, parsing, and inserting data from CSV files, excluding the request queue time.
- `numThreads`: The number of threads utilized for downloading and inserting data. This correlates with the provided `concurrency` parameter input, reflecting any caps applied.
- `insertErrors`: Errors faced during insertions, including parsing errors and Amazon S3 access issues. Error details are available in the CloudWatch logs. Refer to the Troubleshooting section of this document to understand troubleshooting insertErrors. Concurrent modification errors may also cause insert errors in batch loads attempting to modify a vertex property value being concurrently changed by another request.
- `throughputRecordsPerSec`: The total throughput in records per second.
- `loadId`: The loadId for searching errors and load summary. All batch information is published to CloudWatch logs under `/aws/neptune/import-task-logs/<graph-id>/<load-id>`.

Note

Around 2.5Gb of Amazon S3 files can be loaded in a single request on 128 m-NCU. Larger sized datasets could run into `out of memory` errors. To workaround that, the Amazon S3 files can be split across multiple serial batch load requests. The source argument takes a prefix, so files can be partitioned across requests by including prefixes of file names. The limit scales linearly based on m-NCUs, so for example 5Gb of Amazon S3 files can be loaded in a single request on 256 m-NCU. Also, if the dataset contains larger string values for example, then larger volumes of data can also be ingested in a single request, since they would generate fewer number of graph elements per byte of dataset. It is recommended to run tests with your data to determine the exact details for this process.

Important

Duplicate edges get created if the same edge file content is loaded more than once. This could happen if, for example:

1. The same Amazon S3 source or file is accidentally included for load in more than one request that succeeded.
2. The edge data is first loaded with `failOnError` set to false and runs into partial errors, and the errors are fixed and the entire dataset is reloaded. All of the edges that were successfully inserted on the first request would get duplicated after the second request.

Bulk import data into a graph

The task system in Neptune Analytics provides a powerful and flexible way to bulk import data into your graph. The `import` task is specifically designed to handle large-scale data ingestion from various data formats.

To initiate a bulk data import, you would first create an import task by specifying the data source, the target graph, and any necessary configuration options. This can be done through the AWS console or programmatically via the API.

Throughout the import process, you can monitor the progress of the import task through the user interface or via API calls. Progress reports, and any potential errors or warnings will be accessible in your CloudWatch account, allowing for close monitoring and troubleshooting if needed.

Importing of data through Import Task is supported in two ways:

- During graph creation: Create a graph from Amazon S3, a Neptune cluster, or a snapshot
- On an existing empty graph: Bulk import data into an existing Neptune Analytics graph

Create a graph from Amazon S3, a Neptune cluster, or a snapshot

You can create a Neptune Analytics graph directly from Amazon S3 or from Neptune using the CreateGraphUsingImportTask API. This is recommended for importing large graphs from files in Amazon S3 (>50GB of data), importing from existing Neptune clusters, or importing from existing Neptune snapshots. This API automatically analyzes the data, provisions a new graph based on the analysis, and imports data as one atomic operation using maximum available resources.

> **Note**
>
> The graph is made available for querying only after the data loading is completed successfully.

If errors are encountered during the import process, Neptune Analytics will automatically roll back the provisioned resources, and perform the cleanup. No manual cleanup actions are needed. Error details are available in the CloudWatch logs. See troubleshooting for more details.

Topics

Creating a Neptune Analytics graph from Amazon S3

Neptune Analytics supports bulk importing of CSV or ntriples data directly from Amazon S3 into a Neptune Analytics graph using the `CreateGraphUsingImportTask` API. The data formats supported are listed in Data format for loading from Amazon S3 into Neptune Analytics. It is recommended that you try the batch load process with a subset of your data first to validate that it is correctly formatted. Once you have validated that your data files are fully compatible with Neptune Analytics, you can prepare your full dataset and perform the bulk import using the steps below.

A quick summary of steps needed to import a graph from Amazon S3:

- Copy the data files to an Amazon S3 bucket: Copy the data files to an Amazon Simple Storage Service bucket in the same region where you want the Neptune Analytics graph to be created. See Data format for loading from Amazon S3 into Neptune Analytics for the details of the format when loading data from Amazon S3 into Neptune Analytics.
- Create your IAM role for Amazon S3 access: Create an IAM role with `read` and `list` access to the bucket and a trust relationship that allows Neptune Analytics graphs to use your IAM role for importing.
- Use the `CreateGraphUsingImportTask` API to import from Amazon S3: Create a graph using the `CreateGraphUsingImportTask` API. This will generate a `taskId` for the operation.
- Use the `GetImportTask` API to get the details of the import task. The response will indicate the status of the task (ie. INITIALIZING, ANALYZING_DATA, IMPORTING etc.).

- Once the task has completed successfully, you will see a `COMPLETED` status for the import task and also the `graphId` for the newly created graph.
- Use the `GetGraphs` API to fetch all the details about your new graph, including the ARN, endpoint, etc.

Note

If you're creating a private graph endpoint, the following permissions are required:

- ec2:CreateVpcEndpoint
- ec2:DescribeAvailabilityZones
- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets
- ec2:DescribeVpcAttribute
- ec2:DescribeVpcEndpoints
- ec2:DescribeVpcs
- ec2:ModifyVpcEndpoint
- route53:AssociateVPCWithHostedZone

For more information about required permissions, see [Actions defined by Neptune Analytics](https://docs.aws.amazon.com/service-authorization/latest/reference/list_amazonneptuneanalytics.html).

Copy the data files to an Amazon S3 bucket

The Amazon 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.

```
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. In the command
>
> ```
> aws s3 cp datafile.txt s3://examplebucket/mydirectory/datafile.txt
> ```
>
> the object key name is `mydirectory/datafile.txt`

You can also use the AWS management console to upload files to the Amazon S3 bucket. Open the Amazon S3 console, and choose a bucket. In the upper-left corner, choose **Upload** to upload files.

Create your IAM role for Amazon S3 access

Create an IAM role with permissions to `read` and `list` the contents of your bucket. Add a trust relationship that allows Neptune Analytics to assume this role for doing the import task. You could do this using the AWS console, or through the CLI/SDK.

1. Open the IAM console at https://console.aws.amazon.com/iam/. Choose **Roles**, and then choose **Create Role**.
2. Provide a role name.
3. Choose **Amazon S3** as the AWS service.
4. In the **permissions** section, choose `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 Amazon S3 bucket is encrypted, you

need to add `kms:Decrypt` permissions as well. `kms:Decrypt` permission is needed for the exported data from Neptune Database

5. On the **Trust Relationships** tab, choose **Edit trust relationship**, and paste the following trust policy. Choose **Save** to save the trust relationship.

```
{
      "Version": "2012-10-17",
      "Statement": [
          {
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "neptune-graph.amazonaws.com"
                  ]
              },
              "Action": "sts:AssumeRole"
          }
      ]
  }
```

Your IAM role is now ready for import.

Use the CreateGraphUsingImportTask API to import from Amazon S3

You can perform this operation from the Neptune console as well as from AWS CLI/SDK. For more information on different parameters, see https://docs.aws.amazon.com/neptune-analytics/latest/apiref/API_CreateGraphUsingImportTask.html

Via CLI/SDK

```
aws neptune-graph create-graph-using-import-task \
  --graph-name <name> \
  --format <format> \
  --source <s3 path> \
  --role-arn <role arn> \
  [--blank-node-handling "convertToIri"--] \
  [--fail-on-error | --no-fail-on-error] \
  [--deletion-protection | --no-deletion-protection]
  [--public-connectivity | --no-public-connectivity]
  [--min-provisioned-memory]
  [--max-provisioned-memory]
  [--vector-search-configuration]
```

- **Different Minimum and Maximum Provisioned Memory**: When the `--min-provisioned-memory` and `--max-provisioned-memory` values are specified differently, the graph is created with the maximum provisioned memory specified by `--max-provisioned-memory`.
- **Single Provisioned Memory Value**: When only one of `--min-provisioned-memory` or `--max-provisioned-memory` is provided, the graph is created with the specified memory value.
- **No Provisioned Memory Values**: If neither `--min-provisioned-memory` nor `--max-provisioned-memory` is provided, the graph is created with a default provisioned memory of 128 m-NCU (memory optimized Neptune Compute Units).

Example 1: Create a graph from Amazon S3, with no min/max provisioned memory.

```
aws neptune-graph create-graph-using-import-task \
  --graph-name 'graph-1' \
  --source "s3://bucket-name/gremlin-format-dataset/" \
  --role-arn "arn:aws:iam::<account-id>:role/<role-name>" \
  --format CSV
```

Example 2: Create a graph from Amazon S3, with min & max provisioned memory. A graph with m-NCU of 1024 is created.

```
aws neptune-graph create-graph-using-import-task \
  --graph-name 'graph-1' \
  --source "s3://bucket-name/gremlin-format-dataset/" \
  --role-arn "arn:aws:iam::<account-id>:role/<role-name>" \
  --format CSV
  --min-provisioned-memory 128 \
  --max-provisioned-memory 1024
```

Example 3: Create a graph from Amazon S3, and not fail on parsing errors.

```
aws neptune-graph create-graph-using-import-task \
  --graph-name 'graph-1' \
  --source "s3://bucket-name/gremlin-format-dataset/" \
  --role-arn "arn:aws:iam::<account-id>:role/<role-name>" \
  --format CSV
  --no-fail-on-error
```

Example 4: Create a graph from Amazon S3, with 2 replicas.

```
aws neptune-graph create-graph-using-import-task \
  --graph-name 'graph-1' \
  --source "s3://bucket-name/gremlin-format-dataset/" \
  --role-arn "arn:aws:iam::<account-id>:role/<role-name>" \
  --format CSV
  --replica-count 2
```

Example 5: Create a graph from Amazon S3 with vector search index.

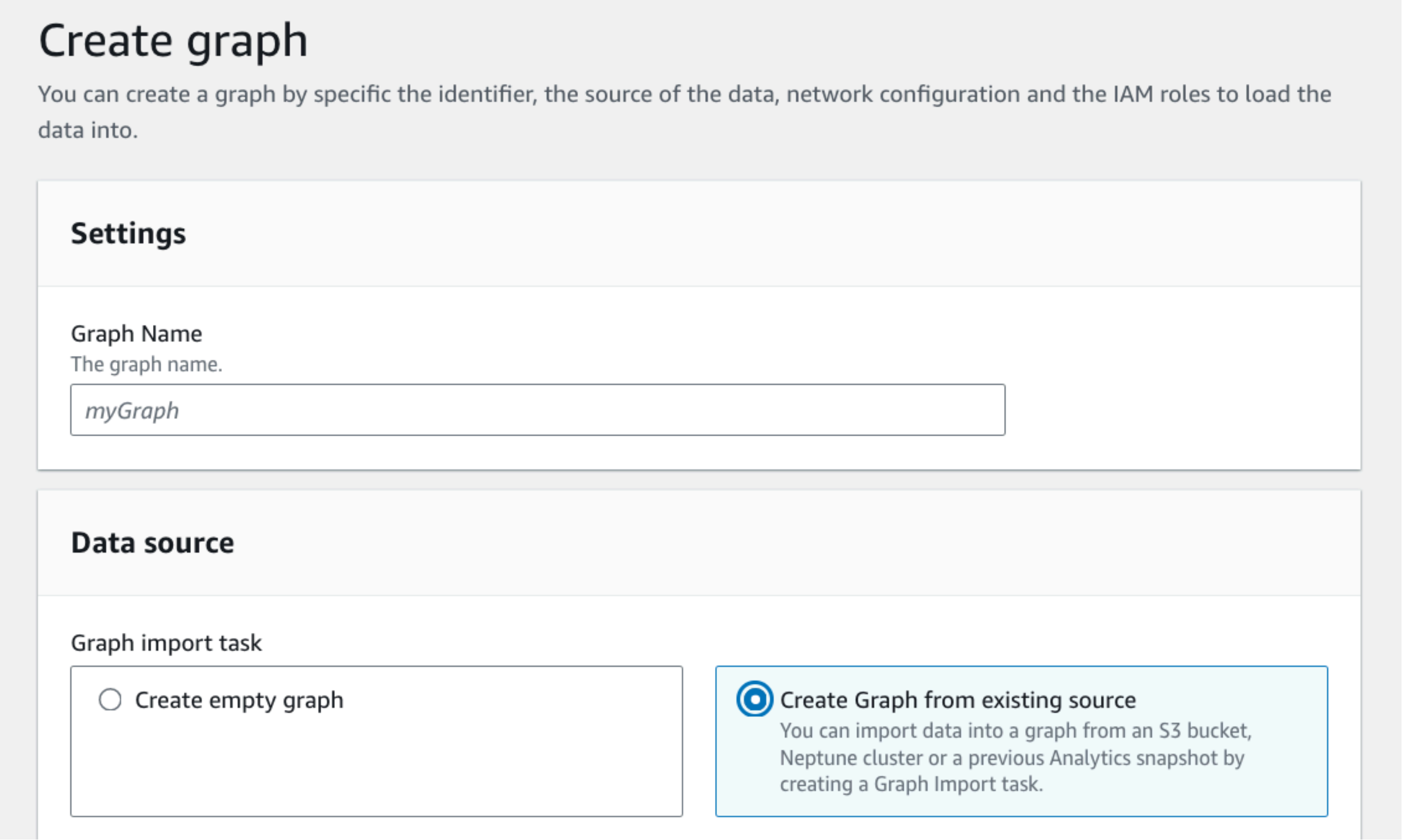
Note

The `dimension` must match the dimension of the embeddings in the vertex files.

```
aws neptune-graph create-graph-using-import-task \
  --graph-name 'graph-1' \
  --source "s3://bucket-name/gremlin-format-dataset/" \
  --role-arn "arn:aws:iam::<account-id>:role/<role-name>" \
  --format CSV
  --replica-count 2 \
  --vector-search-configuration "{\"dimension\":768}"
```

Via Neptune console

1. Start the Create Graph wizard and choose **Create graph from existing source**.



Create graph

You can create a graph by specific the identifier, the source of the data, network configuration and the IAM roles to load the data into.

Settings

Graph Name

The graph name.

myGraph

Data source

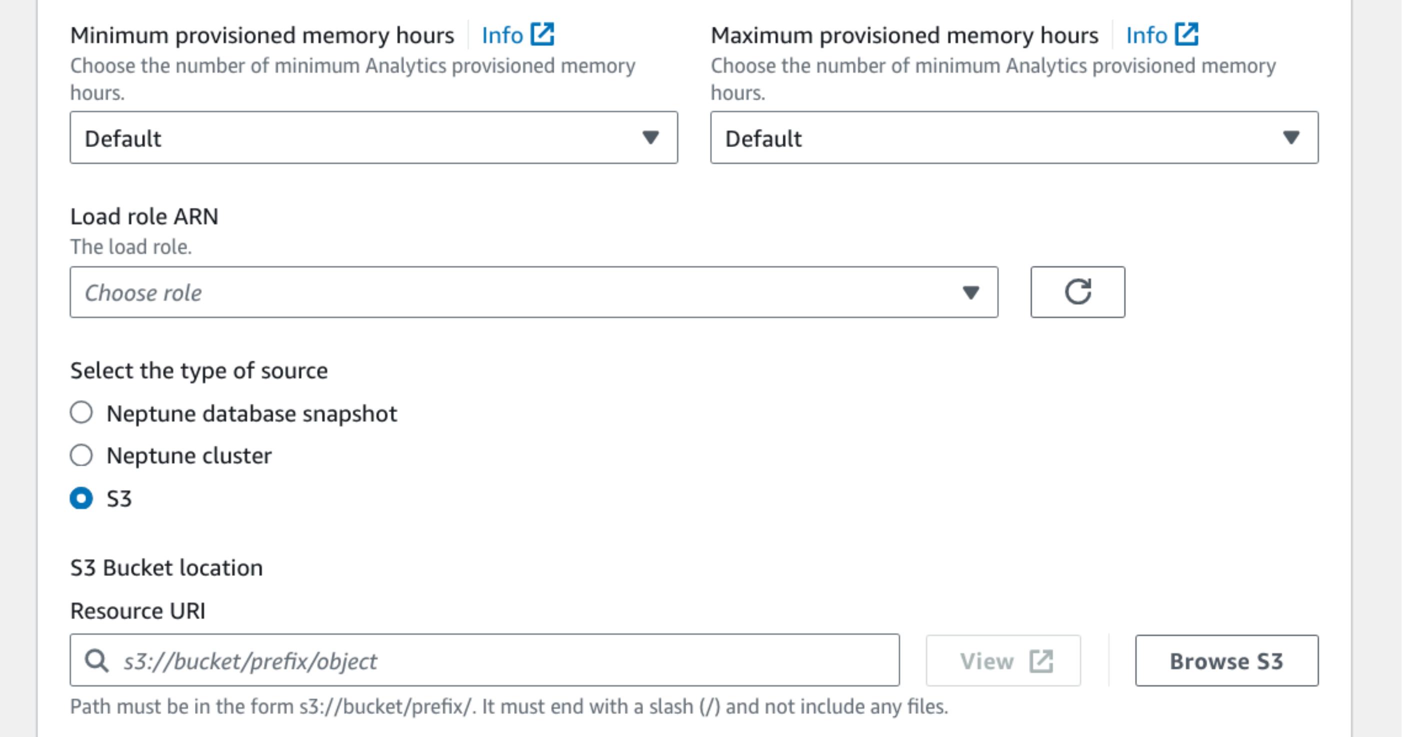
Graph import task

○ Create empty graph

● Create Graph from existing source

You can import data into a graph from an S3 bucket, Neptune cluster or a previous Analytics snapshot by creating a Graph Import task.

2. Choose type of source as Amazon S3, minimum and maximum provisioned memory, Amazon S3 path, and load role ARN.



Minimum provisioned memory hours

Choose the number of minimum Analytics provisioned memory hours.

Default

Maximum provisioned memory hours

Choose the number of minimum Analytics provisioned memory hours.

Default

Load role ARN

The load role.

Choose role

Select the type of source

○ Neptune database snapshot

○ Neptune cluster

● S3

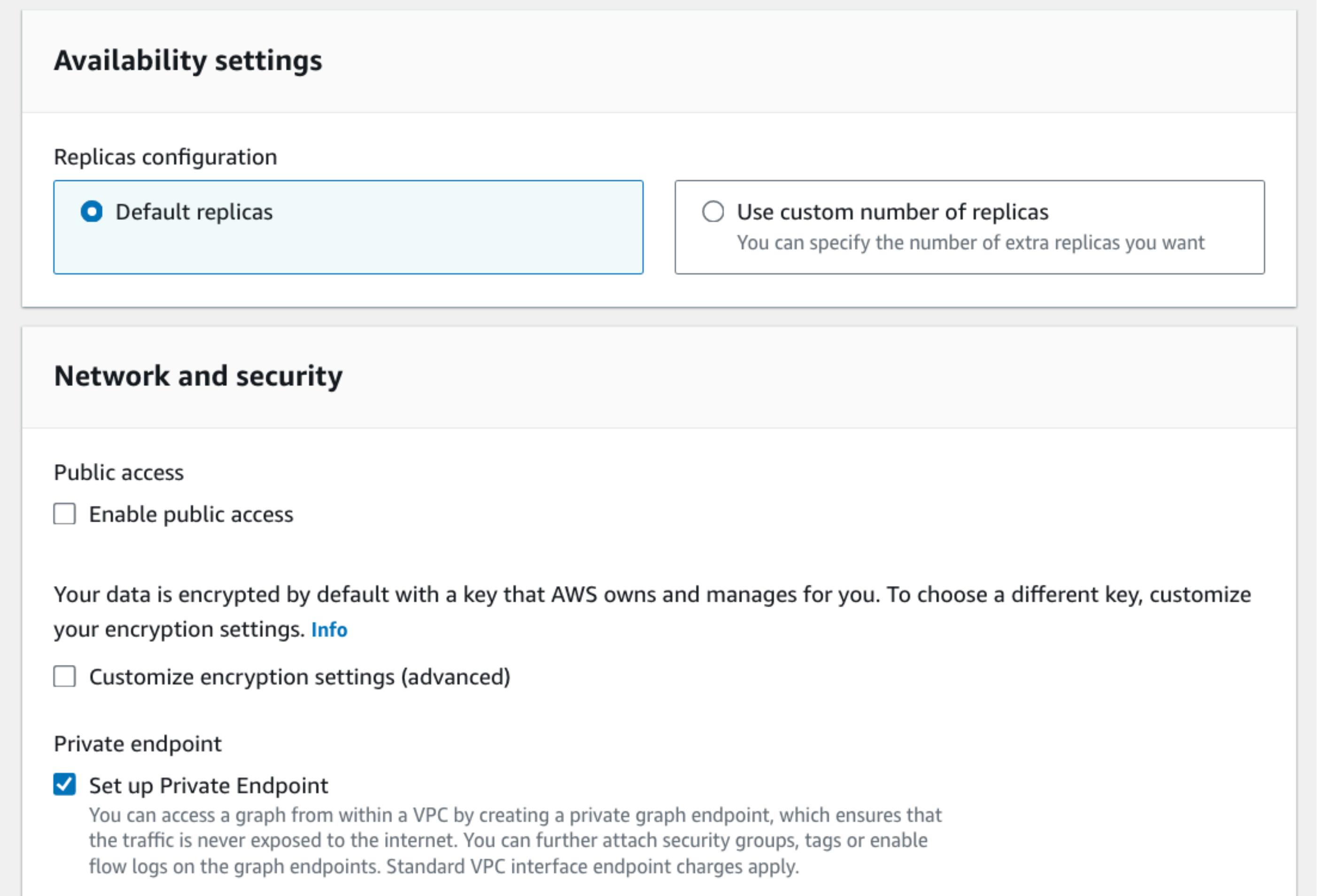
S3 Bucket location

Resource URI

s3://bucket/prefix/object

Path must be in the form s3://bucket/prefix/. It must end with a slash (/) and not include any files.

3. Choose the Network Settings and Replica counts.



Availability settings

Replicas configuration

● Default replicas

○ Use custom number of replicas
You can specify the number of extra replicas you want

Network and security

Public access

□ Enable public access

Your data is encrypted by default with a key that AWS owns and manages for you. To choose a different key, customize your encryption settings. Info

□ Customize encryption settings (advanced)

Private endpoint

☑ Set up Private Endpoint
You can access a graph from within a VPC by creating a private graph endpoint, which ensures that the traffic is never exposed to the internet. You can further attach security groups, tags or enable flow logs on the graph endpoints. Standard VPC interface endpoint charges apply.

4. Create graph.

Creating a Neptune Analytics graph from Neptune cluster or snapshot

Neptune Analytics provides an easy way to bulk import data from an existing Neptune Database cluster or snapshot into a new Neptune Analytics graph, using the `CreateGraphUsingImportTask` API. Data from your source cluster or snapshot is bulk exported into an Amazon S3 bucket that you configure, analyzed to find the right memory configuration, and bulk imported into a new Neptune Analytics graph. You can check the progress of your bulk import at any time using the `GetImportTask` API as well.

A few things to consider while using this feature:

- You can only import from Neptune Database clusters and snapshots running on a version newer than or equal to 1.3.0.

- The exported data from your source Neptune Database cluster or snapshot will reside in your buckets only, and will be encrypted using a KMS key that you provide. The exported data is not directly consumable in any other way into Neptune outside of the `CreateGraphUsingImportTask` API. The exported data is not used after the lifetime of the request, and can be deleted by the user.
- You need to provide permissions to perform the export task on the Neptune Database cluster or snapshot, write to your Amazon S3 bucket, and use your KMS key while writing data.
- If your source is a Neptune Database cluster, a clone is taken from it and used for export. The original Neptune Database cluster will not be impacted. The cloned cluster is internally managed by the service and is deleted upon completion.
- If your source is a Neptune snapshot, a restored DBCluster is created from it, and used for export. The restored cluster is internally managed by the service and is deleted upon completion.
- This process is not recommended for small sized graphs. The export process is async, and works best for medium/large sized graphs with a size greater than 25GB. For smaller graphs, a better alternative is to use the Neptune export feature to generate CSV data directly from your source, upload that to Amazon S3 and then use the Batch load API instead.

A quick summary of steps to import from a Neptune cluster or a Neptune snapshot:

1. Obtain the ARN of your Neptune cluster or snapshot: This can be done from the AWS console or using the Neptune CLI.
2. Create an IAM role with permissions to export from Neptune to Neptune Analytics: Create an IAM role that has permissions to perform an export of your Neptune graph, write to Amazon S3 and use your KMS key for writing data in Amazon S3.
3. Use the `CreateGraphUsingImportTask` API with source = NEPTUNE, and provide the ARN of your source, Amazon S3 path to export the data, KMS key to use for exporting data and additional arguments for your Neptune Analytics graph. This should return a `task-id`.
4. Use `GetImportTask` API to get the details of your task.

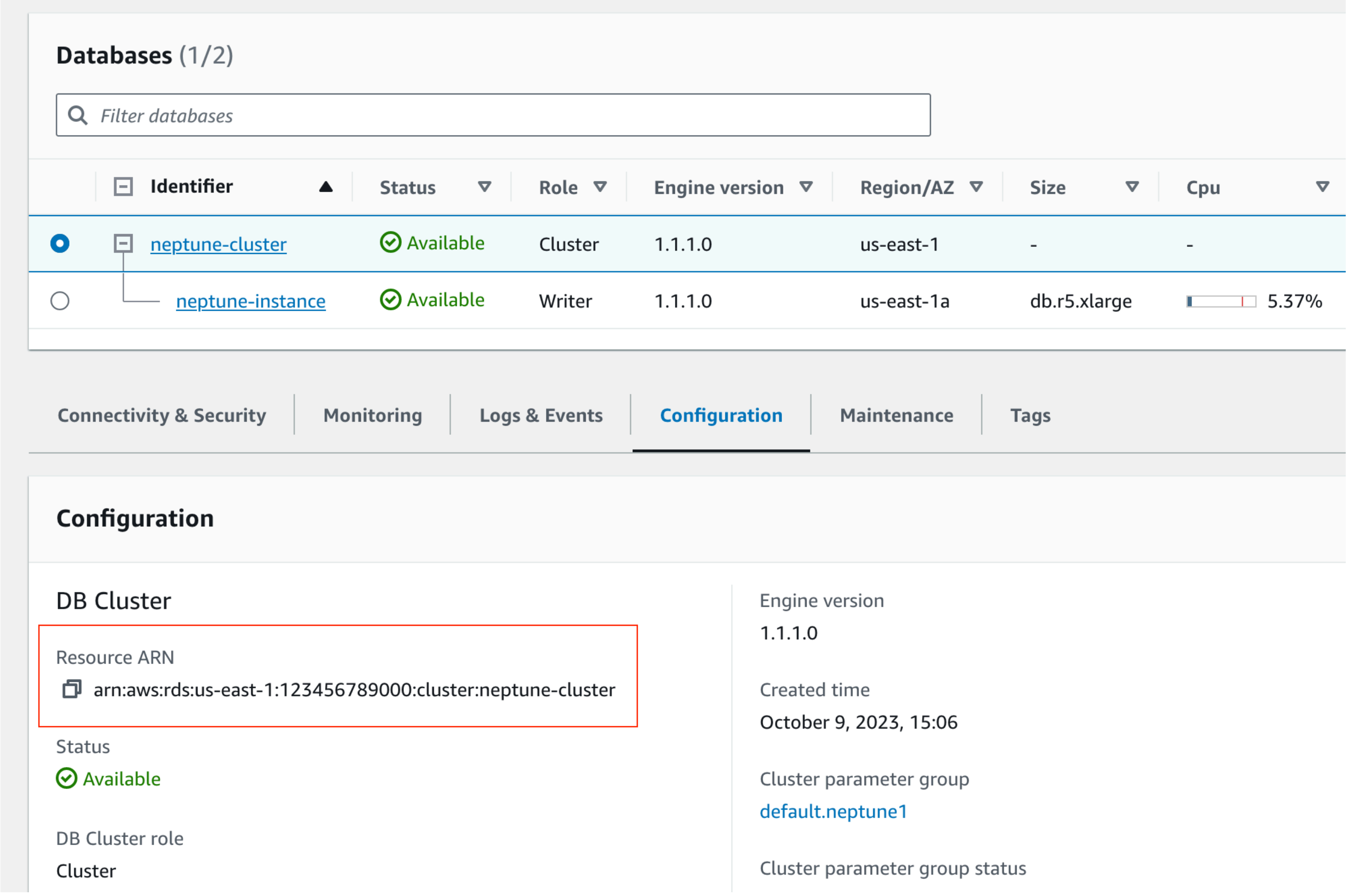
Obtain the ARN of your Neptune cluster or snapshot

Via the CLI:

```
# Obtaining the ARN of an existing DB Cluster
  aws neptune describe-db-clusters   \
      --db-cluster-identifier *<name> \
      --query 'DBClusters[0].DBClusterArn'


  # Obtaining the ARN of an existing DB Cluster Snapshot
  aws neptune describe-db-cluster-snapshots \
      --db-cluster-snapshot-identifier <snapshot name> \
      --query 'DBClusterSnapshots[0].DBClusterSnapshotArn'
```

Via the AWS console. The ARN can be found on the cluster details page.



Databases (1/2)

	Identifier	Status	Role	Engine version	Region/AZ	Size	Cpu
●	neptune-cluster	Available	Cluster	1.1.1.0	us-east-1	-	-
○	neptune-instance	Available	Writer	1.1.1.0	us-east-1a	db.r5.xlarge	5.37%

Configuration

DB Cluster

Resource ARN
arn:aws:rds:us-east-1:123456789000:cluster:neptune-cluster

Status
Available

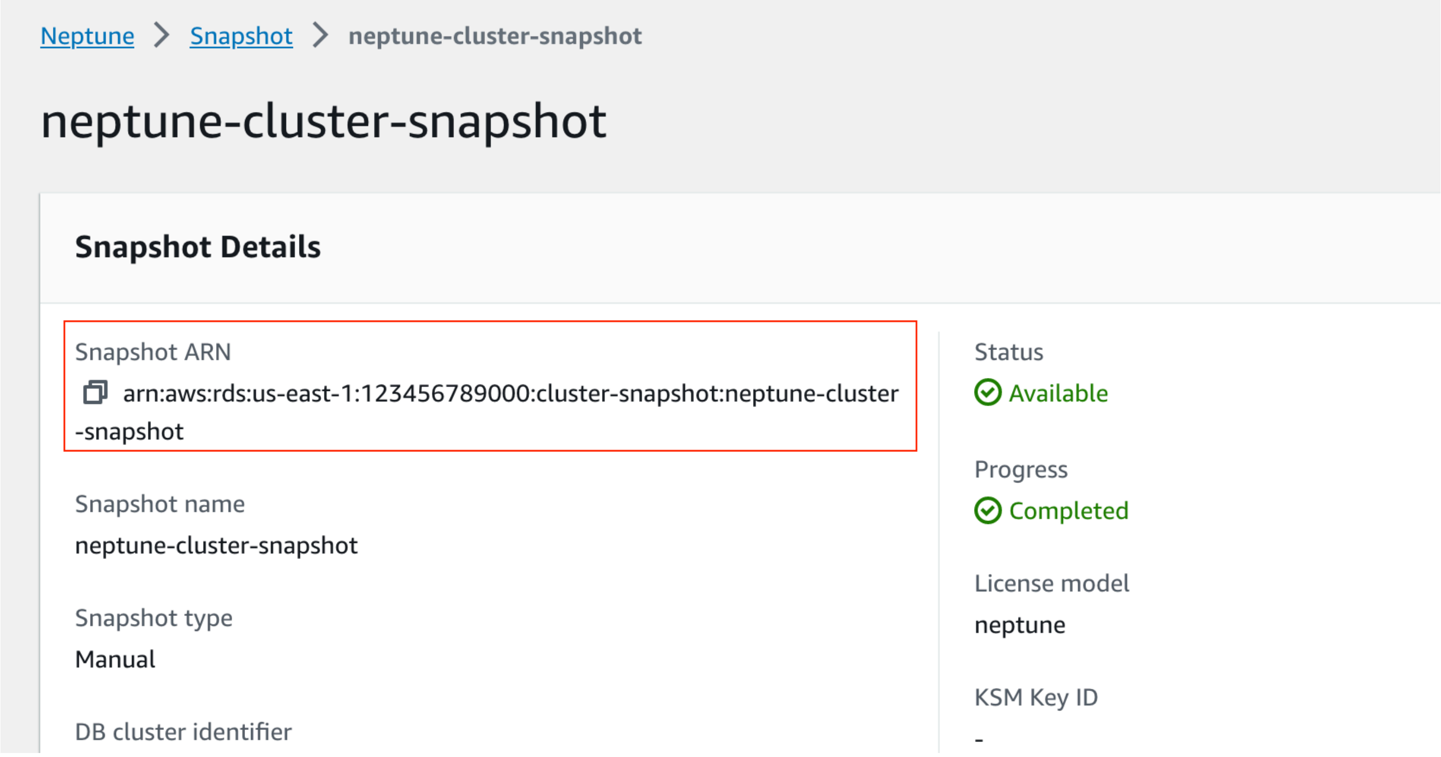
DB Cluster role
Cluster

Engine version
1.1.1.0

Created time
October 9, 2023, 15:06

Cluster parameter group
default.neptune1

Cluster parameter group status



Neptune > Snapshot > neptune-cluster-snapshot

neptune-cluster-snapshot

Snapshot Details

Snapshot ARN
arn:aws:rds:us-east-1:123456789000:cluster-snapshot:neptune-cluster-snapshot

Snapshot name
neptune-cluster-snapshot

Snapshot type
Manual

DB cluster identifier

Status
✓ Available

Progress
✓ Completed

License model
neptune

KSM Key ID
-

Create an IAM role with permissions to export from Neptune to Neptune Analytics

1. Open the IAM console at https://console.aws.amazon.com/iam/. Choose **Roles**, and then choose **Create Role**.
2. Provie a role name.
3. Choose **Amazon S3** as the AWS service.
4. In the **permissions** section, choose:
   - `AmazonS3FullAccess`
   - `NeptuneFullAccess`
   - `AmazonRDSFullAccess`
5. Also create a custom policy with at least the following permissions for the AWS KMS key used:
   - `kms:ListGrants`
   - `kms:CreateGrant`
   - `kms:RevokeGrant`
   - `kms:DescribeKey`
   - `kms:GenerateDataKey`
   - `kms:Encrypt`

- `kms:ReEncrypt`
- `kms:Decrypt`

> **Note**
>
> Make sure there are no resource-level `Deny` policies attached to your AWS KMS key. If there are, explicitly allow the AWS KMS permissions for the `Export` role.

6. On the **Trust Relationships** tab, choose **Edit trust relationship**, and paste the following trust policy. Choose **Save** to save the trust relationship.

```
{
      "Version": "2012-10-17",
      "Statement": [
          {
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "export.rds.amazonaws.com",
                      "neptune-graph.amazonaws.com"
                  ]
              },
              "Action": "sts:AssumeRole"
          }
      ]
  }
```

Your IAM role is now ready for import.

Via CLI/SDK

For importing data via Neptune , the API expects additional import-options as defined here NeptuneImportOptions.

Example 1: Create a graph from a Neptune cluster.

```
aws neptune-graph create-graph-using-import-task \
   --graph-name <graph-name>
   --source arn:aws:rds:<region>:451235071234:cluster:nodelete-p8-test-snb-neptune-
gremlin-data-100 \
   --min-provisioned-memory 1024 \
   --max-provisioned-memory 1024 \
   --role-arn <role-arn> \
   --import-options '{"neptune": {
      "s3ExportKmsKeyId":"arn:aws:kms:<region>:<account>:key/<key>",
      "s3ExportPath": :"<s3 path for exported data>"
   }}'
```

Example 2: Create a graph from a Neptune cluster with the default vertex preserved.

```
aws neptune-graph create-graph-using-import-task \
   --graph-name <graph-name>
   --source arn:aws:rds:<region>:451235071234:cluster:nodelete-p8-test-snb-neptune-
gremlin-data-100 \
   --min-provisioned-memory 1024 \
   --max-provisioned-memory 1024 \
   --role-arn <role-arn> \
   --import-options '{"neptune": {
      "s3ExportKmsKeyId":"arn:aws:kms:<region>:<account>:key/<key>",
      "s3ExportPath": :"<s3 path for exported data>",
      "preserveDefaultVertexLabels" : true
   }}'
```

Example 3: Create a graph from Neptune cluster with the default edge Id preserved

```
aws neptune-graph create-graph-using-import-task \
   --graph-name <graph-name>
   --source arn:aws:rds:<region>:451235071234:cluster:nodelete-p8-test-snb-neptune-
gremlin-data-100 \
   --min-provisioned-memory 1024 \
   --max-provisioned-memory 1024 \
   --role-arn <role-arn> \
   --import-options '{"neptune": {
      "s3ExportKmsKeyId":"arn:aws:kms:<region>:<account>:key/<key>",
      "s3ExportPath": :"<s3 path for exported data>",
      "preserveEdgeIds" : true
   }}'
```

Bulk import data into an existing Neptune Analytics graph

Neptune Analytics now allows you to efficiently import large datasets into an already provisioned graph database using the `StartImportTask` API. This API facilitates the direct loading of data from an Amazon S3 bucket into an **empty** Neptune Analytics graph. This is designed for loading data into existing empty clusters.

Two common use cases for using this feature:

1. Bulk importing data multiple times without provisioning a new graph for each dataset. This helps during the development phase of a project where datasets are being converted into Neptune Analytics compatible load formats.
2. Use cases where graph provisioning privileges need to be separated from data operation privileges. For example, scenarios where graph provisioning needs to be done by only by the infrastructure team, and data loading and querying is done by the data engineering team.

For use cases where you want to create a new graph loaded with data, use the `CreateGraphUsingImportTask` API instead.

For incrementally loading data from Amazon S3 you can use the loader integration with the openCypher `CALL` clause. For more information see Batch load.

Prerequisites

- An empty Amazon Neptune Analytics graph.
- Data stored in an Amazon Amazon S3 bucket in the same region as the graph.
- An IAM role with permissions to access the Amazon S3 bucket. For more information, see Create your IAM role for Amazon S3 access.

Important considerations

- **Data integrity**: The `StartImportTask` API is designed to work with graphs that are empty. If your graph contains data, you can first reset the graph using the reset-graph API. If the Import task finds that the graph is not empty the operation will fail. This operation will delete all data from the graph, so ensure you have backups if necessary. You can use the create-graph-snapshot API to create snapshot of your existing graph.
- **Atomic Operation**: The data import is atomic, meaning it either completes fully or does not apply at all. If the import fails we would reset the state back to an empty graph.
- **Format Support**: Loading data supports the same data format as supported by `create-graph-using-import-task` and `neptune.load()` This API doesn't support importing data from Neptune .

- **Queries**: Queries will stop working while the import is in progress. You will get a `Cannot execute any query until bulk import is complete` error until the import finishes.

Steps for bulk importing data

1. Resetting the graph (if necessary):

   If your graph is not empty, reset it using the following command:

   ```
   aws neptune-graph reset-graph --graph-identifier <graph-id>
   ```

   > **Note**
   >
   > This command will completely remove all existing data from your graph. It is recommended that you take a graph snapshot before performing this action.

2. Start the import task:

   To load data into your Neptune graph, use the `start-import-task` command as follows:

   ```
   aws neptune-graph start-import-task \
   --graph-identifier <graph-id> \
   --source <s3-path-to-data> \
   --format <data-format> \
   --role-arn <IAM-role-ARN> \
   [--fail-on-error | --no-fail-on-error]
   ```

   - `graph-identifier`: The unique identifier of your Neptune graph.
   - `source`: An Amazon S3 URI prefix. All object names with matching prefixes are loaded. See Neptune loader request parameters for Amazon S3 URI prefix examples.
   - `format`: The data format of the Amazon S3 data to be loaded, either `csv`, `openCypher`, or `ntriples`. For more information, see Data formats.
   - `role-arn`: The ARN of the IAM role that Neptune Analytics can assume to access your Amazon S3 data.
   - `(--no-)fail-on-error`: (Optional) Stops the import process early if an error occurs. By default, the system attempts to stop at the first error.

Troubleshooting bulk import

Common errors

1. The Amazon S3 bucket and your graph are in different regions.

   Verify that your graph and the Amazon S3 bucket are in the same region. Neptune Analytics only supports loading data in the same region.

   ```
   export GRAPH_ID="g-f6yababab6"               // Replace with your graph identifier
   export S3_BUCKET_NAME="haetoqq-amb-76"       // Replace with your S3 bucket which
    contains your graph data files.

   # Make sure your graph and S3 bucket are in the same region
   aws neptune-graph get-graph --graph-identifier $GRAPH_ID
   aws s3api get-bucket-location --bucket $S3_BUCKET_NAME
   ```

2. The IAM role used does not have the correct permissions.

   Verify that you have created the IAM role correctly with read permission to Amazon S3 - see Create your IAM role for Amazon S3 access.

   ```
   export GRAPH_EXEC_ROLE="GraphExecutionRole"
   aws iam list-attached-role-policies --role-name $GRAPH_EXEC_ROLE
   # Output should contain "PolicyName": "AmazonS3*Access".
   ```

3. The `AssumeRole` permission is not granted to Neptune Analytics through the AssumeRolePolicy.

   Verify that you have attached the policy that allows Neptune Analytics to assume the IAM role to access the Amazon S3 bucket. See Create your IAM role for Amazon S3 access.

   ```
   export GRAPH_EXEC_ROLE="GraphExecutionRole"   // Replace with your IAM role.


   #Check to make sure Neptune Analytics can assume this role to read from the
    specified S3 bucket.
   aws iam get-role --role-name $GRAPH_EXEC_ROLE --query 'Role.AssumeRolePolicyDocument'
    --output text
   # Output should contain - SERVICE neptune-graph.amazonaws.com
   ```

4. The bulk load files are in a public Amazon S3 bucket, but the files themselves are not made public for reading.

   When adding bulk load files to a public Amazon S3 bucket, ensure that each file's access control list (ACL) is set to allow public reads. For example, to set this through the AWS CLI:

   ```
   aws s3 cp <FileSourceLocation> <FileTargetLocation> --acl public-read
   ```

   This setting can also be done through the Amazon S3 console or the AWS SDKs. For more details, refer to the documentation for Configuring ACLs.

Checking the details and progress of an import task

You can use the GetImportTask API to track the progress and the status of your import task.

```
aws neptune-graph get-import-task --task-id <task-id>
```

An Import task can be in the following state:

- **INITIALIZING**: The task is preparing for import, including provisioning a graph when using the `CreateGraphUsingImportTask` API.
- **ANALYZING_DATA**: The task is taking an initial pass through the dataset to determine the optimal configuration for the graph.
- **IMPORTING**: The data is being loaded into the graph.
- **EXPORTING**: Data is being exported from the Neptune cluster or snapshot. This is only applicable when performing an import task with a source of Neptune and through the `CreateGraphUsingImportTask` API.
- **ROLLING_BACK**: The import task encountered an error. Refer to the troubleshooting section to investigate the errors. The import task will be rolled back and eventually marked as `FAILED`.
- **SUCCEEDED**: Graph creation and data loading have succeeded. Use the `get-graph` API to view details of the final graph.
- **REPROVISIONING**: A temporary state while the graph is being reconfigured during the import task.
- **FAILED**: Graph creation or data loading has failed. Refer to the troubleshooting section to understand the reason for the failure.

- **CANCELLING**: The user has cancelled the import task, and cancellation is in progress.
- **CANCELLED**: The import task has been cancelled, and all resources have been released.

Additionally, import task can be used to track the progress of the load, error count and graph summary.

Canceling an import task

You can cancel a running import task by using the CancelImportTask API.

```
aws neptune-graph cancel-import-task \
--task-id <task-id>
```

The import task will will be canceled and all changes rolled back. The state of the import task will switch to `CANCELING` after `cancel-import-task` API is called and eventually the state will be `CANCELED` when rollback finishes. You can check the current state of your import task using the GetImportTask API.

```
aws neptune-graph get-import-task \
--task-id <task-id>
```

Troubleshooting

For both bulk load and batch load, all the errors and summary of the load is sent to the CloudWatch log group in your account. To view the logs go to CloudWatch, click log groups from the left column, then search for and click `/aws/neptune/import-task-logs/`.

1. **Batch Load**: The logs for each load is saved under `/aws/neptune/import-task-logs/<graph-id>/<load-id>` CloudWatch log stream.
2. **Bulk Load using Import Task**: The logs are saved under `/aws/neptune/import-task-logs/<graph-id>/<task-id>` CloudWatch log stream.

- **S3_ACCESS_DENIED**: The server does not have permissions to list or download the given file. Fix the permissions and retry. See Create your IAM role for Amazon S3 access for help setting up the Amazon S3 permissions.
- **LARGE_STRING_ERROR**: One or more strings exceeded the limit on the size of strings. This data cannot be inserted as is. Update the strings exceeding the limit and retry.

- **PARSING_ERROR**: Error parsing the given value(s). Correct the value(s) and retry. More information on different parsing errors is provided in this section.
- **OUT_OF_MEMORY**: No more data can be loaded in the current m-NCU. If encountered during import task, set a higher m-NCU and retry. If encountered during batch load, scale the number of m-NCU and retry the batch load.
- **PARTITION_FULL_ERROR**: No more data can be loaded in the internal server configuration. If encountered during import task, the import workflow would change the server configuration and retry. If encountered during batch load, reach out to the AWS service team to unblock loading of new data.

Common parsing errors and solutions

Error template	Solution
Invalid data type encountered for header val:badtype when parsing line `[:ID,firstName:String,val:badtype,:LABEL]` .	Incorrect Datatype provided. Check the documentation for supported data types. See Data formats for more information.
Multi-valued columns are not supported `firstName:String[]` when parsing line `[:ID,firstName:String[],val:String,:LABEL]` .	The `opencypher` format does not support multivalued user defined properties. Try using the `csv` format to insert multivalued vertex properties, or remove multivalued properties.
Bad header for a file in '`OPEN_CYPHER`' format, could not determine node or relationship file, found system columns from '`csv`' format when parsing line `[~id,firstName:String,val:int,:LABEL]` .	Both the `opencypher` and `csv` format expect certain header columns to be present. Make sure you have entered them correctly. Check the Data formats documentation for required fields by format.
Bad header for a file in '`OPEN_CYPHER`' format, could not determine node or relationship file.	The header of the files does not have the required system columns. Check the Data formats for required fields by format.

Error template	Solution
Relationship file in 'OPEN_CYPHER ' format should contain both :START_ID and :END_ID columns when parsing line [:START_ID,firstName:String] .	The header of the edge files does not have all the required system columns. Check the Data formats for required fields by format.
Invalid data type. Found system columns from 'OPEN_CYPHER ' format :ID when parsing line [:ID,firstName:String,val:Int,~label] .	The opencypher and csv formats have different system column names, and they begin with : and ~ respectively. User defined properties cannot begin with those reserved prefixes in the respective formats. Confirm the format name and system column names, or update user defined properties to not use reserved prefixes.
Named column name is not present for header field :BLAH when parsing line [:ID,:BLAH,firstName:String] .	The opencypher and csv formats have different system column names, and they begin with : and ~ respectively. User defined properties cannot begin with those reserved prefixes in the respective formats. Confirm the format name and system column names, or update user defined properties to not use reserved prefixes.
System column other than ID cannot be stored as a property: <columnHeader>.	The opencypher and csv formats have different system column names, and they begin with : and ~ respectively. User defined properties cannot begin with those reserved prefixes in the respective formats. Confirm the format name and system column names, or update user defined properties to not use reserved prefixes.
Duplicate user column firstName when parsing line [:ID,:LABEL, firstName:String, firstName:String] .	The file contains duplicate user defined property column names in the header. Remove all of the duplicate columns.

Error template	Solution
Duplicate system column `:ID` found when parsing line `[:ID,:ID,firstName :String,:LABEL]` .	The file contains duplicate system column names in the header. Remove all of the duplicate columns.
Invalid column name provided for loading embeddings: `[abcd]` for filename: someFilename. Embedding column name must be the same as their corresponding vector index name when parsing line `[:ID,firstName:String,abcd:Vector,:LABEL] in [filename]` .	An incorrect name is used for the vector embeddings.
"date" type is curretly not supported. "datetime" may be an alternative type.	Use `datetime` as the field type as date type suppoorted yet in Neptune Analytics.
Headers must be non-empty.	Headers need to be non empty. If the file has an empty line in the beginning, remove the empty line.
Failure encounted while parsing the `csv` file.	Likely reason is the number of columns in the row doesn't match the number of columns provided in the header. If you dont have a value for a column, provide an empty value. For example: `123,vertex,,,` .
Could not process value of type:`http://www.w3.org/2001/XMLSchema#int` for value: a when parsing line `[v1,v19683,con,a]` in [file].	There is a mismatch between the type of the value provided for that column in the row and the type specified in the header. In this specific case the column header is annotated with integer type but a is not parseable as an integer.
Could not load vector embedding: `[a,bc]`. Check the dimensionality for this vector.	The size of the vector does not match the dimension defined in the vector search configuration for the graph.

Error template	Solution
Could not load vector embedding: `[a,NaN]`. Check the value for this vector.	Float and double values in scientific notation are currently not supported. Also `Infinity`, `-Infinity` , `INF`, `-INF`, and `NaN` are not recognized.

Graph snapshots

Neptune Analytics provides you the ability to create a named snapshot of your analytics graph, and also the ability to restore from existing graph snapshots. A graph snapshot is a compacted deep copy of your entire graph. Snapshots are created asynchronously, and do not affect the performance of your running graph. You can restore a snapshot into a new graph at any time.

Topics

Creating a graph snapshot

CLI/SDK

Find the id of your graph

```
aws neptune-graph list-graphs
```

This command will give you a list of your graphs. Find the graph id of the graph you want to take a snapshot of and write it down.

Create a graph snapshot

```
aws neptune-graph create-graph-snapshot \
--graph-identifier <GRAPH_ID> \
--snapshot-name <SNAPSHOT_NAME>
```

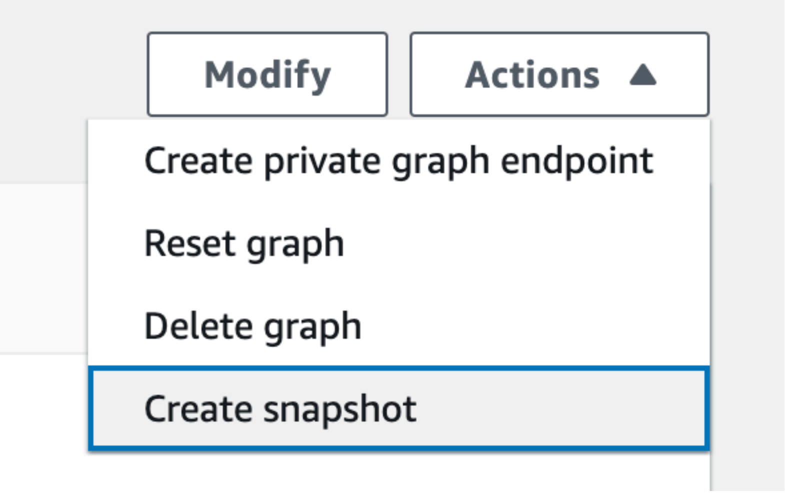
Parameters:

1. `graph-identifier` - The graph id you want to take the snapshot from.
2. `snapshot-name` - The name you want to use for your graph snapshot.

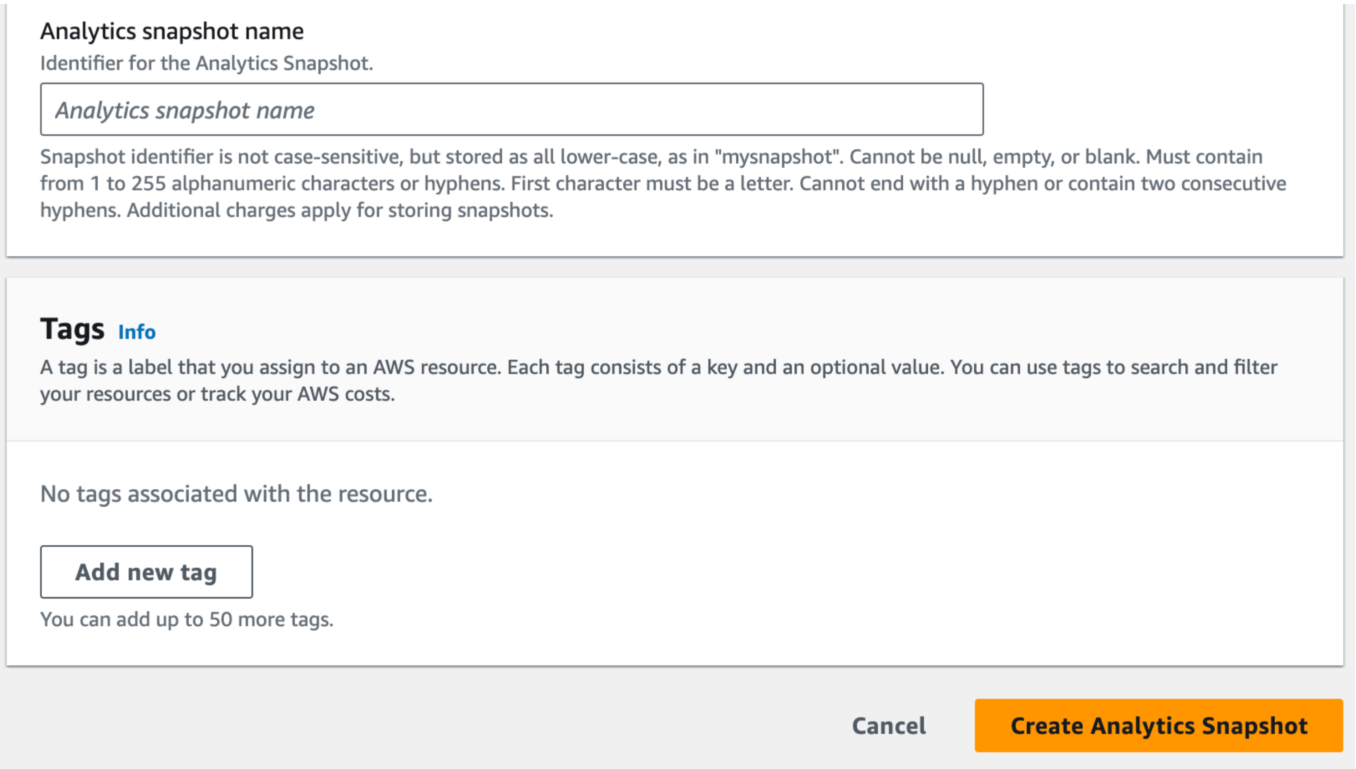
Neptune Console

1. Select the graph you want to take a snapshot of in **Analytics**, **Graphs**.

2. Choose **Actions**, and choose **Create snapshot**.



3. Give the snapshot a name and choose **Create Analytics Snapshot**.



Analytics snapshot name

Identifier for the Analytics Snapshot.

Analytics snapshot name

Snapshot identifier is not case-sensitive, but stored as all lower-case, as in "mysnapshot". Cannot be null, empty, or blank. Must contain from 1 to 255 alphanumeric characters or hyphens. First character must be a letter. Cannot end with a hyphen or contain two consecutive hyphens. Additional charges apply for storing snapshots.

Tags Info

A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

No tags associated with the resource.

You can add up to 50 more tags.

Listing existing graph snapshots

CLI/SDK

List all graph snapshots

```
aws neptune-graph list-graph-snapshots
```

Look up a single graph snapshot

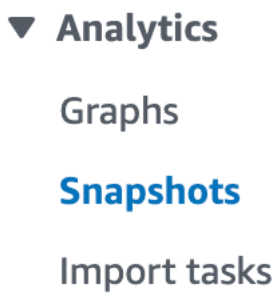
```
aws neptune-graph get-graph-snapshot \
--snapshot-id <SNAPSHOT_ID>
```

Status:

- CREATING: The snapshot is currently being created.
- AVAILABLE: The snapshot is available and can be restored from.
- FAILED: The snapshot failed to create.
- DELETING: The snapshot is currently being deleted.

Neptune Console

You can view your snapshots by expanding **Analytics** and choosing **Snapshots**.



Restoring from a graph snapshot

When you create a snapshot of a graph, Neptune Analytics creates a storage volume snapshot of the graph, backing up all of its data. You can later create a new Neptune Analytics graph by restoring from this snapshot. When you restore the graph, you provide the name of the graph snapshot to restore from, and then provide a name for the new graph that is created by the restore.

CLI/SDK

Restore an analytics graph from a snapshot

```
aws neptune-graph restore-graph-from-snapshot \
--graph-name <NEW_GRAPH_NAME> \
--snapshot-id <SNAPSHOT_ID>
```

Parameters:

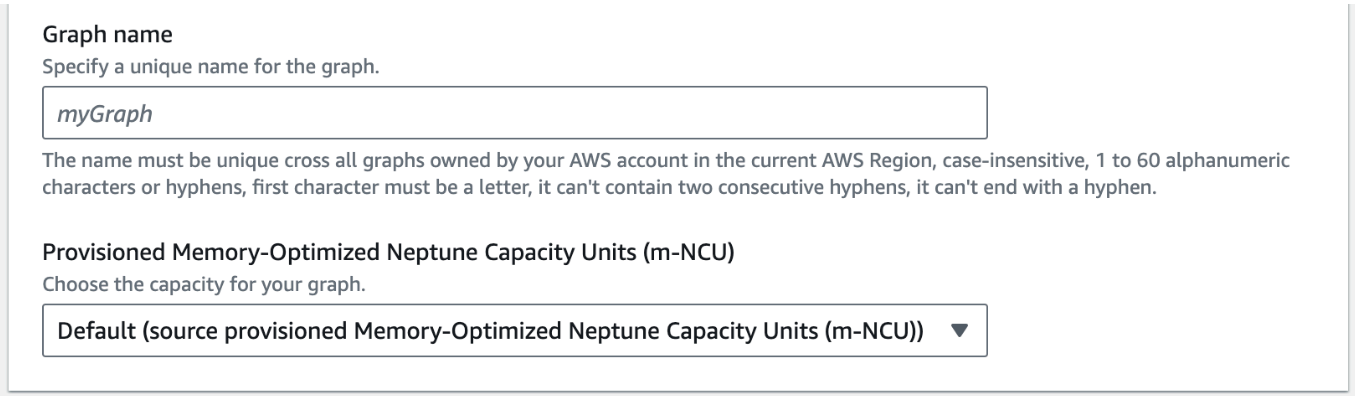
1. `graph-name` - The name of the new Neptune Analytics graph that will be created from the snapshot.
2. `snapshot-id` - The snapshot identifier you want to restore from.

Optional parameters:

1. `min-provisioned-memory` - The minimum provisioned memory to use for the new graph. Default: 64.
2. `max-provisioned-memory` - The maximum provisioned memory to use for the new graph. Default: 1024, or the approved upper limit for your account. Neptune Analytics will analyze the data to find the best memory configuration between min-provisioned-memory and max-provisioned-memory to create the graph.
3. `public-access, no-public-access` - Whether connectivity over public networks (internet) is enabled or not. Default: `no-public-access`.
4. `replica-count` - The number of replicas to provision on the new graph after import. Default: 0, Min: 0, Max:2.

Neptune Console

1. Find the snapshot you want to restore by expanding **Analytics** and choosing **Snapshots**.
2. Select the snapshot and choose **Restore snapshot**.
3. Give the graph a unique name, and choose provisioned m-NCU.



Graph name

Specify a unique name for the graph.

myGraph

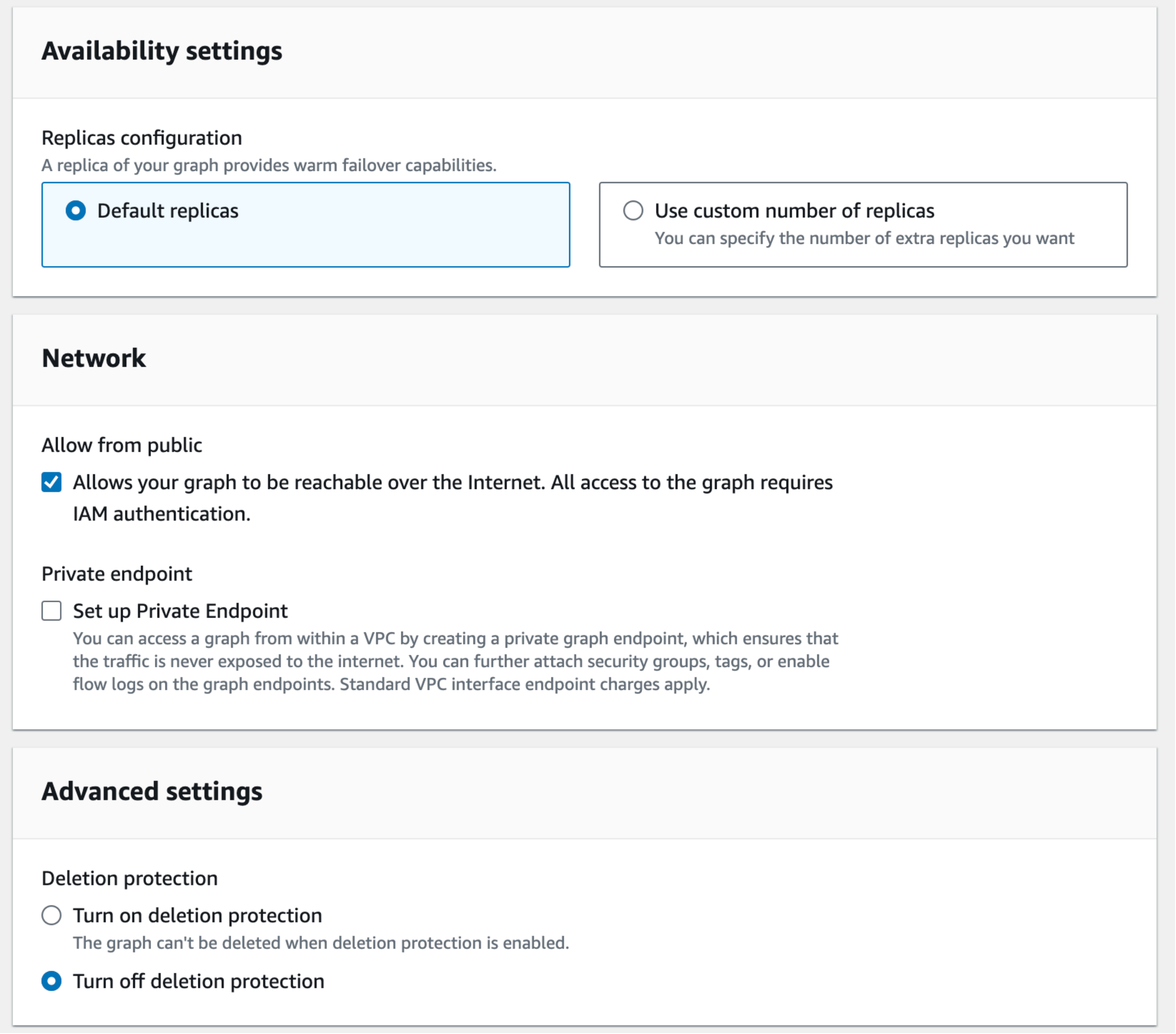
The name must be unique cross all graphs owned by your AWS account in the current AWS Region, case-insensitive, 1 to 60 alphanumeric characters or hyphens, first character must be a letter, it can't contain two consecutive hyphens, it can't end with a hyphen.

Provisioned Memory-Optimized Neptune Capacity Units (m-NCU)

Choose the capacity for your graph.

Default (source provisioned Memory-Optimized Neptune Capacity Units (m-NCU)) ▼

4. Update availibility, network, and advanced settings if necessary, and choose the **Restore snapshot** button.



Availability settings

Replicas configuration

A replica of your graph provides warm failover capabilities.

● Default replicas

○ Use custom number of replicas
You can specify the number of extra replicas you want

Network

Allow from public

☑ Allows your graph to be reachable over the Internet. All access to the graph requires IAM authentication.

Private endpoint

□ Set up Private Endpoint
You can access a graph from within a VPC by creating a private graph endpoint, which ensures that the traffic is never exposed to the internet. You can further attach security groups, tags, or enable flow logs on the graph endpoints. Standard VPC interface endpoint charges apply.

Advanced settings

Deletion protection

○ Turn on deletion protection
The graph can't be deleted when deletion protection is enabled.

● Turn off deletion protection

5. You can review the status of your restored graph by expanding **Analytics** and choosing **Graphs**.

Deleting a graph snapshot

CLI/SDK

Delete a snapshot

```
aws neptune-graph delete-graph-snapshot \
--snapshot-id <SNAPSHOT_ID>
```

Neptune Console

1. Expand **Analytics** and choose **Snapshots**.
2. Select the snapshot you want to delete, and choose the **Delete** button.
3. Type "confirm" in the text box to confirm you want to delete the snapshot, then choose the **Delete** button.

Managing your Neptune Analytics graphs

Neptune Analytics graphs involve multiple instances that are connected in a replication topology. Managing graphs often involves deploying changes to multiple servers and making sure that all Neptune Analytics graph replicas are keeping up with the primary graph. Neptune Analytics automatically performs continuous backups, and does not require extensive planning or downtime for performing backups.

Topics

Modifying a Neptune Analytics graph

You can change the settings of a Neptune Analytics graph to accomplish tasks such as changing public connectivity or its provisioned-memory.

It is recommended that you test any changes using a test graph before modifying any production graphs, so that you are able to fully understand the impact of each change.

Memory scaling

Neptune Analytics is a memory-optimized graph database engine for analytics. It stores large graph datasets in memory. You can scale the database to a larger memory size by updating the graph to higher m-NCU.

Maintaining a Neptune Analytics graph

Periodically, Neptune Analytics performs maintenance on Neptune resources. Maintenance most often involves updates to the following resources in your graph:

- Underlying hardware

- Underlying operating system (OS)
- Graph engine version

Neptune Analytics doesn’t have a maintenance window for the graphs. It automatically performs maintenance operations which require the Neptune service to take your graph offline for a short time. Maintenance items that require a resource to be offline include required operating system or engine patching. Required patching is automatically scheduled only for patches that are related to security and instance reliability. Such patching occurs infrequently, typically once every few months. There are no actions required from the users for this to take place.

Deleting a Neptune Analytics graph

You can delete a Neptune Analytics graph when you no longer need it. Before deleting the graph, you can save a snapshot of your data. You can then restore that snapshot at a later date to create a new graph containing the same data. For more information about creating snapshots, see Graph snapshots.

Neptune Analytics doesn't provide a single-step method to delete a graph and its snapshot. Also, the graph cannot be deleted if delete-protection is enabled. This design choice is intended to prevent you from accidentally losing data or taking your application offline. Neptune Analytics graph applications are typically mission critical and require high availability.

Deleting a Neptune Analytics graph

AWS Console

Choose the graph you want to delete, then choose **Delete graph** from the drop-down **Actions** menu. You can choose the following options to preserve the data from the graph in case it is needed later.

- Create a final snapshot of the graph. The default setting is to create a final snapshot.

If you graph has private graph endpoints configured then you need to delete all of the private graph endpoints first. To delete the private graph endpoints:

In the navigation pane, choose **graphs**, and then choose the graph that you want to delete. On the graph page, go to the graph private endpoints section and choose the private graph endpoint you want to delete. Select the delete button and enter "confirm" in the text box.

CLI/API

You can call the delete-graph CLI command, or the DeleteGraph API operation. You can choose the following options to preserve the data from the graph in case it is needed later.

- Create a final snapshot of the graph
- Retain automated backups

```
aws neptune-graph delete-graph --graph-id g-sample
```

If your graph has private graph endpoints configured, you will need to delete the private graph endpoints first.

```
aws neptune-graph delete-private-graph-endpoint --graph-identifier g-sample --vpc-id
 your-vpc-id
```

Tagging Neptune Analytics graph resources

You can use Neptune Analytics graph tags to add metadata to your graph. You can use the tags to add your own notations about graph and graph snapshots. Doing so can help you to document your Neptune Analytics graph resources.

You can also use these tags with IAM policies. You can use them to manage access to Neptune Analytics graph resources and to control what actions can be applied to the Neptune Analytics graph resources. You can also use these tags to track costs by grouping expenses for similarly tagged resources. You can tag the following Neptune Analytics graph resources:

- Neptune Analytics graph
- Neptune Analytics graph snapshots

A Neptune Analytics graph tag is a name-value pair that you define and associate with a Neptune Analytics graph 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 Analytics graph resource. You can use a tag key to define a category, and the tag value might be an item in that category. For example, you might define a tag key of "env" and a tag value of "preprod". In this case, these indicate that the Neptune Analytics graph resource is assigned to the preprod environment. It is recommended that you use a consistent set of tag keys to make it easier to track metadata associated with Neptune Analytics graph resources.

Additionally, you can use conditions in your IAM policies to control access to AWS resources based on the tags used on that resource. You can do this by using the global `aws:ResourceTag/tag-key` condition key. For more information, see Controlling access to AWS resources in the AWS Identity and Access Management user guide.

You can use the AWS management console, the AWS CLI, or the Neptune graph API to add, list, and delete tags on Neptune Analytics graph resources. When using the CLI or API, make sure to provide the Amazon Resource Name (ARN) for the Neptune Analytics graph to work with. For more information about constructing an ARN, see Working with ARNs in Neptune Analytics graph.

> **Note**
>
> Tags are cached for authorization purposes. Because of this, additions and updates to tags on Neptune Analytics resources can take several minutes before they are available.

Using tags to produce detailed billing reports

You can also use tags to track costs by grouping expenses for similarly tagged 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.

Adding, listing, and removing tags

The process to tag a Neptune Analytics graph resource is similar for all resources. The following procedure shows how to tag a Neptune Analytics graph

AWS Console

To add a tag to a Neptune Analytics graph

1. Sign in to the AWS Management Console, and open the Amazon Neptune console at https://console.aws.amazon.com/neptune/.

2. In the navigation pane, choose **Graphs**.
3. Choose the name of the graph that you want to tag. This will show the graph details.
4. In the details section, scroll down to the **Tags** section.
5. Choose **Manage tags**. The Manage tags window appears.
6. Enter a value for Tag key and value.
7. To add additional tags, choose the **Add another tag** button and enter values for its Tag key and value. Repeat this step as many times as necessary.
8. Choose the **Save** button to save the changes.

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/.
2. In the navigation pane, choose **Graphs**.
3. Choose the name of the graph that you want to tag. This will show the graph details.
4. In the details section, scroll down to the **Tags** section.
5. Choose **Manage tags**. The Manage tags window appears.
6. Choose the **Remove** button for the tag you want to delete.
7. Choose the **Save** button to save your changes.

AWS CLI

You can add, list or remove tags for a graph using the AWS CLI.

- To add one or more tags to a Neptune Analytics graph resource, use the AWS CLI command `tag-resource`.

  To list the tags on a Neptune Analytics graph resource, use the AWS CLI command `list-tags-for-resource`.

  To remove one or more tags from a Neptune Analytics graph resource, use the AWS CLI command `untag-resource`.

  To learn more about how to construct the required ARN, see Working with ARNs in Neptune Analytics graph.

Working with ARNs in Neptune Analytics graph

You can get the ARN of a Neptune Analytics graph resource by using the AWS Management Console or the AWS CLI.

Console:

1. For Graphs: Go to your graph page in the Neptune console and look for the “Resource ARN” field.
2. For graph snapshots: Go to your graph page in the Neptune console and look for the "Snapshot ARN" field.

AWS CLI

To get an ARN from the AWS CLI for a particular Neptune Analytics graph resource, you use the `get` command for that resource. The following table shows each AWS CLI command, and the ARN property used with the command to get an ARN.

Resource	AWS CLI command	ARN property
Graph	get-graph	arn
GraphSnapshot	get-graph-snapshot	arn

As an example, running the following command:

```
aws neptune-graph get-graph --graph-id g-vgebxfyat7 --query "arn"
```

would return `"arn:aws:neptune-graph:us-east-1:123456789012:graph/g-vgebxfyat7"`.

Monitoring Neptune Analytics

Logging Neptune Analytics API calls using AWS CloudTrail

Neptune Analytics is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Neptune Analytics. CloudTrail captures all API calls for Neptune Analytics as events. The calls captured include calls from the Neptune Analytics console and code calls to the Neptune Analytics API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Neptune Analytics. If you don't configure a trail, you can still view the most recent management events in the CloudTrail console in the **Event history** section. Using the information collected by CloudTrail, you can determine the request that was made to Neptune Analytics, the IP address from which the request was made, who made the request, when it was made, and additional details.

For robust monitoring and alerting, you can also integrate CloudTrail events with Amazon CloudWatch logs. To enhance your analysis of Neptune Analytics service activity and identify changes in activities for an AWS account, you can query AWS CloudTrail logs using Amazon Athena. For example, you can use queries to identify trends and further isolate activity by attributes such as source IP address or user.

To learn more about CloudTrail, including how to configure and enable it, see the AWS CloudTrail user guide.

Neptune Analytics information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When supported event activity occurs in Neptune Analytics, that activity is recorded in a CloudTrail event along with other AWS service events in the **Event history** section. 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 Analytics, create a trail. A trail enables CloudTrail to deliver log files to an Amazon S3. By default, when you create a trail in the console, the trail applies to all AWS regions. The trail logs events from all regions in the AWS partition and delivers the log files to the Amazon S3 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 the following:

- 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

Control plane events in CloudTrail

The following control plane API actions are logged by default as events in CloudTrail:

- CreateGraph
- ListGraphs
- GetGraph
- UpdateGraph
- ResetGraph
- DeleteGraph
- CreateGraphUsingImportTask
- ListImportTasks
- GetImportTask
- CancelImportTask
- CreatePrivateGraphEndpoint
- ListPrivateGraphEndpoints
- GetPrivateGraphEndpoint
- DeletePrivateGraphEndpoint
- CreateGraphSnapshot
- ListGraphSnapshots
- GetGraphSnapshot
- RestoreGraphFromSnapshot
- DeleteGraphSnapshot
- TagResource
- ListTagsForResource

- UntagResource

Data plane events in CloudTrail

To enable logging of the following API actions in CloudTrail, you'll need to enable logging of data plane API activity in CloudTrail. See Logging data events for more information. By default, CloudTrail doesn't log data events.

> **Note**
>
> Additional charges apply for data events. For more information, see AWS CloudTrail pricing.

Data plane events can be filtered by resource type for granular control over which Neptune Analytics API calls you want to selectively log and pay for in CloudTrail. For example, by specifying `AWS::NeptuneGraph::Graph` as a resource type, you can log only calls to the Neptune Analytics APIs. You can add an additional filter to exclude some events if you don't want them to be logged. For more information, see AdvancedFieldSelectors in the AWS CloudTrail API reference.

Neptune Analytics logs the following data plane API actions as data events:

- GetGraphSummary
- ExecuteQuery
- GetQuery
- ListQueries
- CancelQuery

Understanding Neptune Analytics 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 other information. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

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 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.

The following examples demonstrate CloudTrail logs of these event types:

- CreateGraph

```
{
"eventVersion": "1.08",
"userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
    "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "sessionContext": {
        "sessionIssuer": {
            "type": "Role",
            "principalId": "AKIAI44QH8DHBEXAMPLE",
            "arn": "arn:aws:iam::111122223333:role/admin-role",
            "accountId": "111122223333",
            "userName": "bob"
        },
        "webIdFederationData": {},
        "attributes": {
            "creationDate": "2023-11-22T13:45:16Z",
            "mfaAuthenticated": "false"
        }
    },
    "invokedBy": "AWS Internal"
},
"eventTime": "2023-11-22T13:53:45Z",
"eventSource": "neptune-graph.amazonaws.com",
"eventName": "CreateGraph",
"awsRegion": "us-east-1",
"sourceIPAddress": "192.0.2.0",
"userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0 botocore/1.10.63",
"requestParameters": {
    "graphName": "bobgraph",
    "provisionedMemory": 128,
    "clientToken": "bobtoken",
    "deletionProtection": false
},
"responseElements": {
    "graph": {
        "allowFromPublic": false,
        "arn": "arn:aws:neptune-graph:us-east-1:111122223333:graph/g-b52example",
        "createTime": 1700661225.003,
        "deletionProtection": false,
        "endpoint": "g-b52example.neptune-graph-gamma.us-east-1.amazonaws.com",
        "id": "g-b52example",
        "kmsKeyIdentifier": "AWS_OWNED_KEY",
        "name": "bobgraph",
        "provisionedMemory": 128,
        "replicaCount": 0,
        "status": "CREATING"
    }
},
"requestID": "4997ab5c-822d-4823-9c10-EXAMPLE",
"eventID": "58fc2480-7407-47f9-bc14-EXAMPLE",
"readOnly": false,
"eventType": "AwsApiCall",
"managementEvent": true,
"recipientAccountId": "111122223333",
"eventCategory": "Management"
}
```

- CreateGraph (Access Denied)

```
{
"eventVersion": "1.08",
"userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
    "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "sessionContext": {
        "sessionIssuer": {
            "type": "Role",
            "principalId": "AKIAI44QH8DHBEXAMPLE",
            "arn": "arn:aws:iam::444455556666:role/admin-role",
            "accountId": "444455556666",
            "userName": "bob"
        },
        "webIdFederationData": {},
        "attributes": {
            "creationDate": "2023-11-22T13:36:22Z",
            "mfaAuthenticated": "false"
        }
    }
},
"eventTime": "2023-11-22T13:36:22Z",
"eventSource": "neptune-graph.amazonaws.com",
"eventName": "CreateGraph",
"awsRegion": "us-east-1",
"sourceIPAddress": "192.0.2.0",
"userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0 botocore/1.10.63",
"errorCode": "AccessDenied",
"requestParameters": {
    "graphName": "bobgraph",
    "replicaCount": 0,
    "clientToken": "2040f466-220d-49e5-a45c-EXAMPLE",
    "allowFromPublic": false,
    "provisionedMemory": 128,
    "deletionProtection": false
},
"responseElements": {
    "Message": "User: arn:aws:sts::444455556666:assumed-role/bobrole/bobsession is
 not authorized to perform: neptune-graph:CreateGraph on resource: arn:aws:neptune-
graph:us-east-1:444455556666:graph/*"
},
"requestID": "89f04d5b-14d1-4c3a-b44d-EXAMPLE",
"eventID": "373c5468-99ac-4fed-9def-EXAMPLE",
"readOnly": false,
"eventType": "AwsApiCall",
"managementEvent": true,
"recipientAccountId": "111122223333",
"eventCategory": "Management"
```

- ListGraphs

```
{
"eventVersion": "1.08",
"userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
    "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "sessionContext": {
        "sessionIssuer": {
            "type": "Role",
            "principalId": "AKIAI44QH8DHBEXAMPLE",
            "arn": "arn:aws:iam::111122223333:role/admin-role",
            "accountId": "111122223333",
            "userName": "bob"
        },
        "webIdFederationData": {},
        "attributes": {
            "creationDate": "2023-11-22T13:34:55Z",
            "mfaAuthenticated": "false"
        }
    }
},
"eventTime": "2023-11-22T13:42:56Z",
"eventSource": "neptune-graph.amazonaws.com",
"eventName": "ListGraphs",
"awsRegion": "us-east-1",
"sourceIPAddress": "192.0.2.0",
"userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0 botocore/1.10.63",
"requestParameters": null,
"responseElements": null,
"requestID": "fb7e8333-61a3-4bcd-a6d5-EXAMPLE",
"eventID": "85002909-acf5-499f-a814-EXAMPLE",
"readOnly": true,
"eventType": "AwsApiCall",
"managementEvent": true,
"recipientAccountId": "111122223333",
"eventCategory": "Management"
}
```

- GetGraphSummary

```
{
"eventVersion": "1.09",
"userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
    "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
    "accountId": "111122223333",
    "sessionContext": {
        "sessionIssuer": {
            "type": "Role",
            "principalId": "AKIAI44QH8DHBEXAMPLE",
            "arn": "arn:aws:iam::111122223333:role/admin-role",
            "accountId": "111122223333",
            "userName": "bob"
        },
        "attributes": {
            "creationDate": "2023-11-22T13:34:55Z",
            "mfaAuthenticated": "false"
        }
    }
},
"eventTime": "2023-11-22T13:42:56Z",
"eventSource": "neptune-graph.amazonaws.com",
"eventName": "GetGraphSummary",
"awsRegion": "us-east-1",
"sourceIPAddress": "192.0.2.0",
"userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0 botocore/1.10.63",
"requestParameters": {
    "requestType": "GET",
    "requestParameters": {},
    "requestPayload": "/summary",
    "requestContentType": "application/json",
    "requestHeaders": {
        "content-length": "0",
        "Accept": "application/xml",
        "x-amz-date": "20231122T133455Z",
        "User-Agent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0 botocore/1.10.63",
        "Connection": "keep-alive",
        "X-Forwarded-For": "192.0.2.0",
        "Host": "localhost:8080",
        "Accept-Encoding": "gzip, deflate",
        "Content-Type": "application/json"
    },
    "authorizedIamActions": []
},
"responseElements": null,
"requestID": "fb7e8333-61a3-4bcd-a6d5-EXAMPLE",
"eventID": "85002909-acf5-499f-a814-EXAMPLE",
"readOnly": true,
"resources": [
    {
```

```
        "accountId": "111122223333",
        "type": "AWS::NeptuneGraph::Graph",
        "ARN": "arn:aws:neptune-graph:us-east-1:111122223333:graph/g-dbiexample"
    }
],
"eventType": "AwsApiCall",
"managementEvent": false,
"recipientAccountId": "111122223333",
"sharedEventID": "ce9ee550-df43-45a8-9445-EXAMPLE",
"eventCategory": "Data"
}
```

- ExecuteQuery

```
{
"eventVersion": "1.09",
"userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
    "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
    "accountId": "111122223333",
    "sessionContext": {
        "sessionIssuer": {
            "type": "Role",
            "principalId": "AKIAI44QH8DHBEXAMPLE",
            "arn": "arn:aws:iam::111122223333:role/admin-role",
            "accountId": "111122223333",
            "userName": "bob"
        },
        "attributes": {
            "creationDate": "2023-11-22T14:04:45Z",
            "mfaAuthenticated": "false"
        }
    }
},
"eventTime": "2023-11-22T14:04:41Z",
"eventSource": "neptune-graph.amazonaws.com",
"eventName": "ExecuteQuery",
"awsRegion": "us-east-1",
"sourceIPAddress": "192.0.2.0",
"userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0 botocore/1.10.63",
"requestParameters": {
    "requestType": "POST",
    "requestPayload": "[Redacted]",
    "requestContentType": "application/x-www-form-urlencoded",
    "requestHeaders": {
        "X-Amz-Date": "20231122T140445Z",
        "x-triton-proxy-request-id": "9dece4eb-7018-429f-970f-EXAMPLE",
        "Connection": "Keep-Alive",
        "User-Agent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0 botocore/1.10.63",
        "X-Forwarded-For": "192.0.2.0",
        "content-type": "application/x-www-form-urlencoded",
        "Host": "g-dbiexample.neptune-graph-gamma.us-east-1.amazonaws.com",
        "Accept-Encoding": "gzip,deflate",
        "Content-Length": "40"
    },
    "authorizedIamActions": []
},
"responseElements": {
    "responseTime": "2023-11-22T14:04:45.348Z",
    "responseCode": 200,
    "responseHeaders": {},
    "responseSize": "0",
    "responseContent": ""
},
"requestID": "37001a17-7d4e-4c34-9f3b-EXAMPLE",
"eventID": "5a6323fe-9ea2-4efe-81f0-EXAMPLE",
"readOnly": false,
"resources": [
    {
        "accountId": "111122223333",
        "type": "AWS::NeptuneGraph::Graph",
        "ARN": "arn:aws:neptune-graph:us-east-1:111122223333:graph/g-dbiexample"
    }
],
"eventType": "AwsApiCall",
"managementEvent": false,
"recipientAccountId": "111122223333",
"eventCategory": "Data"
}
```

Security in Neptune Analytics

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are 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 Neptune Analytics, 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 Analytics. The following topics show you how to configure Neptune Analytics 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 Analytics resources.

Topics

Data protection in Neptune Analytics

The AWS shared responsibility model applies to data protection in Neptune Analytics. 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 the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

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 Analytics 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.

Identity and access management for Neptune Analytics

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 Analytics resources. IAM is an AWS service that you can use with no additional charge.

Topics

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Neptune Analytics.

Service user – If you use the Neptune Analytics service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Neptune Analytics features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Neptune Analytics, see Troubleshooting Neptune Analytics identity and access.

Service administrator – If you're in charge of Neptune Analytics resources at your company, you probably have full access to Neptune Analytics. It's your job to determine which Neptune Analytics features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Neptune Analytics, see How Neptune Analytics works with IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Neptune Analytics. To view example Neptune Analytics identity-based policies that you can use in IAM, see Identity-based policy examples for Neptune Analytics.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see Signing AWS API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see When to create an IAM user (instead of a role) in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling an AWS CLI or

AWS API operation or by using a custom URL. For more information about methods for using roles, see Using IAM roles in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Creating a role for a third-party Identity Provider in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.
  - **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.

  - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see When to create an IAM role (instead of a user) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choosing between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries –** A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs) –** 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 Service control policies in the *AWS Organizations User Guide*.
- **Session policies –** Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How Neptune Analytics works with IAM

Before you use IAM to manage access to Neptune Analytics, learn what IAM features are available to use with Neptune Analytics.

IAM features you can use with Neptune Analytics

IAM feature	Neptune Analytics support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys	Yes
ACLs	No
ABAC (tags in policies)	Partial
Temporary credentials	Yes
Principal permissions	Yes
Service roles	Yes
Service-linked roles	Yes

To get a high-level view of how Neptune Analytics and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for Neptune Analytics

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for Neptune Analytics

To view examples of Neptune Analytics identity-based policies, see Identity-based policy examples for Neptune Analytics.

Resource-based policies within Neptune Analytics

Supports resource-based policies: No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Policy actions for Neptune Analytics

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of Neptune Analytics actions, see Actions Defined by Neptune Analytics in the *Service Authorization Reference*.

Policy actions in Neptune Analytics use the following prefix before the action:

```
neptune-graph
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
  "neptune-graph:action1",
  "neptune-graph:action2"
      ]
```

To view examples of Neptune Analytics identity-based policies, see Identity-based policy examples for Neptune Analytics.

Policy resources for Neptune Analytics

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of Neptune Analytics resource types and their ARNs, see Resources Defined by Neptune Analytics in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions Defined by Neptune Analytics .

To view examples of Neptune Analytics identity-based policies, see Identity-based policy examples for Neptune Analytics.

Policy condition keys for Neptune Analytics

Supports service-specific policy condition keys: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

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*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of Neptune Analytics condition keys, see Condition Keys for Neptune Analytics in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions Defined by Neptune Analytics .

To view examples of Neptune Analytics identity-based policies, see Identity-based policy examples for Neptune Analytics.

ACLs in Neptune Analytics

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

ABAC with Neptune Analytics

Supports ABAC (tags in policies): Partial

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see What is ABAC? in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using temporary credentials with Neptune Analytics

Supports temporary credentials: Yes

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see AWS services that work with IAM in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see Switching to a role (console) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM.

Cross-service principal permissions for Neptune Analytics

Supports forward access sessions (FAS): Yes

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

Service roles for Neptune Analytics

Supports service roles: Yes

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.

> ⚠ **Warning**
>
> Changing the permissions for a service role might break Neptune Analytics functionality. Edit service roles only when Neptune Analytics provides guidance to do so.

Service-linked roles for Neptune Analytics

Supports service-linked roles: Yes

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing Neptune Analytics service-linked roles, see Using service-linked roles (SLRs) in Neptune Analytics.

For details about creating or managing service-linked roles for other services, see AWS services that work with IAM. Find a service in the table that includes a `Yes` in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for Neptune Analytics

By default, users and roles don't have permission to create or modify Neptune Analytics resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Creating IAM policies in the *IAM User Guide*.

For details about actions and resource types defined by Neptune Analytics, including the format of the ARNs for each of the resource types, see Actions, Resources, and Condition Keys for Neptune Analytics in the *Service Authorization Reference*.

Topics

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Neptune Analytics resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.

- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see IAM Access Analyzer policy validation in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Configuring MFA-protected API access in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the Neptune Analytics console

To access the Neptune Analytics console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Neptune Analytics resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the Neptune Analytics console, also attach the Neptune Analytics *ConsoleAccess* or *ReadOnly* AWS managed policy to the entities. For more information, see Adding permissions to a user in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Troubleshooting Neptune Analytics identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Neptune Analytics and IAM.

Topics

- I am not authorized to perform an action in Neptune Analytics
- I am not authorized to perform iam:PassRole
- I want to allow people outside of my AWS account to access my Neptune Analytics resources

I am not authorized to perform an action in Neptune Analytics

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the mateojackson IAM user tries to use the console to view details about a fictional *my-example-widget* resource but doesn't have the fictional neptune-graph:*GetWidget* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 neptune-graph:GetWidget on resource: my-example-widget
```

In this case, the policy for the mateojackson user must be updated to allow access to the *my-example-widget* resource by using the neptune-graph:*GetWidget* action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the iam:PassRole action, your policies must be updated to allow you to pass a role to Neptune Analytics.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Neptune Analytics. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Neptune Analytics resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Neptune Analytics supports these features, see How Neptune Analytics works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Compliance validation for Neptune Analytics

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. AWS provides the following resources to help with compliance:

- Security and Compliance Quick Start Guides – These deployment guides discuss architectural considerations and provide steps for deploying baseline environments on AWS that are security and compliance focused.
- Architecting for HIPAA Security and Compliance on Amazon Web Services – This whitepaper describes how companies can use AWS to create HIPAA-eligible applications.

> **Note**
>
> Not all AWS services are HIPAA eligible. For more information, see the HIPAA Eligible Services Reference.

- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your

compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.

- Amazon GuardDuty – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Resilience in Neptune Analytics

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 zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, Neptune Analytics offers several features to help support your data resiliency and backup needs.

Infrastructure Security in Neptune Analytics

As a managed service, Neptune Analytics is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access Neptune Analytics through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Cross-service confused deputy prevention

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

We recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies to limit the permissions that ServiceNameLongEntity gives another service to the resource. Use `aws:SourceArn` if you want only one resource to be associated with the cross-service access. Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. If you don't know the full ARN of the resource or if you are specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcard characters (*) for the unknown portions of the ARN. For example, `arn:aws:servicename:*:123456789012:*`.

If the `aws:SourceArn` value does not contain the account ID, such as an Amazon S3 bucket ARN, you must use both global condition context keys to limit permissions.

The value of `aws:SourceArn` must be ResourceDescription.

The following example shows how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in ServiceNameEntity to prevent the confused deputy problem.

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Sid": "ConfusedDeputyPreventionExamplePolicy",
    "Effect": "Allow",
    "Principal": {
      "Service": "servicename.amazonaws.com"
    },
    "Action": "servicename:ActionName",
    "Resource": [
      "arn:aws:servicename:::ResourceName/*"
    ],
    "Condition": {
      "ArnLike": {
        "aws:SourceArn": "arn:aws:servicename:*:123456789012:*"
      },
      "StringEquals": {
        "aws:SourceAccount": "123456789012"
      }
    }
  }
}
```

Using service-linked roles (SLRs) in Neptune Analytics

Neptune Analytics graphs use 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 Analytics graphs. Service-linked roles are predefined by Neptune Analytics graphs and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes using Neptune Analytics graphs easier because you don't have to add the necessary permissions manually. Neptune Analytics defines the permissions in its service-linked roles, and unless defined otherwise, only Neptune Analytics graphs 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 Analytics graph resources because you can't inadvertently remove the permissions 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 are marked with **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 Analytics Graphs

Neptune Analytics graphs uses the service-linked role named `AWSServiceRoleForNeptuneGraph` to allow them to call AWS services on behalf of your DB clusters.

This service-linked role has an IAM managed permissions policy attached to it named AWSServiceRoleForNeptuneGraphPolicy that grants it permissions to operate in your account. See AWS managed policies for Amazon Neptune. This policy provides read-only access to all Amazon Neptune Analytics resources along with read-only permissions for dependent services, as follows:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "GraphMetrics",
      "Effect": "Allow",
      "Action": [
        "cloudwatch:PutMetricData"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "cloudwatch:namespace": [
            "AWS/Neptune",
            "AWS/Usage"
          ]
        }
      }
    },
    {
      "Sid": "GraphLogGroup",
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup"
      ],
      "Resource": [
        "arn:aws:logs:*:*:log-group:/aws/neptune/*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:ResourceAccount": "${aws:PrincipalAccount}"
        }
      }
    },
    {
      "Sid": "GraphLogEvents",
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams"
      ],
      "Resource": [
        "arn:aws:logs:*:*:log-group:/aws/neptune/*:log-stream:*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:ResourceAccount": "${aws:PrincipalAccount}"
        }
      }
    }
  ]
}
```

Note

To allow an IAM entity such as a user, group, or role to be able to create, edit, or delete a service-linked role, you must set the appropriate permissions, like this:

```
{
  "Action": "iam:CreateServiceLinkedRole",
  "Effect": "Allow",
  "Resource": "arn:aws:iam::*:role/aws-service-role/neptune-graph.amazonaws.com/
AWSServiceRoleForNeptuneGraph",
  "Condition": {
    "StringLike": {
        "iam:AWSServiceName":"neptune-graph.amazonaws.com"
    }
}
```

If those permissions have not been set, or have not yet propagated, you may receive the following error message when you try to create a service-linked role:

```
Unable to create the resource. Verify that you have permission
to create service linked role. Otherwise wait and try again later.
```

For more information, see Service-linked role permissions in the IAM User Guide.

Creating a service-linked role for Neptune Analytics

You don't have to create a service-linked role manually for Neptune Analytics. When you create a graph, Neptune Analytics automatically creates the service-linked role for you.

Editing a service-linked role for Neptune Analytics

Neptune Analytics doesn't allow you to edit the `AWSServiceRoleForNeptuneGraph` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference it. However, you can edi t 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

If you no longer need to use a feature or service that requires a service-linked role, it's best to delete that role so you don't have an unused entity that is not actively monitored or maintained.

However, before you can delete the service-linked role, you must first confirm that the role has no active sessions, and remove any resources that it uses.

To check whether a 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://console.aws.amazon.com/iam/.
2. In the navigation pane of the IAM console, choose *`Roles`*. Then choose the name (not the check box) of the `AWSServiceRoleForNeptuneGraph` role.
3. On the **Summary** page for the chosen role, choose the **Access Advisor** tab.

> **Note**
>
> If you are unsure whether Neptune Analytics is using the `AWSServiceRoleForNeptuneGraph` role, you can try to delete the role. If the service is using the role, then the deletion fails and you can view the AWS 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.

To delete your clusters so that you can delete `AWSServiceRoleForNeptuneGraph`

1. Open the Neptune console at https://console.aws.amazon.com/neptune/.
2. In the navigation pane, choose **Graphs**.
3. Choose a cluster that you want to delete.
4. For **Actions**, choose **Delete**.
5. If you are prompted to **Create final Snapshot?**, choose **Yes** or **No**. If you choose **Yes** enter the name of your final snapshot for **Final snapshot name**.
6. Choose **Delete**.

You can use the IAM console, the IAM CLI, or the IAM API to delete the `AWSServiceRoleForNeptuneGraph` service-linked role. For more information, see Deleting a service-linked role in the IAM User Guide.

Querying Neptune Analytics

Neptune Analytics currently supports only the openCypher query language to access a graph. 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.

Topics

Query APIs

The Neptune Analytics data API provides support for data operations including query execution, query status checking, query cancellation, and graph summarizing via the HTTPS endpoint, the AWS CLI, and the SDK.

Topics

ExecuteQuery

ExecuteQuery runs queries against a Neptune Analytics graph. Supported language: openCypher.

ExecuteQuery inputs

- graph-identifier (required)

  Type: `String`

  The identifier representing a graph.
- region (required)

  Type: `String`

  The region where the graph is present.
- query-string (required)

  Type: `String`

  Default: none

  A string representing a query.
- language (required)

  Type: `Enum`

  Default: none

  The query language the query is written in. Currently, only `OPEN_CYPHER` is supported.
- parameters (optional)

  Type: `Map`

  A map from `String` to `String` where the key is the parameter name and the value is the parameter value.
- plan-cache (optional)

  Type: `Enum`

  Query plan cache is a feature that saves the query plan and reuses it on successive executions of the same query, reducing query latency. Query plan cache works for both read-only and mutation queries. The plan cache is an LRU cache with a five minute TTL and a capacity of 1000. It supports the following values:

  - `AUTO`: The engine will automatically decide to cache the query plan. If the query is parameterized and the runtime is shorter than 100ms, the query plan is automatically cached.
  - `ENABLED`: The query plan is cached regardless of the query runtime. The plan cache uses the query string as the key, this means that if a query is slightly different (i.e. different constants), it will not be able to reuse the plan cache of similar queries.
  - `DISABLED`: The query plan cache is not used.

  For more information on the query plan cache, see Query plan cache.

- explain-mode (optional)

  Type: `Enum`

  The explain mode parameters allow getting a query explain instead of the actual query results. A query explain can be used to gather insights about the query execution such as planning decisions, time spent on each operator, number of records flowing etc. If this parameter is not set the query is executed normally and the result is returned. The acceptable values for query explain are:

  - `STATIC`: Returns a query explain without executing the query. This can give an estimate on what the query plan looks like without actually executing the query. The static query plan may differ from the actual query plan. Actual queries may make planning decisions based on runtime statistics, which may not be considered when fetching a static query plan. A static query plan is useful when it is necessary to observe a plan for a query that either does not complete or runs for too long.
  - `DETAILS`: Returns a detailed query plan that shows what the running query did. This includes information such as operators runtime, number of records flowing through the plan, runtime planning decisions and more. If a query does not succeed in `NONE` mode, it will not succeed in `DETAILS` mode either. In this instance, you would want to use `STATIC` mode.

  For more information on query explain and its output, see Query explain.

- query-timeout-milliseconds (optional)

  Type: `Enum`

If specified, provides an upper bound to the query run time. This parameter will override the graph default timeout (30 minutes). Neptune Analytics graph have a maximum query runtime of 60 minutes. If the specified timeout is greater than the maximum query runtime, the query will only run for the maximum query runtime.

- Using the default settings, any CLI or SDK request will timeout in 60 seconds and attempt a retry. For the cases where you are running queries that can take longer than 60 seconds, it is recommended to set the CLI/SDK timeout to `0` (no timeout), or a much larger value to avoid unnecesssary retries.

  It is also recommended to set `MAX_ATTEMPTS` for CLI/SDK to 1 for `execute_query` to avoid any retries by CLI/SDK.

  For the Boto client, set the `read_timeout` to `None`, and the `total_max_attempts` to `1`.

  ```
  import boto3
  from botocore.config import Config
  n = boto3.client('neptune-graph',
                   config=(Config(retries={"total_max_attempts": 1, "mode":
   "standard"}, read_timeout=None)))
  ```

  For the CLI, set the `--cli-read-timeout` parameter to `0` for no timeout, and set the environment variable `AWS_MAX_ATTEMPTS` to 1 to prevent retries.

  ```
  export AWS_MAX_ATTEMPTS=1
  ```

  ```
  aws neptune-graph execute-query \
  --graph-identifier <graph-id> \
  --region <region> \
  --query-string "MATCH (p:Person)-[r:KNOWS]->(p1) RETURN *;" \
  --cli-read-timeout 0
  --language open_cypher /tmp/out.txt
  ```

ExecuteQuery examples

AWS CLI

```
# Sample query
```

```
aws neptune-graph execute-query \
--graph-identifier <graph-id> \
--region <region> \
--query-string "MATCH (p:Person)-[r:KNOWS]->(p1) RETURN *;" \
--language open_cypher \
/tmp/out.txt

# Sample query that prints directly to the console.
aws neptune-graph execute-query \
--graph-identifier <graph-id> \
--region <region> \
--query-string "MATCH (p:Person)-[r:KNOWS]->(p1) RETURN *;" \
--language open_cypher \
/dev/stdout

# parameters supported
query-string [REQUIRED] : String
language [REQUIRED] : open_cypher
explain-mode [OPTIONAL] : static | details
query-timeout-milliseconds [OPTIONAL] : Integer
plan-cache [OPTIONAL] : enabled | disabled | auto
parameters [OPTIONAL] : Map
```

AWSCURL

```
# Sample query
awscurl -X POST "https://<graph-id>.<endpoint>/queries" \
-H "Content-Type: application/x-www-form-urlencoded" \
--region <region> \
--service neptune-graph \
-d "query=MATCH (p:Person)-[r:KNOWS]->(p1) RETURN *;"
```

ExecuteQuery output

```
{
  "results": [{
      "p": {
        "~id": "fa1ef9b0-fa32-4b37-8051-78f2bf0e0d63",
        "~entityType": "node",
        "~labels": ["Person"],
        "~properties": {
          "name": "Simone"
        }
      },
      "p1": {
        "~id": "edaded10-b22b-4818-a22e-ddebfcf37acb",
        "~entityType": "node",
        "~labels": ["Person"],
        "~properties": {
          "name": "Mirro"
        }
      },
      "r": {
        "~id": "neptune_reserved_1_1154145192329347075",
        "~entityType": "relationship",
        "~start": "fa1ef9b0-fa32-4b37-8051-78f2bf0e0d63",
        "~end": "edaded10-b22b-4818-a22e-ddebfcf37acb",
        "~type": "KNOWS",
        "~properties": {}
      }
    }]
}
```

Parameterized queries

Neptune Analytics supports parameterized openCypher queries. This allows you to use the same query structure multiple times with different arguments. Since the query structure doesn't change, Neptune Analytics tries to cache the plan for these parameterized queries that run in less than 100 milliseconds.

The following 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 definied as follows:

```
parameters={"name": "john", "age": 20}
```

AWS CLI

```
# Sample query
```

```
aws neptune-graph execute-query \
--graph-identifier <graph-id> \
--region <region> \
--query-string "MATCH (n {name: \$name, age: \$age}) RETURN n" \
--parameters "{\"name\": \"john\", \"age\": 20}"
--language open_cypher /tmp/out.txt
```

AWSCURL

```
# Sample query
awscurl -X POST "https://[graph-id].<endpoint>/queries" \
-H "Content-Type: application/x-www-form-urlencoded" \
--region <region> \
--service neptune-graph \
-d "query=MATCH (n {name: \$name, age: \$age}) RETURN n;&parameters={\"name\":
 \"john\", \"age\": 20}"
```

ListQueries

ListQueries API fetches the list of running/waiting/cancelling queries on the graph.

ListQueries syntax

```
aws neptune-graph list-queries \
    --graph-identifier <graph-id> \
    --region <region> \
    --max-results <result_count>
    --state [all | running | waiting | cancelling]
```

ListQueries inputs

- graph-identifier (required)

  Type: `String`

  Identifier representing your graph.

- region (required)

  Type: `String`

  Region where the graph is present.

- max-results (required)

  Type: `Integer`

  The maximum number of results to be fetched by the API.
- state (optional)

  Type: `String`

  Supported values: all | running | waiting | cancelling

  If `state` parameter is not specified, the API fetches all types.

ListQueries outputs

```
# Sample Response
{
    "queries": [
        {
            "id": "130ab841-8b4b-46c3-afbe-af00274c7fd9",
            "queryString": "MATCH p=(n)-[*]-(m) RETURN p;",
            "waited": 0,
            "elapsed": 1686,
            "state": "RUNNING"
        }
    ]
}
```

The output contains a list of query objects, each containing:

- id: `String` - representing the unique identifier of the query.
- queryString: `String` - The actual query text. The queryString may be truncated if the actual query string is too long.
- waited: `Integer` - The time in milliseconds for which the query has waited in the waiting queue before being picked up by a worker thread.
- elapsed: `Integer` - The time in milliseconds representing the running time of the query.
- state: Current state of the query (running | waiting | cancelling).

The default list order is queries that are `running`, followed by `waiting` and `cancelling`.

ListQueries Examples

AWS CLI

```
aws neptune-graph list-queries \
    --graph-identifier <graph-id> \
    --region us-east-1 \
    --max-results 200
    --state waiting
```

AWSCURL

```
awscurl -X GET "https://<graph-id>.<endpoint>/queries?state=WAITING&maxResults=200"
 \
   -H "Content-Type: application/x-www-form-urlencoded" \
   --region us-east-1 \
   --service neptune-graph
```

GetQuery

The GetQuery API can be used to get the status of a specific query request.

GetQuery inputs

- graph-identifier (required)

  Type: `String`

  The identifier representing a graph.
- region (required)

  Type: `String`

  The region where the graph is present.
- query-id (required)

  Type: `String`

  The id of the query request for which you want to get information.

GetQuery outputs

- id: The same id used in this request.
- queryString: Non-truncated query string associated to this `query-id`.
- waited: Time in milliseconds this query request had to wait to be executed.
- elapsed: Time in milliseconds the query spent while in execution.
- state: Current state of the query - running | waiting | cancelling.

```
{
    "id" : "d6873456-40a7-44d7-be5c-46b4acfdc171",
    "queryString" : "UNWIND range(1,100000) AS i MATCH (n) RETURN i, n",
    "waited" : 1,
    "elapsed" : 8645,
    "state" : "RUNNING"
}
```

GetQuery examples

AWS CLI

```
aws neptune-graph get-query \
    --graph-identifier <graph-id> \
    --region <region> \
    --query-id <query-id>
```

AWSCURL

```
awscurl -X GET "https://<graph-id>.<endpoint>/queries/<query-id>" \
   -H "Content-Type: application/x-www-form-urlencoded" \
   --region us-east-1 \
   --service neptune-graph
```

CancelQuery

CancelQuery cancels a specific query request.

CancelQuery inputs

- graph-identifier (required)

  Type: `String`

  The identifier representing a graph.
- region (required)

  Type: `String`

  The region where the graph is present.
- query-id (required)

  Type: `String`

  The id of the query request for which you want to cancel.

CancelQuery outputs

CancelQuery does not have any output.

CancelQuery examples

AWS CLI

```
aws neptune-graph cancel-query \
    --graph-identifier <graph-id> \
    --region <region> \
    --query-id <query-id>
```

AWSCURL

```
awscurl -X DELETE "https://<graph-id>.<endpoint>/queries/<query-id>"  --region us-
east-1 --service neptune-graph
```

GraphSummary

You can use the GetGraphSummary API to quickly gain a high-level understanding of your graph data, size and content. In a graph application, this API can be used to improve the search results by providing discovered node or edge labels as part of the search.

The GetGraphSummary API retrieves a read-only list of node and edge labels and property keys, along with counts of nodes, edges, and properties. The API also accepts an optional parameter named mode, which can take one of two values, namely basic (the default) and detailed. The detailed graph summary response contains two additional fields, nodeStructures and edgeStructures.

GetGraphSummary inputs

GetGraphSummary accepts two inputs:

- graph-identifier (required) - The unique identifier of the graph.
- mode (optional) - Can be `basic` or `detailed`.

GetGraphSummary outputs

The response contains the following fields:

- `version` - The version of this graph summary response.
- `lastStatisticsComputationTime` - The timestamp, in ISO 8601 format, of the time at which Neptune Analytics 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 disctinct 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.
  - `numEdgeProperites` - 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` (only present for mode=detailed) - Contains a list of node structures, each containing 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` (only present for mode=detailed) - Contains a list of edge structures each containing the following fields:
  - `count` - Number of edges that have this specific structure.
  - `edgeProperties` - List of edge properties present in this specific structure.

GetGraphSummary examples

AWS CLI

```
# Sample query
aws neptune-graph get-graph-summary \
--graph-identifier <graph-id> \
--region <region>
--mode detailed

# parmeters supported
mode [Optional] : basic | detailed
```

AWSCURL

```
# Sample query
awscurl "https://<graph-id>.<endpoint>/summary" \
-H "Content-Type: application/x-www-form-urlencoded" \
--region <region> \
--service neptune-graph
```

Sample output payload:

```
# this is the graph summary with "mode=detailed"
{
    "version": "v1",
    "lastStatisticsComputationTime": "2024-01-25T19:50:42+00:00",
    "graphSummary": {
        "numNodes": 3749,
        "numEdges": 57645,
        "numNodeLabels": 4,
        "numEdgeLabels": 2,
        "nodeLabels": [
            "continent",
            "country",
            "version",
            "airport"
        ],
        "edgeLabels": [
            "contains",
            "route"
        ],
        "numNodeProperties": 14,
        "numEdgeProperties": 1,
        "nodeProperties": [
            {
                "code": 3749
            },
            {
                "desc": 3749
            },
            {
                "type": 3749
            },
            {
                "city": 3504
            },
            {
                "country": 3504
            },
            {
                "elev": 3504
            },
            {
                "icao": 3504
            },
            {
                "lat": 3504
            },
            {
                "lon": 3504
            },
            {
                "longest": 3504
            },
            {
                "region": 3504
            },
            {
                "runways": 3504
            },
            {
                "author": 1
            },
            {
                "date": 1
            }
        ],
        "edgeProperties": [
            {
                "dist": 50637
            }
        ],
        "totalNodePropertyValues": 42785,
        "totalEdgePropertyValues": 50637,
        "nodeStructures": [          // will not be present with mode=basic
            {
                "count": 3475,
                "nodeProperties": [
                    "city",
                    "code",
                    "country",
                    "desc",
                    "elev",
                    "icao",
                    "lat",
                    "lon",
                    "longest",
                    "region",
                    "runways",
                    "type"
                ],
                "distinctOutgoingEdgeLabels": [
                    "route"
                ]
            },
            {
                "count": 238,
                "nodeProperties": [
                    "code",
                    "desc",
                    "type"
                ],
                "distinctOutgoingEdgeLabels": [
                    "contains"
                ]
            },
            {
                "count": 29,
                "nodeProperties": [
                    "city",
                    "code",
                    "country",
                    "desc",
                    "elev",
                    "icao",
                    "lat",
                    "lon",
                    "longest",
                    "region",
                    "runways",
                    "type"
                ],
                "distinctOutgoingEdgeLabels": []
            },
            {
                "count": 6,
                "nodeProperties": [
                    "code",
                    "desc",
                    "type"
                ],
                "distinctOutgoingEdgeLabels": []
            },
            {
                "count": 1,
                "nodeProperties": [
                    "author",
                    "code",
                    "date",
                    "desc",
                    "type"
                ],
                "distinctOutgoingEdgeLabels": []
            }
        ],
        "edgeStructures": [          //will not be present with mode=basic
            {
                "count": 50637,
                "edgeProperties": [
                    "dist"
                ]
            }
        ]
    }
}
```

IAM role mappings

When you're calling Neptune Analytics API methods on a cluster, you require 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
ListQueries	Action: ListQueries
GetQuery	Action: GetQueryStatus
Cancel Query	Action: CancelQuery
GetGraphSummary	Action: GetGraphSummary
ExecuteQuery	Action: ReadDataViaQuery Action: WriteDataViaQuery Action: DeleteDataViaQuery

For more information, see Actions, resources and condition keys for Neptune Analytics.

Query plan cache

When a query is submitted to Neptune , the query string is parsed and translated into a query plan, which then gets optimized and executed by the engine. Often, the applications are backed by common query patterns that are instantiated with different values, and query plan cache would be optimal to reduce latency of those common query patterns. The query plan cache does this by storing a parameterized version of frequently used query plans (at most 1000 at any point), which gets reused and instantiated properly based on new parameter values provided, if any.

⚠ **Warning**

We have detected an issue in query plan cache when `skip` or `limit` is used in an inner `WITH` clause and are parameterized. 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. For more information see Mitigation for query plan cache issue.

Why use the query plan cache?

Reusing the query plan can reduce the latency, as the later executions skip parsing and optimization steps.

Where can it be used?

Query plan cache can be used for all type of queries. By default, it automatically caches plan for low-latency parameterized queries, whose execution time is less than 100ms.

How to force enable/disable the query plan cache?

For read-only queries, query plan cache is enabled by default for low-latency queries. A plan is cached only when latency is lower than the threshold of 100ms. This behavior can be overridden on a per-query basis by HTTP parameter. HTTP parameter `--plan-cache` can take `enabled` or `disabled` as a value.

```
# Forcing plan to be cached or reused
% aws neptune-graph execute-query \
   --graph-identifier <graph-id> \
   --query-string "MATCH (n) RETURN n LIMIT 1"
   --region <region> \
   --plan-cache "enabled"
   --language open_cypher /tmp/out.txt

% aws neptune-graph execute-query \
   --graph-identifier <graph-id> \
   --query-string "RETURN \$arg"
   --region <region> \
   --plan-cache "enabled" \
   --parameters "{\"arg\": 123}"
   --language open_cypher /tmp/out.txt
```

How to check if a plan is cached?

To check if a plan is cached, use `explain`. For read-only queries, if the query was submitted and the plan was cached, explain would show explain details relevant to the query plan cache.

```
% aws neptune-graph execute-query \
   --graph-identifier <graph-id> \
   --query-string "MATCH (n) RETURN n LIMIT 1"
   --region <region> \
   --plan-cache "enabled" \
   --explain-mode "static" \
    --language open_cypher /tmp/out.txt
```

```
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).

> **Note**
>
> For a mutation query, explain is not yet supported.

Eviction

A query plan is evicted by cache TTL or maximum number of cached query plans reached. When the query plan is hit, the TTL is refreshed. The defaults are:

- The maximum number of plans cached per instance is 1000.
- TTL: 300_000 milliseconds or 5 minutes. Note that cache hit refreshes the TTL back to 5 min.

Conditions when a query plan is not cached

The following list demonstrates conditions for when a query plan would not be cached.

- If submitted with query-specific parameter `--plan-cache "disabled"`.
  - If a cache is wanted, you can rerun the query without `--plan-cache "disabled"`.
- If the query evaluation time is larger than latency threshold, it’s not cached since it’s a long-running query and is considered to not benefit from query plan cache.
- If the query contains pattern that does not return any results.
  - i.e. `MATCH (n:nonexistentLabel) return n` when there are zero nodes with specified label.

  - i.e. `MATCH (n {name: $param}) return n with parameters={"param": "abcde"}` when there are zero nodes with name=abcde.
- If the query parameter is composite type (list, map).

```
aws neptune-graph execute-query \
   --graph-identifier <graph-id> \
   --query-string "RETURN \$arg"
   --region <region> \
   --plan-cache "enabled" \
   --parameters "{\"arg\": [1, 2, 3]}"
   --language open_cypher /tmp/out.txt

 aws neptune-graph execute-query \
   --graph-identifier <graph-id> \
   --query-string "RETURN \$arg"
   --region <region> \
   --plan-cache "enabled" \
   --parameters "{\"arg\": {\"a\": 1}}"
   --language open_cypher /tmp/out.txt
```

- If the query parameter is a string that has not been part of data load or data insertion.
  - If `CREATE (n {name: "X"})`, is done to insert “X”.
  - `RETURN “X”` is cached, while `RETURN “Y”` isn’t, as “Y” has not been inserted and does not exist in the database.

Mitigation for query plan cache issue

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 HTTP parameter `planCache=disabled` or SDK parameter `-\-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, or add a random comment to create a new plan for each request.

Option 1: Using request parameter

Curl example

```
curl -k https://<endpoint>:8182/opencypher -d 'query=MATCH (n:Person) WHERE n.age >
 $age WITH n skip $skip LIMIT $limit RETURN n.name, n.age' -d 'parameters={"age": 21,
 "skip": 2, "limit": 3}' -d planCache=disabled
```

SDK example

```
aws neptune-graph execute-query \
   -\-graph-identifier <graph-id> \
   -\-query-string "MATCH (n:Person) WHERE n.age > $age WITH n skip $skip LIMIT $limit
 RETURN n.name, n.age"
   -\-region <region> \
   -\-plan-cache "disabled" \
   -\-language open_cypher
```

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}
```

Option 3: Using a random comment

```
MATCH (n:Person)
WHERE n.age > $age
WITH n skip $skip LIMIT $limit
RETURN n.name, n.age // 411357f6-00d2-4f03-92ce-060d8e037c0b
```

```
parameters={"age": 21, "skip": 2, "limit": 3}
```

Query explain

The openCypher `explain` feature is a feature that helps users to understand how the query is executed. Usually this is used in the context of query performance analysis.

Explain inputs

To invoke `explain`, you can pass the explain-mode parameter to an ExecuteQuery request specifying the desired explain mode (i.e., level of detail), where this explain mode value can be one of the following:

- `static` - In static mode, `explain` doesn't run the query, but instead prints only the static structure of the query plan.
- `details` - In details mode, `explain` 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. Additional details, such as the actual openCypher query string and the estimated range count for the pattern underlying a join operator, are also shown.

The following code examples provide the `explain-mode` when using either the AWS CLI or AWSCURL.

AWS CLI

```
aws neptune-graph execute-query \
--region <region> \
--graph-identifier <graph-id> \
--query-string <query-string> \
--explain-mode <explain-mode> \
--language open_cypher /tmp/out.txt
```

AWSCURL

```
awscurl -X POST "https://<graph-id>.<endpoint>/queries" \
-H "Content-Type: application/x-www-form-urlencoded" \
--region <region> \
--service neptune-graph \
-d "query=<query>&explain=<mode>"
```

Explain outputs

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 by functor in the arguments section, on the value stored in the specified variable
- DFEAlgoWriteProperty – Explain operator for the property-writing portion of mutate algorithm invocations.
- DFEBFSAlgo – Explain operator for invocations of the Breadth First Search algorithm, which searches for nodes from a starting vertex (or starting vertices, also called multi-source BFS) in a graph in breadth-first order.
- DFEBindRelation – Binds together variables with the specified names.
- DFEChunkLocalSubQuery – This is a non-blocking operation that acts as a wrapper around subqueries being performed.
- DFEClosenessCentralityAlgo – Explain operator for invocations of the Closeness Centrality algorithm, which computes a metric that can be used as a positive measure of how close a given node is to all other nodes or how central it is in the graph.
- DFECommonNeighborsAlgo – Explain operator for invocations of the Common Neighbors algorithm, which counts the number of common neighbors of two input nodes.
- DFECreateConstant – Extends the given input relation with new columns containing constant values.
- DFEDegreeAlgo – Explain operator for invocations of the Degree algorithm, which calculates the number of edges that are incident to a vertex.

- 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 – This is a blocking operation that organizes the rows of a relation according to a set of variables, outputting a single group identifier column that is one-to-one with the rows of the input relation. Groups here are defined by the join variables used to build the hash index (See DFEHashIndexBuild for where this hash index might be built.)
- 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 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.)
- DFEJaccardSimilarityAlgo – Explain operator for invocations of the Jaccard similarity algorithm, which measures the similarity between two sets of nodes.
- 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.
- DFELabelPropagationAlgo – Explain operator for invocations of the Label Propagation algorithm, which is used for community detection.
- DFELoopSubQuery – 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 join variables across both relations, then this operator simply returns the left hand input relation as is.

- 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 join variables, then this operator will return an empty relation.
- DFEOptionalJoin – Performs the optional join A OPTIONAL B ≡ (A JOIN B) UNION (A MINUS_NE B). This is a blocking operation.
- DFEOverlapSimilarityAlgo – Explain operator for invocations of the Overlap Similarity algorithm, which measures the overlap between the neighbors of two nodes.
- DFEPageRankAlgo – Explain operator for invocations of the Page Rank algorithm, which calculates a score for a given node based on the number, quality, and importance of the edges pointing to that node.
- 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 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.
- DFESCCAlgo – Explain operator for invocations of the Strongly Connected Components algorithm, which calculates the maximally connected subgraphs of a directed graph where every node is reachable from every other node.
- 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).
- DFESSSPAlgo – Explain operator for invocations of the single source shortest path (SSSP) algorithms (Delta-stepping and Bellman-ford).
- 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.
- DFETopKSSSPAlgo – Explain operator for invocations of the TopK hop-limited single source (weighted) shortest path algorithm algorithm, which finds the single-source weighted shortest paths from a source node to its neighbors out to the distance specified by maxDepth.
- DFETotalNeighborsAlgo – Explain operator for invocations of the Total Neighbors algorithm, which counts the total number of unique neighbors of two input vertices.
- DFEUnfold – 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.
- DFEVSSAlgo – Explain operator for invocations of the Vector similarity search algorithms, which find similar vectors based on the distance to each other.
- DFEWCCAlgo – Explain operator for invocations of the Weakly Connected Components algorithm, which finds the weakly-connected components in a directed graph.

- SolutionInjection – Appears before everything else in the explain output, with a value of one in the `Units Out` column. However, it serves a no-op, and doesn't actually inject any solutions into the DFE engine.
- TermResolution – Appears at the end of plans and translates of objects from the Neptune engine into openCypher objects.

Columns in openCypher `explain` output

The query plan information generated 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.

Note

Depending on the level of detail selected via the explain mode parameter, some of these columns may not appear in the output.

Explain examples

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:

```
## sample query
aws neptune-graph execute-query \
--region <region> \
--graph-identifier <graph-id> \
--query-string "MATCH (n {code: 'ATL'}) RETURN n" \
--explain-mode details \
--language open_cypher /tmp/out.txt

## 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  # -      # -      # DFESubquery           # subQuery=subQuery1 # -    # 0         #
 0         # 0.00  # 8.00      #
###############################################################################################
#    #        #        # Summed execution time #                    #      #           #
           #       # 8.00      #
###############################################################################################


subQuery1
###############################################################################################
# ID # Out #1 # Out #2 # Name                   # Arguments
                            # Mode # Units In # Units Out # Ratio # Time (ms) #
###############################################################################################
# 0  # 1      # -      # DFEPipelineScan (DFX)  # pattern=project ?n ?n_code2 (?
n,code,?n_code2) [VERTEX_PROPERTY] # -    # 0         # 1         # 0.00  # 0.03      #
#    #        #        #                        # inlineFilters=[(?n_code2 IN
 ["ATL"^^xsd:string])]                #      #          #           #       #
 #
```

```
#    #        #        #                        # patternEstimate=1
                            #      #          #           #       #           #
###############################################################################################
# 1  # 2      # -      # DFEProject (DFX)       # columns=[?n]
                            # -    # 1        # 1         # 1.00  # 0.03      #
###############################################################################################
# 2  # 3      # -      # DFESerialize (DFX)     # columnsToSerialize=[?n]
                            # -    # 1        # 0         # 0.00  # 0.08      #
###############################################################################################
# 3  # -      # -      # DFEDrain (DFX)         # -
                            # -    # 0        # 0         # 0.00  # 0         #
###############################################################################################
#    #        #        # Summed execution time  #
                            #      #          #           #       # 0.15      #
###############################################################################################
```

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 is the `DFESubquery` operator. `DFESubquery` encapsulates the parts of the query execution plan that are 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`.

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 of the operators start with `DFE`.

Inside `subQuery1`, there can be zero (as in this case) or more `DFEChunkLocalSubQuery` or `DFELoopSubQuery` operators that encapsulate a part of the pushed execution plan that is executed in a memory-bounded mechanism. A `DFEChunkLocalSubQuery` 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 vertices `?n` with property `code` saved as a variable `?n_code2`. The `inlineFilters` argument shows the filtering for the `code` property equaling `ATL`.

Next, the `DFEProject` operator propagates forward only the `?n` variable we're interested in. Finally, the `DFESerialize` operator performs result serialization, transforming the input solutions into a readable format.

Statistics

Neptune Analytics uses similar statistics for planning query execution as in Neptune Database. Computing these statistics is performed as an integrated part of the Neptune Analytics storage system. There are a number of differences between the features and usage of statistics between Neptune Analytics and Neptune Database:

1. Initial statistics generation is performed as part of either the initial import task or an initial data load occurring before any query-driven updates. Subsequently, statistics re-computation is triggered automatically based on the amount of update operations performed by the database.
2. Like with Neptune Database, Neptune Analytics has a size limit for statistics data, beyond which statistics will be disabled. The number of predicate statistics, may not exceed one million (the same as Neptune Database). There is no hard limit on the number of characteristic sets present in the underlying data. However, beyond 10,000 characteristic sets, the system will begin to merge statistics data in order to limit the overall size of data being managed.
3. Statistics generation is fully managed by the storage system. There are no APIs to disable or re-compute statistics.
4. There are no CloudWatch metrics relating to statistics generation.

Exceptions

The following table lists query-side exceptions that could be encountered while using a query.

Neptune Analytics error code	HTTP status	Retriable	Description
Validation Exception	400	No	Something is wrong with the required information - Eg. a malformed query.

Neptune Analytics error code	HTTP status	Retriable	Description
AccessDeniedException	403	No	User is not authorized to perform the requested operation.
ResourceNotFoundException	404	No	Requested resource is not available.
ThrottlingException	429	Yes	The server has received too many concurrent requests.
InternalServerErrorException	500	Yes	The server failed to process the request for an unknown reason.
UnprocessableException	422	No	Request cannot be processed due to known reasons - Eg. The query timed out.
ConflictException	409	Yes	Concurrently running queries attempted to modify resources or data records concurrently and the conflict could not be resolved automatically. Please retry with an exponential back-off strategy.

Neptune Analytics openCypher data model

For details on the openCypher data model, please refer to the Neptune Database documentation. There are some differences in modeling of vertices without labels. Neptune Database adds vertices with a default label if one is not explicitly provided. All but the last label of a vertex can be deleted.

What is a vertex?

As well as loading both vertices and edges, unlike Neptune Database, Neptune Analytics also allows loading just edges and is still able to run algorithms and queries from that starting point. This is useful if your main interest is, for example, loading a file of edge data from a CSV file and running an algorithm over the data without needing to provide any additional vertex information. This has some implications on how vertices are managed. For the Neptune Analytics query engine, a vertex implicitly exists if it either has an explicit label, a property, or an edge. Likewise, a vertex gets implicitly deleted if all its labels, properties, and edges get removed. Unlike Neptune Database, Neptune Analytics stores a label for a vertex only if one is explicitly provided by the user, and all labels of a vertex can be deleted.

This affects some common openCypher queries. An attempt to create a vertex that has neither a label nor properties or edges has no effect. That is, queries such as `CREATE (n)` or `CREATE (n {`~id`: "xyz"})` do not add any vertices to the graph. `CREATE (n {key:value})`, where `key` is different from `` `~id` ``, creates a vertex with the property `key`, and `CREATE (n)-[knows]->(m)` creates two vertices with the one shared edge.

`CREATE (n {key:value})`, where key is different from `` `~id` ``, creates a vertex with the property `key`, and a subsequent `MATCH (n)` will discover that vertex. A query such as `MATCH (n {key:value}) REMOVE n.key` will remove the only property for the (edge- and label-less) vertex, which implicitly deletes the vertex. A subsequent `MATCH (n)` query will not return that vertex. Likewise, `CREATE (n:Label)` adds a vertex with the label `Label` (and no other properties or edges). Now, `MATCH (n) REMOVE n:Label` deletes the only label of the vertex, which implicitly deletes the vertex.

Similarly, `CREATE (n)-[knows]->(m)` creates two nodes and one edge. `MATCH (n)` will discover those two vertices. Now, `MATCH (n)-[r:knows]->(m) DELETE r` will delete that edge, and implicitly deletes the two vertices. Those two vertices are no longer returned when running a `MATCH (n)` query.

Merge on empty vertices, `MERGE (n)` or `MERGE (n {`~id`: "xyz"})`, are not permitted and will throw an exception. `MERGE (n {key:value})` creates a vertex with property `key` if a matching vertex does not exist.

The following table illustrates the differences between Neptune Database and Neptune Analytics.

Query (run on empty graph)	Neptune Database	Neptune Analytics
`CREATE (n)`	Adds a vertex with label "vertex" to the graph. Each repeat request adds a new vertex to the graph.	No change to the graph, query returns without exception. Repeat requests similarly do not change the graph, and query returns without exception.
`CREATE (n {`~id`: "xyz"})`	Adds a vertex with id "xyz" and label "vertex" to the graph. Repeat request fails with exception.	No change to the graph, query returns without exception. Repeat requests similarly do not change the graph, and query returns without exception.
`CREATE (n {key:value})`	Adds a vertex with label "vertex" and property "key" to the graph.	Adds a vertex with property "key" to the graph. This vertex has no label.

`CREATE (n {key:value})`
`MATCH (n {key:value}) REMOVE n.key`
	The REMOVE query removes the "key" property on the vertex. The graph contains a vertex with label "vertex" but no property.
	`MATCH (n)` returns the vertex.
		The remove query removes the property on the vertex, and as a side effect the vertex gets deleted from the graph.
		`MATCH (n)` does not return the vertex.

`CREATE (n:Label {`~id`: "xyz", key:value})`	The REMOVE query errors out, the last label on a vertex cannot be deleted.	The REMOVE query removes the label. The graph contains a graph with id "xyz" and property "key".

Query (run on empty graph)	Neptune Database	Neptune Analytics
`` MATCH (n {`~id`: "xyz"}) REMOVE n:Label ``		
`CREATE (n)-[:knows]->(m)`	Adds two vertices with label "vertex" and an edge with label "knows" to the graph. `MATCH (n)` returns both those vertices.	Adds an edge between two new vertices to the graph. `MATCH (n)` returns both those vertices.

`CREATE (n)-[:knows]->(m)`
`MATCH (n)-[r:knows]->(n) DELETE r`
	Deletes the edge. The graph contains two isolated vertices.
	`MATCH (n)` returns both those vertices.
		Deletes the edge, and as a side effect the two vertices also get deleted from the graph. The graph is now empty.
		`MATCH (n)` does not return the two vertices.

`MERGE (n)`	Adds a vertex with label "vertex" if graph is empty. Matches all vertices in a non-empty graph.	Throws an exception.
`` MERGE (n {`~id`: "xyz"}) ``	Adds a vertex with label "vertex" and id "xyz" if one does not exist in the graph. Matches vertex with id "xyz".	Throws an exception.
`MERGE (n {key:value})`	Adds a vertex with label "vertex" and property "key" to the graph, if such a vertex does not already exists.	Adds a vertex with property "key" to the graph, if such a vertex does not already exist. This vertex has no label.

Query (run on empty graph)	Neptune Database	Neptune Analytics
`MERGE (n)-[knows]->(m)`	Adds two vertices with label "vertex" and an edge with label "knows" to the graph, if an edge with label knows does not exist. `MATCH (n)` returns both those vertices.	Adds an edge between two new vertices to the graph, if an edge with label "knows" does not exist. The two vertices have no label. `MATCH (n)` returns both those vertices.

Note

A workaround to implicit deletion of a vertex when all of its labels, properties, and edges get removed is to assign immutable labels to all vertices. This way, the deletion of all properties, edges, or mutable labels of a vertex will not lead to an implicit deletion. A vertex would not get deleted until explicitly deleted.
Likewise a workaround to no-op vertex create queries is to always create a vertex with a label or a property. To combine it with the previous point, always create a vertex with an immutable label. Extending this to bulk or batch loads, include all vertices in some vertex files and assign a property or an immutable label to all vertices.

Neptune Analytics openCypher specification compliance

Please refer to the Neptune Database documentation found here for openCypher specification compliance, with the exception that Neptune Analytics does not support custom edge IDs.

Vertex and edge IDs

Custom IDs for vertices

Neptune Analytics supports both querying and creating vertices with custom IDs. See openCypher custom IDs for more details.

Custom IDs for edges

Neptune Analytics does not support edge creation with custom edge IDs. Custom IDs are not permitted in CREATE or MERGE clauses. Edges are assigned IDs by Neptune , using a reserved prefix `neptune_reserved_`. Edges can be queried by the server assigned ids, just as in Neptune Database.

```
# Supported
MATCH (n)-[r:knows {`~id`: 'neptune_reserved_1_123456789'}]->(m)
RETURN r

# Unsupported
CREATE (n:Person {name: 'John'})-[:knows {`~id`: 'john-knows->jim'}]->(m:Person {name:
 'Jim'})

# Unsupported
MERGE (n)-[r:knows {`~id`: 'neptune_reserved_1_123456789'}]->(m)
RETURN r
```

Server assigned IDs are recycled. After an edge is deleted, a new edge created could get assigned the same ID.

Note

The edges could get assigned new IDs if the graph gets restructured and the older IDs would then become invalid. If the edges are reassigned IDs, older IDs would match no other edges. It is not recommended to store these IDs externally for long-term querying purposes.

IRIs and language-tagged literals

Neptune Analytics supports values hat are of type IRI or languag-tagged literal. See Handling RDF values for more information.

Transaction isolation levels in Neptune Analytics

Neptune Analytics has some differences with isolation level supported by Neptune Database.

Read-only query isolation in Neptune Analytics: Neptune Analytics evaluates read-only queries under snapshot isolation, just like Neptune Database.

Mutation query isolation in Neptune Analytics: Reads for mutation queries are normally executed under snapshot isolation, unlike Neptune Database. This is less stricter isolation than Neptune Database as the conditions in the query for proceeding to a write satisfied in a snapshot could have changed concurrently before the query commits.

For some specific steps, such as node/relationship deletion or conditional creation of new data using the MERGE step, reads also look at the concurrent writes, to avoid inconsistencies. Below are some examples where concurrent execution of queries one and two always lead to a consistent state. At most, one vertex gets created in example #1. The age is set to 10 or 11 in example #2, not both. And in example #3, either the vertex is fully deleted or the age is set to 11 without any deletion or removal of other properties.

```
# EXAMPLE 1
Query 1: MERGE (m:Person {ssn: '123456789'})
Query 2: MERGE (n:Person {ssn: '123456789'})
```

```
# EXAMPLE 2
Query 1: MATCH (n {ssn : '123456789'}) SET n.age=10
Query 2: MATCH (n {ssn : '123456789'}) SET n.age=11
```

```
# EXAMPLE 3
Query 1: MATCH (n {ssn : '123456789'}) DETACH DELETE n
Query 2: MATCH (n {ssn : '123456789'}) SET n.age = 11
```

Conflict detection: Different from Neptune Database, conflicts are evaluated more precisely over individual graph elements (properties or edges) rather than over a range of data. Queries one and two in example #4 would not conflict when run concurrently because they search and merge on different property values ('lname1' and 'lname2'). However, queries one and two in example #5 merge on different property-value sets, but they could still confict when run concurrently because they share a property-value (firstName: 'fname').

```
# EXAMPLE 4
Query 1: MERGE (n {lastName: 'lname1'})
Query 2: MERGE (n {lastName: 'lname2'})
```

```
# EXAMPLE 5
Query 1: MERGE (n {firstName: 'fname', lastName: 'lname1'})
Query 2: MERGE (n {firstName: 'fname', lastName: 'lname2'})
```

Neptune Analytics algorithms

Graph algorithms are powerful tools for gaining insights into data. Neptune Analytics provides a set of optimized in-database implementations of common graph algorithms that are exposed as openCypher procedures. These algorithms analyze inherent aspects of the underlying graph structure, such as connectedness (path finding), relative importance (centrality), and community membership (community detection).

Neptune Analytics natively supports over 25 optimized graph algorithms and variants in the 5 most popular categories that help customers extract insights from their graphs, which are listed in the following table.

Category	Action	Algorithms	Common Uses
Pathfinding	Find the existence, quality, or availability of a path between nodes.	• Breadth-First Search • Single-Source Shortest Path • Top-K Source Shortest Path • Source-Target Shortest Path • EgoNet	• Logistics optimization • Social network recommendations • Route optimization
Centrality	Determines the absolute or relative importance of a node in the graph.	• Degree • PageRank • Closeness Centrality	• Fraud ring/Collusion detection • Social network influencer identification • Supply chain risk analysis
Similarity	Compare the similarities between different graph structures.	• Common Neighbors • Total Neighbors	• Biological structural analysis • Social network cluster comparison

Category	Action	Algorithms	Common Uses
		• Jaccard Similarity • Overlap Similarity	• Link prediction
Clustering and Community Detection	Identify meaningful groups or clusters within graph structures.	• Weakly Connected Components (WCC) • Strongly Connected Components (SCC) • Label Propagation	• Social network clusters • Fraud ring identification • Householding • Biological interaction
Vector Similarity Search	Identify approximate nearest neighbor (ANN) nodes by comparing vector embeddings using the Hierarchical Navigable Small World (HNSW) algorithm.	• Distance • Top-K	• RAG applications • Knowledge graph backed chat bots • Approximate nearest neighbors

Many of these algorithms require interacting with most or all the nodes and edges in a graph, often in an iterative fashion. As a result, they are too computationally expensive to process using normal analytic technologies. Neptune Analytics has built highly optimized implementations that allow them to run over graphs of any size.

Algorithms in Neptune Analytics are integrated naturally into openCypher through the CALL clause, as illustrated below. This lets you combine algorithms naturally with openCypher clauses, functions, and semantics to build very complex queries. For example, you could look for the top 10 most important airports in the US-AK region like this:

```
MATCH (n:airport {region: 'US-AK'})
CALL neptune.algo.pageRank(n, {edgeLabels: ['route'], numOfIterations: 10})
YIELD rank
RETURN n.code, rank
```

```
ORDER BY rank DESC LIMIT 10
```

You can run algorithms in the SDKs using the `ExecuteOpenCypherQuery` operation or in boto3 and the AWS CLI using the `execute-query` command. If you don't want to use the SDK or CLI, you can use you can use [awscurl](#) to sign your Neptune Analytics requests using [signed using Signature Version 4 (Sig4)](#). For example, you can run a simple breadth-first search like this:

```
awscurl -X POST -H "Content-Type: application/x-www-form-urlencoded" \
  https://(graphIdentifier).(region).neptune-graph.amazonaws.com/opencypher \
  --service neptune-graph \
  --region (region) \
  -d "query=CALL neptune.algo.bfs([\"101\", \"102\"], {edgeLabels: [\"route\"]})"
```

You could run the same query using the AWS CLI like this:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string 'CALL neptune.algo.bfs(["101", "102"], {edgeLabels: ["route"]})' \
  --language open_cypher \
  /tmp/out.txt
```

Algorithms having signatures with different kinds of input in Neptune Analytics are exposed as separate algorithms. Unless otherwise indicated, the examples here are using the [Air Routes dataset](#).

Neptune Analytics currently supports five main categories of algorithm:

- [Path finding algorithms](#) – These find the existence, quality, or availability of a path or paths between two or more nodes in the graph. A path in this sense is a set of nodes and connecting edges.

  By efficiently determining the optimal route between two nodes, path-finding algorithms enable you to model real-world systems like roads or social networks as interconnected nodes and edges. Finding the shortest paths between various points is crucial in applications like route planning for GPS systems, logistics optimization, and even in solving complex problems in fields like biology or engineering.
- [Centrality algorithms](#) – These are used to determine the absolute or relative importance or influence of a node or nodes in the graph.

By identifying the most influential or important nodes within a network, centrality algorithms can provide insights about key players or critical points of interaction. This is valuable in social network analysis, where it helps pinpoint influential individuals, and in transportation networks, where it aids in identifying crucial hubs for efficient routing and resource allocation.

- Similarity algorithms – Graph similarity algorithms allow you to compare and analyze the similarities and dissimilarities between different graph structures, which can provide insight into relationships, patterns, and commonalities across diverse datasets. This is invaluable in various fields, such as biology, for comparing molecular structures, such as social networks, for identifying similar communities, and such as recommendation systems, for suggesting similar items based on user preferences.
- Clustering or community-detection algorithms – Community-detection algorithms can identify meaningful groups or clusters of nodes in a network, revealing hidden patterns and structures that can provide insights into the organization and dynamics of complex systems. This is valuable in social network analysis, and in biology, for identifying functional modules in protein-protein interaction networks, and more generally for understanding information flow and influence propagation in many different domains.
- Vector Similarity Search – Vector similarity algorithms work by using vector based representations of data, a.k.a. embeddings, to answer questions about the data's context and its similarity and connection to other data. This is valuable in applications such as Retrieval Augmented Generation (RAG) applications, knowledge graph backed chatbots, and recommendation engines.

Path-finding algorithms in Neptune Analytics

Path-finding algorithms are a category of graph algorithms that focus on finding a path, a connected set of nodes and edges, between two or more sets of nodes within a graph. They are often used to find available or optimized paths based on the existence, quantity, or quality of the paths and the values of properties along those paths.

By efficiently determining the best route between two nodes, path-finding algorithms enable you to model real-world systems like roads or social networks as interconnected nodes and edges. Finding the shortest paths between various points is crucial in applications like route planning for GPS systems, logistics optimization, and even in solving complex problems in fields like biology or engineering.

Breadth-first search (BFS) path finding algorithms

Breadth-first search (BFS) path-finding algorithms search for nodes in breadth-first order, starting from a single vertex. They can also, in the multi-source case, start from more than one vertex.

They can systematically explore and evaluates all neighboring nodes from a starting point before moving on to the neighbors of those nodes, which ensures that the algorithm searches the shallowest levels of the graph first.

Breadth-first-search is used in computer networking to find the shortest path between two devices, and in social networks to understand how information spreads through connections, and in games to explore possible moves and strategies.

Time complexity – The time complexity of breadth-first search algorithms is `O(|V|+|E|)`, where `|V|` is the number of vertices in the graph and `|E|` is the number of edges in the graph.

A breadth-first algorithm can be invoked as a *standalone* operation whose inputs are explicitly defined, or as a *query-algorithm integration* which takes as its input the output of an immediately preceding `MATCH` clause.

Neptune Analytics supports these BFS algorithms:

- `.bfs` – This standard breadth-first search algorithm starts from the source vertex of the graph and returns a column of visited vertices.
- `.bfs.parents` – This variant of BFS starts from a source vertex or vertices and finds the parent of each vertex during the search. It returns a key column of the vertices and a value column of the parents of the key vertices.

- `.bfs.levels` – This variant of BFS starts from a source vertex or vertices and finds the levels of each vertex during the search. It returns a key column of the vertices and a value column of integers that are the level values of the key vertices.

  Note that the level of a source vertex is 0.

Standard breadth-first search (BFS) algorithm

Standard breadth-first search (BFS) is an algorithm for finding nodes from a starting node or nodes in a graph in breadth-first order.

It returns the source node or nodes that it started from, and all of the nodes visited by each search.

> **Note**
>
> Because every source node passed in leads to its own execution of the algorithm, your queries should limit the number of source nodes as much as possible.

`.bfs` syntax

```
CALL neptune.algo.bfs(
  [source-node list (required)],
  {
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    maxDepth: maximum number of hops to traverse from a source node (optional),
    traversalDirection: traversal direction (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD the outputs to generate (source and/or node)
RETURN the outputs to return
```

`.bfs` inputs

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none.*

  The source-node list contains the node or nodes used as the starting locations for the algorithm.

  - Each starting node triggers its own execution of the algorithm.
  - If the source-node list is empty then the query result is also empty.
  - If the algorithm is called following a `MATCH` clause (this is known as query-algorithm integration), the output of the `MATCH` clause is used as the source-node list for the algorithm.

- **a configuration object that contains:**
  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.
  - **vertexLabel** *(optional)* – *type:* `string`; *example:* `"airport"`; *default:* no node filtering.

    If you provide a node label to filter on then only nodes matching that label will be traversed. This does not, however, filter out any nodes in the source node list.
  - **maxDepth** *(optional)* – *type:* positive integer or 0 or -1; *default:* -1.

    The maximum number of hops to traverse from a source node. If set at `-1` then there's no maximum depth limit. If set to `0`, only the nodes in the source node list are returned.
  - **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

    The direction of edge to follow. Must be one of: `"inbound"`, `"oubound"`, or `"both"`.
  - **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

    Controls the number of concurrent threads used to run the algorithm.

    If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

`.bfs` outputs

The `.bfs` algorithm returns:

- **source** – *type:* `Node[]`.

  The source nodes.
- **node** – *type:* `Node[]`.

  The nodes that the algorithm traversed from each source node.

`.bfs` query examples

This is a standalone example, where the query provides an explicit source node list.

```
CALL neptune.algo.bfs(
  ["101", "102"],
  {
    edgeLabels: ["route"],
    vertexLabel: "airport",
    maxDepth: 11,
    traversalDirection: "both",
    concurrency: 2
  }
)
YIELD node
```

You can run that query using the `execute-query` operation in the AWS CLI like this:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string 'CALL neptune.algo.bfs(["101", "102"],
      {edgeLabels: ["route"], vertexLabel: "airport", maxDepth: 11,
      traversalDirection: "both", concurrency: 2})' \
  --language open_cypher \
  /tmp/out.txt
```

A query like this one would return an empty result because the source list is empty:

```
CALL neptune.algo.bfs([], {edgeLabels: ["route"]})
```

By default, both the source nodes (`"source"` output) and the visited nodes (`"node"` output) are returned. You can use `YIELD` to specify which of those outputs you would like to see. For example, to see only the `"node"` outputs:

```
CALL neptune.algo.bfs(["101"], {edgeLabels: ["route"]}) YIELD node
```

The examples below are query integration examples, where `.bfs` follows a `MATCH` clause and uses the output of the `MATCH` clause as its source node list:

```
MATCH (n) WITH n LIMIT 5
CALL neptune.algo.bfs(n, {edgeLabels: ["route"]})
```

```
YIELD node
RETURN node
```

The MATCH clause can also explitly specify a starting node list using the `id()` function, like this:

```
MATCH (n) where id(n)="101"
CALL neptune.algo.bfs(n, {edgeLabels: ["route"]})
YIELD node
RETURN node
```

Also:

```
MATCH (n) where id(n) IN ["101", "102"]
CALL neptune.algo.bfs(n, {edgeLabels: ["route"]})
YIELD node
RETURN COUNT(node)
```

⚠ **Warning**

It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.bfs` output

Here is an example of the output returned by .bfs when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.bfs(['101'], {maxDepth: 1}) yield source, node
 return source, node limit 2" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [{
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.681099891662599,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "lon": 100.74700164794901,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "1483",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 39.490000000000002,
          "elev": 4557,
          "longest": 9186,
          "city": "Ordos",
          "type": "airport",
          "region": "CN-15",
          "desc": "Ordos Ejin Horo Airport",
          "code": "DSN",
          "lon": 109.861388889,
          "country": "CN",
          "icao": "ZBDS",
          "runways": 1
        }
      }
    }, {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.681099891662599,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "lon": 100.74700164794901,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "103",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 55.972599029541001,
          "elev": 622,
          "longest": 12139,
          "city": "Moscow",
          "type": "airport",
          "region": "RU-MOS",
          "desc": "Moscow, Sheremetyevo International Airport",
          "code": "SVO",
          "lon": 37.414600372314503,
          "country": "RU",
          "icao": "UUEE",
          "runways": 2
        }
      }
    }]
}
```

Parents breadth-first search (BFS) algorithm

The `parents` variant of breadth-first search is an algorithm for finding nodes from a starting node or vertices in breadth-first order and then performing a breadth-first search for the parent of each node.

It returns a key column of vertices, and a value column of the vertices that are the parents of the key vertices. The parent of a source node is itself.

> **Note**
>
> Because every source node passed in initiates its own execution of the algorithm, your queries should limit the number of source nodes as much as possible.

`.bfs.parents` syntax

```
CALL neptune.algo.bfs.parents(
  [source-node list (required)],
  {
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    maxDepth: maximum number of hops to traverse from a source node (optional),
    traversalDirection: traversal direction (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD the outputs to generate (source and/or node and/or parent)
RETURN the outputs to return
```

`.bfs.parents` inputs

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The source-node list contains the node or nodes used as the starting locations for the algorithm.

  - Each starting node triggers its own execution of the algorithm.
  - If the source-node list is empty then the query result is also empty.
  - If the algorithm is called following a `MATCH` clause (this is known as query-algorithm integration), the output of the `MATCH` clause is used as the source-node list for the algorithm.

- **a configuration object that contains:**
  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.
  - **vertexLabel** *(optional)* – *type:* `string`; *example:* `"airport"`; *default:* no node filtering.

    If you provide a node label to filter on then only vertices matching that label will be traversed. This does not, however, filter out any nodes in the source node list.
  - **maxDepth** *(optional)* – *type:* positive inteeger or 0 or -1; *default:* -1.

    The maximum number of hops to traverse from a source node. If set at `-1` then there's no maximum depth limit. If set to `0`, only the vertices in the source node list are returned.
  - **traversalDirection** *(optional)* – *type:* `string`; *default:* `outbound`.

    The direction of edge to follow. Must be one of: `inbound`, oubound, or `both`.
  - **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

    Controls the number of concurrent threads used to run the algorithm.

    If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

`.bfs.parents` outputs

The `.bfs.parents` algorithm returns:

- **source** – *type:* `Node[]`.

  The source nodes.
- **node** – *type:* `Node[]`.

  The vertices that the algorithm traversed from each source node.
- **parent** – *type:* `Node[]`.

  The parents of those traversed nodes.

`.bfs.parents` query examples

Thus is a standalone examples, where the source node list is explicitly provided in the query:

```
CALL neptune.algo.bfs.parents(
  ["105", "113"],
  {
    edgeLabels: ["route"],
    vertexLabel: "airport",
    maxDepth: 2,
    traversalDirection: "both",
    concurrency: 2
  }
)
YIELD node, parent
```

A query like this one would return an empty result because the source list is empty:

```
CALL neptune.algo.bfs.parents([], {edgeLabels: ["route"]})
```

This is a query integration example, where `.bfs.parents` follows a `MATCH` clause that provides the source node list for `.bfs.parents`:

```
Match (n) with n LIMIT 5
CALL neptune.algo.bfs.parents(n, {edgeLabels: ["route"]})
YIELD node
RETURN n, node
```

This query is an example of aliasing the algorithm output:

```
MATCH (n {code: "AUS"})
CALL neptune.algo.bfs.parents(n, { edgeLabels: ["route"], maxDepth: 2})
YIELD node AS ReachedNode
RETURN ReachedNode
```

This query searches for routes to BFS from BKK, returning the starting node (BKK), 5 visited vertices, and their parents:

```
MATCH (n) where n.code CONTAINS "BKK"
CALL neptune.algo.bfs.parents(n, {edgeLabels: ["route"]})
YIELD node, parent
```

```
RETURN n, node, parent
LIMIT 5
```

> ⚠ **Warning**
>
> It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.bfs.parents` output

Here is an example of the output returned by .bfs.parents when run against the [sample air-routes dataset [nodes]](), and [sample air-routes dataset [edges]](), when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.bfs.parents(['101'], {maxDepth: 1})
                  YIELD source, node, parent
                  RETURN source, node, parent
                  LIMIT 2"
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "lon": 100.747001647949,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "1483",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 39.49,
          "elev": 4557,
          "longest": 9186,
          "city": "Ordos",
          "type": "airport",
          "region": "CN-15",
          "desc": "Ordos Ejin Horo Airport",
          "code": "DSN",
          "lon": 109.861388889,
          "country": "CN",
          "icao": "ZBDS",
          "runways": 1
        }
      },
      "parent": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "lon": 100.747001647949,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      }
```

```
    },
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "lon": 100.747001647949,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "103",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 55.972599029541,
          "elev": 622,
          "longest": 12139,
          "city": "Moscow",
          "type": "airport",
          "region": "RU-MOS",
          "desc": "Moscow, Sheremetyevo International Airport",
          "code": "SVO",
          "lon": 37.4146003723145,
          "country": "RU",
          "icao": "UUEE",
          "runways": 2
        }
      },
      "parent": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "lon": 100.747001647949,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      }
    }
  ]
}
```

Levels breadth-first search (BFS) algorithm

The `levels` variant of breadth-first search is an algorithm for searching nodes from a starting node or nodes in breadth-first order. From there it performs a breadth-first search and records the hop level from the starting node of each node that it finds.

It returns a key column of nodes, and a value column containing the level values of those key nodes.

The level of a source node is 0. Note that because every source node passed into breadth-first search `levels` initiates its own execution of the algorithm, your queries should filter to a subset of the graph before executing BFS `levels` whenever possible.

`.bfs.levels` syntax

```
CALL neptune.algo.bfs.levels(
  [source-node list (required)],
  {
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    maxDepth: maximum number of hops to traverse from a source node (optional),
    traversalDirection: traversal direction (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD the outputs to generate (source and/or node)
RETURN the outputs to return
```

`.bfs.levels` inputs

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The source-node list contains the node or nodes used as the starting locations for the algorithm.

  - Each starting node triggers its own execution of the algorithm.
  - If the source-node list is empty then the query result is also empty.
  - If the algorithm is called following a `MATCH` clause (this is known as query-algorithm integration), the output of the `MATCH` clause is used as the source-node list for the algorithm.
- **a configuration object that contains:**
  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **vertexLabel** *(optional)* – *type:* `string`; *example:* `"airport"`; *default:* no node filtering.

  If you provide a node label to filter on then only nodes matching that label will be traversed. This does not, however, filter out any nodes in the source node list.

- **maxDepth** *(optional)* – *type:* positive integer or 0 or -1; *default:* -1.

  The maximum number of hops to traverse from a source node. If set at `-1` then there's no maximum depth limit. If set to `0`, only the nodes in the source node list are returned.

- **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

  The direction of edge to follow. Must be one of: `"inbound"`, `"oubound"`, or `"both"`.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

`.bfs.levels` outputs

The `.bfs.levels` algorithm returns:

- **source** – *type:* `Node[]`.

  The source nodes.

- **node** – *type:* `Node[]`.

  The nodes that the algorithm traversed from each source node.

- **level** – *type:* `integer[]`.

  The hop levels of those traversed nodes.

`.bfs.levels` standalone query examples

The examples below are standalone examples, where the query provides an explicit source node list.

A query like this one would return an empty result because the source list is empty:

```
CALL neptune.algo.bfs.levels(
  ["101", "102"],
  {
    edgeLabels: ["route"],
    vertexLabel: "airport",
    maxDepth: 6,
    traversalDirection: "both",
    concurrency: 2
  }
)
YIELD node
```

You can run the algorithm using the `execute-query` operation in the AWS CLI like this:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string 'CALL neptune.algo.bfs.levels(["101", "102"], {edgeLabels:
 ["route"]})' \
  --language open_cypher \
  /tmp/out.txt
```

By default, all the outputs are generated. You can use `YIELD` to specify which of those outputs to generate. For example, to generate only the `"node"` and `level` outputs:

```
CALL neptune.algo.bfs.levels(["101"], {edgeLabels: ["route"]}) YIELD node, level
```

`.bfs.levels` query integration examples

The examples below are query integration examples, where `.bfs.levels` follows a `MATCH` clause and uses the output of the `MATCH` clause as its source node list:

```
MATCH (n) WITH n LIMIT 5
CALL neptune.algo.bfs.levels(n, {edgeLabels: ["route"]})
YIELD node, level
```

```
RETURN n, node, level
```

This query illustrates various ways to constrain the input and output:

```
MATCH (n) where id(n)="101"
CALL neptune.algo.bfs.levels(n, { edgeLabel: "route", maxDepth: 2})
YIELD node, level WHERE node.city CONTAINS "New"
RETURN n.city, node.city, level
```

Warning

It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.bfs.levels` output

Here is an example of the output returned by .bfs.levels when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.bfs.levels(['101'], {maxDepth: 1}) yield source,
 node, level return source, node, level limit 2" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "lon": 100.747001647949,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "1483",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 39.49,
          "elev": 4557,
          "longest": 9186,
          "city": "Ordos",
          "type": "airport",
          "region": "CN-15",
          "desc": "Ordos Ejin Horo Airport",
          "code": "DSN",
          "lon": 109.861388889,
          "country": "CN",
          "icao": "ZBDS",
          "runways": 1
        }
      },
      "level": 1
    },
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "lon": 100.747001647949,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "103",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 55.972599029541,
          "elev": 622,
          "longest": 12139,
          "city": "Moscow",
          "type": "airport",
          "region": "RU-MOS",
          "desc": "Moscow, Sheremetyevo International Airport",
          "code": "SVO",
          "lon": 37.4146003723145,
          "country": "RU",
          "icao": "UUEE",
          "runways": 2
        }
      },
      "level": 1
    }
  ]
}
```

Single-source shortest-path algorithms

A single-source-shortest-path algorithm finds the shortest paths (or the distance of the shortest paths) between a given vertex and all reachable vertices in the graph (including itself).

By determining the most efficient routes from a single starting node to all other nodes in the graph, single-source-shortest-path can be used calculate the shortest distances or lowest cost required to reach each destination. This is applicable in GPS systems to find the fastest routes between a starting point and differeent destinations, and in logistics to optimize delivery routes, and in transportation planning for efficient navigation through road networks.

Neptune Analytics supports the following single-source-shortest-path (SSSP) algorithms:

- `.sssp.bellmanFord` – Computes the shortest path distances from a source vertex to all other vertices in the graph using the Bellman-Ford algorithm. Positive edge weights must be provided using the `edgeWeightProperty`, and the traversal direction must not be set to `both`.
- `.sssp.bellmanFord.parents` – Identifies the parent vertices along the shortest paths from the source vertex to all other vertices in the graph using the Bellman-Ford algorithm. Positive edge weights must be provided using the `edgeWeightProperty`, and the traversal direction must not be set to `both`.
- `.sssp.bellmanFord.path` – Finds the shortest path between a given source vertex and a target vertex in the graph using the Bellman-Ford algorithm. To compute all shortest paths from a given source vertex, the regular SSSP algorithm can be used. Positive edge weights must be provided using the `edgeWeightProperty`, and the traversal direction must not be set to `both`.
- `.sssp.deltaStepping` – Computes the shortest path distances from a source vertex to all other vertices in the graph using a delta-stepping algorithm. Positive edge weights must be provided using the `edgeWeightProperty`, and the traversal direction must not be set to `both`.
- `.sssp.deltaStepping.parents` – Identifies the parent vertices along the shortest paths from the source vertex to all other vertices in the graph using a delta-stepping algorithm. Positive edge weights must be provided using the `edgeWeightProperty`, and the traversal direction must not be set to `both`.
- `.sssp.deltaStepping.path` – Finds the shortest path between a given source vertex and a target vertex in the graph using the delta-stepping algorithm. To compute all shortest paths from a given source vertex, the regular SSSP algorithm can be used. Positive edge weights must be provided using the `edgeWeightProperty`, and the traversal direction must not be set to `both`.
- `.topksssp` – The TopK hop-limited single source shortest path algorithm finds the single-source weighted shortest paths starting from a source vertex to all its `maxDepth` neighbors. The distance or cost from the source vertex to each target vertex is accumulated on the edge weights of the path. The topK distances of the paths are sorted in descending or ascending order.

  The algorithm can be run unweighted as well as weighted. When you run it unweighted, it's equivalent to `.bfs.levels`.

Bellman-Ford single source shortest path (SSSP) algorithm

The `.sssp.bellmanFord` algorithm computes the shortest path distances from a single source vertex to all other vertices in the graph using the Bellman-Ford algorithm.

Neptune Analytics implements the algorithm such that:

- Positive edge weights must be provided using the `edgeWeightProperty` field
- Negative edge weights are not supported.
- The traversal direction cannot be set to `both`.

`.sssp.bellmanFord` syntax

```
CALL neptune.algo.sssp.bellmanFord(
  [source-node list (required)],
  {
    edgeWeightProperty: edge weight predicate for traversal (required)
    edgeWeightType: numeric type of the edge weight property (required)
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    traversalDirection: traversal direction (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD the outputs to generate (source and/or node)
RETURN the outputs to return
```

`.sssp.bellmanFord` inputs

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes to use as the starting location(s) for the algorithm.

  - Each starting node triggers its own execution of the algorithm.
  - If the source-node list is empty then the query result is also empty.
  - If the algorithm is called following a `MATCH` clause (this is known as query-algorithm integration), the output of the `MATCH` clause is used as the source-node list for the algorithm.
- **a configuration object that contains:**
  - **edgeWeightProperty** *(required)* – *type:* `string`; *default: none*.

The edge weight predicate for traversal.

- **edgeWeightType** *(required)* – *type:* `string`; *valid values:* `"int"`, `"long"`, `"float"`, `"double"`.

  The numeric data type of the values in the property specified by `edgeWeightProperty`.

- **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

  To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **vertexLabel** *(optional)* – *type:* `string`; *example:* `"airport"`; *default:* no node filtering.

  A node label for node filtering. If a node label is provided, vertices matching the label are the only vertices that are included, including vertices in the input list.

- **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

  The direction of edge to follow. Must be one of: `"inbound"` or `"oubound"`.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `.sssp.bellmanFord` algorithm

For every node that can be reached from the specified source list, the algorithm returns:

- **source** – The source node.
- **node** – A node found traversing from the source.
- **distance** – The distance between the source node and the found node.

`.sssp.bellmanFord` query examples

This is a standalone query, where a source node (or nodes) is explicitly provided:

```
CALL neptune.algo.sssp.bellmanFord(
  ["101"],
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    edgeWeightType: "int"
  }
)
```

This is a query integration example, where `.sssp.bellmanFord` follows a `MATCH` clause and uses the output of the `MATCH` clause as its source node list:

```
MATCH (source:airport {code: 'ANC'})
CALL neptune.algo.sssp.bellmanFord(
  source,
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    edgeWeightType: "int",
    vertexLabel: "airport",
    traversalDirection: "outbound",
    concurrency: 1
  }
)
YIELD node, parent, distance
RETURN source, node, parent, distance
```

⚠ **Warning**

It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.sssp.bellmanFord` output

Here is an example of the output returned by .sssp.bellmanFord when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.sssp.bellmanFord(['101'],
       {edgeWeightProperty: 'dist', edgeWeightType: 'int'})
     yield source, node, distance
     return source, node, distance
     limit 2" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "prscore": 0.002498496090993285,
          "degree": 308,
          "lon": 100.747001647949,
          "wccid": 2357352929951779,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "2709",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 65.4809036254883,
          "elev": 49,
          "longest": 8711,
          "city": "Nadym",
          "type": "airport",
          "region": "RU-YAN",
          "desc": "Nadym Airport",
          "code": "NYM",
          "prscore": 0.00016044313088059425,
          "degree": 18,
          "lon": 72.6988983154297,
          "wccid": 2357352929951779,
          "country": "RU",
          "icao": "USMM",
          "runways": 1
        }
      },
      "distance": 3812
    },
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "prscore": 0.002498496090993285,
          "degree": 308,
          "lon": 100.747001647949,
          "wccid": 2357352929951779,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "2667",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 56.8567008972168,
          "elev": 2188,
          "longest": 6562,
          "city": "Ust-Kut",
          "type": "airport",
          "region": "RU-IRK",
          "desc": "Ust-Kut Airport",
          "code": "UKX",
          "prscore": 0.000058275499999999997,
          "degree": 4,
          "lon": 105.730003356934,
          "wccid": 2357352929951779,
          "country": "RU",
          "icao": "UITT",
          "runways": 1
        }
      },
      "distance": 2993
    }
  ]
}
```

Bellman-Ford single source shortest path (SSSP) parents algorithm

The `.sssp.bellmanFord.parents` algorithm uses the Bellman-Ford algorithm to find the parent nodes along with the shortest path distances from the source node to all other nodes in the graph.

Neptune Analytics implements the algorithm such that:

- Positive edge weights must be provided using the `edgeWeightProperty` field
- Negative edge weights are not supported.
- The traversal direction cannot be set to `both`.

`.sssp.bellmanFord.parents` syntax

```
CALL neptune.algo.sssp.bellmanFord.parents(
  [source-node list (required)],
  {
    edgeWeightProperty: edge weight predicate for traversal (required)
    edgeWeightType: numeric type of the edge weight property (required)
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    traversalDirection: traversal direction (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD the outputs to generate (source and/or node)
RETURN the outputs to return
```

`.sssp.bellmanFord.parents` inputs

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes to use as the starting location(s) for the algorithm.

  - Each starting node triggers its own execution of the algorithm.
  - If the source-node list is empty then the query result is also empty.
  - If the algorithm is called following a `MATCH` clause (this is known as query-algorithm integration), the output of the `MATCH` clause is used as the source-node list for the algorithm.

- **a configuration object that contains:**
  - **edgeWeightProperty** *(required)* – *type:* `string`; *example:* `"distnce"`; *default: none*.

    The edge weight predicate for traversal.
  - **edgeWeightType** *(required)* – *type:* `string`; *valid values:* `"int"`, `"long"`, `"float"`, `"double"`.

    The numeric data type of the values in the property specified by `edgeWeightProperty`.
  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.
  - **vertexLabel** *(optional)* – *type:* `string`; *example:* `"airport"`; *default:* no node filtering.

    A node label for node filtering. If a node label is provided, vertices matching the label are the only vertices that are included, including vertices in the input list.
  - **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

    The direction of edge to follow. Must be one of: `"inbound"` or `"oubound"`.
  - **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

    Controls the number of concurrent threads used to run the algorithm.

    If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `.sssp.bellmanFord.parents` algorithm

For every node that can be reached from the specified source list, the algorithm returns:

- **source** – The source node.
- **node** – A node found traversing from the source.
- **distance** – The distance between the source node and the found node.
- **parent** – The parent of the found node. Note that the parent of the source vertex is itself.

`.sssp.bellmanFord.parents` query examples

This is a standalone query, where a source node (or nodes) is explicitly provided:

```
CALL neptune.algo.sssp.bellmanFord.parents(
  ["101"],
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    edgeWeightType: "int"
  }
)
```

This is a query integration example, where where `.sssp.bellmanFord.parents` follows a `MATCH` clause and uses the output of the `MATCH` clause as its source node list:

```
MATCH (source:airport {code: 'ANC'})
CALL neptune.algo.sssp.bellmanFord.parents(
  source,
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    edgeWeightType: "int",
    vertexLabel: "airport",
    traversalDirection: "outbound",
    concurrency: 1
  }
)
YIELD node, parent, distance
RETURN source, node, parent, distance
```

> **Warning**
>
> It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample .sssp.bellmanFord.parents output

Here is an example of the output returned by .sssp.bellmanFord.parents when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.sssp.bellmanFord.parents(['101'],
       {edgeWeightProperty: 'dist', edgeWeightType: 'int'})
     yield source, node, parent
     return source, node, parent
     limit 2" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "prscore": 0.002498496090993285,
          "degree": 308,
          "lon": 100.747001647949,
          "wccid": 2357352929951779,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "2709",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 65.4809036254883,
          "elev": 49,
          "longest": 8711,
          "city": "Nadym",
          "type": "airport",
          "region": "RU-YAN",
          "desc": "Nadym Airport",
          "code": "NYM",
          "prscore": 0.00016044313088059425,
          "degree": 18,
          "lon": 72.6988983154297,
          "wccid": 2357352929951779,
          "country": "RU",
          "icao": "USMM",
          "runways": 1
        }
      },
      "parent": {
        "~id": "810",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 55.0125999450684,
          "elev": 365,
          "longest": 11818,
          "city": "Novosibirsk",
          "type": "airport",
          "region": "RU-NVS",
          "desc": "Tolmachevo Airport",
          "code": "OVB",
          "prscore": 0.0012910010991618038,
          "degree": 162,
          "lon": 82.6507034301758,
          "wccid": 2357352929951779,
          "country": "RU",
          "icao": "UNNT",
          "runways": 2
        }
      }
    },
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "prscore": 0.002498496090993285,
          "degree": 308,
          "lon": 100.747001647949,
          "wccid": 2357352929951779,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "2667",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 56.8567008972168,
          "elev": 2188,
          "longest": 6562,
          "city": "Ust-Kut",
          "type": "airport",
          "region": "RU-IRK",
          "desc": "Ust-Kut Airport",
          "code": "UKX",
          "prscore": 0.000058275499999999997,
          "degree": 4,
          "lon": 105.730003356934,
          "wccid": 2357352929951779,
          "country": "RU",
          "icao": "UITT",
          "runways": 1
        }
```

```
   },
   "parent": {
    "~id": "1038",
    "~entityType": "node",
    "~labels": ["airport"],
    "~properties": {
     "lat": 52.2680015563965,
     "elev": 1675,
     "longest": 10384,
     "city": "Irkutsk",
     "type": "airport",
     "region": "RU-IRK",
     "desc": "Irkutsk Airport",
     "code": "IKT",
     "prscore": 0.0008466026629321277,
     "degree": 84,
     "lon": 104.388999938965,
     "wccid": 2357352929951779,
     "country": "RU",
     "icao": "UIII",
     "runways": 1
    }
   }
  }
 ]
}
```

Bellman-Ford single source single target shortest path algorithm

The `.sssp.bellmanFord.path` algorithm uses the Bellman-Ford algorithm to find the shortest path along with the shortest path distances from a source node to a target node in the graph. If there are multiple shortest paths between the source node and the target node, only one will be returned. The algorithm can run only weighted, with `edgeWeightProperty` provided.

Neptune Analytics implements the algorithm such that:

- Positive edge weights must be provided using the `edgeWeightProperty` field.
- Negative edge weights are not supported.
- The traversal direction cannot be set to `both`.

`.sssp.bellmanFord.path` syntax

```
CALL neptune.algo.sssp.bellmanFord.path(
  [source node(s) (required)], [target node(s) (required)]
  {
    edgeWeightProperty: edge weight predicate for traversal (required)
    edgeWeightType: numeric type of the edge weight property (required)
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    traversalDirection: traversal direction (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD the outputs to generate (source and/or node)
RETURN the outputs to return
```

`.sssp.bellmanFord.path` inputs

- **source node(s)** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes to use as the starting location(s) for the algorithm.

  - Each starting node triggers its own execution of the algorithm.
  - If the source node(s) is empty then the query result is also empty.
  - If the algorithm is called following a `MATCH` clause (this is known as query-algorithm integration), the output of the `MATCH` clause is used as the source node(s) for the algorithm.
- **target node(s)** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

The node or nodes to use as the ending location(s) for the algorithm.

- Each source-target node pair produces an output of the algorithm.
- If the algorithm is called following a `MATCH` clause (this is known as query-algorithm integration), the output of the `MATCH` clause is used as the target node(s) for the algorithm.

- **a configuration object that contains:**
  - **edgeWeightProperty** *(required)* – *type:* `string`; *default: none*.

    The edge weight predicate for traversal.
  - **edgeWeightType** *(required)* – *type:* `string`; *valid values:* `"int"`, `"long"`, `"float"`, `"double"`.

    The numeric data type of the values in the property specified by `edgeWeightProperty`.
  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.
  - **vertexLabel** *(optional)* – *type:* `string`; *example:* `"airport"`; *default:* no node filtering.

    A node label for node filtering. If a node label is provided, vertices matching the label are the only vertices that are included, including vertices in the input list.
  - **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

    The direction of edge to follow. Must be one of: `"inbound"` or `"oubound"`.
  - **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

    Controls the number of concurrent threads used to run the algorithm.

    If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `.sssp.bellmanFord.path` algorithm

For every pair of source and target nodes, the algorithm returns:

- **source** – The source vertex.
- **target** – The target vertex.
- **distance** – The total shortest path distance from source to target.
- **vertexPath** – A list of vertices in the path in visit order (including the source and the target).
- **allDistances** – A list of cumulative distances to the vertices in the traversal path (including the source and the target).
- **path** – An openCypher path object representing the shortest path between the source and the target. (A list of vertices from the source vertex to the target vertex, interleaved with the corresponding edges, representing the shortest path. Sequence of vertex id (source), edge id, vertex id, edge id, ..., vertex id (target)).
  - Starts and ends with a vertex, and has edges in between each vertex.
  - Includes the source and the target vertices.

`.sssp.bellmanFord.path` query examples

This is a standalone query, where a source node and target node are explicitly provided:

```
CALL neptune.algo.sssp.bellmanFord.path(
  "9", "37",
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    edgeWeightType: "int"
  }
)
```

This is a query integration example, where `.sssp.bellmanFord.path` follows a `MATCH` clause and uses the output of the `MATCH` clause as its source node(s) and target node(s):

```
MATCH (source:airport {code: 'FLL'})
MATCH (target:airport {code: 'HNL'})
CALL neptune.algo.sssp.bellmanFord.path(
  source, target,
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    edgeWeightType: "int",
    vertexLabel: "airport",
    traversalDirection: "outbound",
    concurrency: 1
  }
)
YIELD distance, vertexPath, allDistances, path
RETURN source, target, distance, vertexPath, allDistances, path
```

Warning

It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes are returned; and that every source-target node pair produces an output, which can result in a very large query output if a large number of nodes are returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.sssp.bellmanFord.path` output

Here is an example of the output returned by .sssp.bellmanFord.path when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "MATCH (n:airport {code: 'FLL'})
  MATCH (m:airport {code: 'HNL'})
  CALL neptune.algo.sssp.bellmanFord.path(n, m, {
  edgeWeightProperty: "dist",
  edgeWeightType: "int",
  edgeLabels: ["route"],
  vertexLabel: "airport",
  traversalDirection: "outbound",
  concurrency: 1
  })
  YIELD source, target, distance, vertexPath, allDistances, path
  RETURN source, target, distance, vertexPath, allDistances, path" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [{
      "source": {
        "~id": "9",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "region": "US-FL",
          "runways": 2,
          "country": "US",
          "city": "Fort Lauderdale",
          "type": "airport",
          "icao": "KFLL",
          "lon": -80.152702331542997,
          "code": "FLL",
          "lat": 26.0725994110107,
          "longest": 9000,
          "elev": 64,
          "desc": "Fort Lauderdale/Hollywood International Airport"
        }
      },
      "target": {
        "~id": "37",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "region": "US-HI",
          "runways": 4,
          "country": "US",
          "city": "Honolulu",
          "type": "airport",
          "icao": "PHNL",
          "lon": -157.92199707031199,
          "code": "HNL",
          "lat": 21.318700790405298,
          "longest": 12312,
          "elev": 13,
          "desc": "Honolulu International Airport"
        }
      },
      "distance": 4854,
      "vertexPath": [{
          "~id": "9",
          "~entityType": "node",
          "~labels": ["airport"],
          "~properties": {
            "region": "US-FL",
            "runways": 2,
            "country": "US",
            "city": "Fort Lauderdale",
            "type": "airport",
            "icao": "KFLL",
            "lon": -80.152702331542997,
            "code": "FLL",
            "lat": 26.0725994110107,
            "longest": 9000,
            "elev": 64,
            "desc": "Fort Lauderdale/Hollywood International Airport"
          }
        }, {
          "~id": "11",
          "~entityType": "node",
          "~labels": ["airport"],
          "~properties": {
            "region": "US-TX",
            "runways": 5,
            "country": "US",
            "city": "Houston",
            "type": "airport",
            "icao": "KIAH",
            "lon": -95.341400146484403,
            "code": "IAH",
            "lat": 29.984399795532202,
            "longest": 12001,
            "elev": 96,
            "desc": "George Bush Intercontinental"
          }
        }, {
          "~id": "37",
          "~entityType": "node",
          "~labels": ["airport"],
          "~properties": {
            "region": "US-HI",
            "runways": 4,
            "country": "US",
            "city": "Honolulu",
            "type": "airport",
            "icao": "PHNL",
            "lon": -157.92199707031199,
            "code": "HNL",
            "lat": 21.318700790405298,
            "longest": 12312,
            "elev": 13,
            "desc": "Honolulu International Airport"
          }
        }],
      "allDistances": [0, 964, 4854],
      "path": [{
          "~id": "9",
          "~entityType": "node",
          "~labels": ["airport"],
          "~properties": {
            "region": "US-FL",
            "runways": 2,
            "country": "US",
            "city": "Fort Lauderdale",
            "type": "airport",
            "icao": "KFLL",
            "lon": -80.152702331542997,
            "code": "FLL",
            "lat": 26.0725994110107,
            "longest": 9000,
            "elev": 64,
            "desc": "Fort Lauderdale/Hollywood International Airport"
          }
        }, {
          "~id": "neptune_reserved_1_1152921504607567884",
          "~entityType": "relationship",
          "~start": "9",
          "~end": "11",
          "~type": "route",
          "~properties": {
            "dist": 964
          }
        }, {
          "~id": "11",
          "~entityType": "node",
          "~labels": ["airport"],
          "~properties": {
            "region": "US-TX",
            "runways": 5,
            "country": "US",
            "city": "Houston",
            "type": "airport",
            "icao": "KIAH",
            "lon": -95.341400146484403,
            "code": "IAH",
            "lat": 29.984399795532202,
            "longest": 12001,
            "elev": 96,
            "desc": "George Bush Intercontinental"
          }
        }, {
          "~id": "neptune_reserved_1_1152921508902600717",
          "~entityType": "relationship",
          "~start": "11",
          "~end": "37",
          "~type": "route",
          "~properties": {
            "dist": 3890
          }
        }, {
          "~id": "37",
          "~entityType": "node",
          "~labels": ["airport"],
          "~properties": {
            "region": "US-HI",
            "runways": 4,
            "country": "US",
            "city": "Honolulu",
            "type": "airport",
            "icao": "PHNL",
            "lon": -157.92199707031199,
            "code": "HNL",
            "lat": 21.318700790405298,
            "longest": 12312,
            "elev": 13,
            "desc": "Honolulu International Airport"
          }
        }]
    }]
}
```

Delta-stepping single source shortest path (SSSP) algorithm

The `.sssp.deltaStepping` algorithm computes the shortest path distances from a single source vertex to all other vertices in the graph using a delta-stepping algorithm.

Neptune Analytics implements the algorithm such that:

- Positive edge weights must be provided using the `edgeWeightProperty` field
- Negative edge weights are not supported.
- The traversal direction cannot be set to `both`.

`.sssp.deltaStepping` syntax

```
CALL neptune.algo.sssp.deltaStepping(
  [source-node list (required)],
  {
    edgeWeightProperty: edge weight predicate for traversal (required)
    edgeWeightType: numeric type of the edge weight property (required)
    delta: the stepping delta (optional)
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    traversalDirection: traversal direction (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD the outputs to generate (source and/or node)
RETURN the outputs to return
```

`.sssp.deltaStepping` inputs

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes to use as the starting location(s) for the algorithm.

  - Each starting node triggers its own execution of the algorithm.
  - If the source-node list is empty then the query result is also empty.
  - If the algorithm is called following a `MATCH` clause (this is known as query-algorithm integration), the output of the `MATCH` clause is used as the source-node list for the algorithm.
- **a configuration object that contains:**
  - **edgeWeightProperty** *(required)* – *type:* `string`; *default: none*.

The edge weight predicate for traversal.

- **edgeWeightType** *(required)* – *type:* `string`; *valid values:* `"int"`, `"long"`, `"float"`, `"double"`.

  The numeric data type of the values in the property specified by `edgeWeightProperty`.

- **delta** *(optional)* – *type:* float; *example:* `3.0`; *default:* `2.0`.

  The delta stepping value.

- **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

  To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **vertexLabel** *(optional)* – *type:* `string`; *example:* `"airport"`; *default:* no node filtering.

  A node label for node filtering. If a node label is provided, vertices matching the label are the only vertices that are included, including vertices in the input list.

- **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

  The direction of edge to follow. Must be one of: `"inbound"` or `"oubound"`.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `sssp.deltaStepping` algorithm

For every node that can be reached from the specified source list, the algorithm returns:

- **source** – The source node.
- **node** – A node found traversing from the source.
- **distance** – The distance between the source node and the found node.

`.sssp.deltaStepping` query examples

This is a standalone query, where a source node (or nodes) is explicitly provided:

```
CALL neptune.algo.sssp.deltaStepping(
  ["101"],
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    edgeWeightType: "int"
  }
)
```

This is a query integration example, where where `.sssp.deltaStepping` follows a `MATCH` clause and uses the output of the `MATCH` clause as its source node list:

```
MATCH (source:airport {code: 'ANC'})
CALL neptune.algo.sssp.deltaStepping(
  source,
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    edgeWeightType: "int",
    vertexLabel: "airport",
    traversalDirection: "outbound",
    concurrency: 1
  }
)
YIELD node, parent, distance
RETURN source, node, parent, distance
```

> ⚠ **Warning**
>
> It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.sssp.deltaStepping` output

Here is an example of the output returned by .sssp.deltaStepping when run against the [sample air-routes dataset [nodes]](), and [sample air-routes dataset [edges]](), when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.sssp.deltaStepping(['101'],
       {edgeWeightProperty: 'dist', edgeWeightType: 'int'})
     yield source, node, distance
     return source, node, distance
     limit 2" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "prscore": 0.002498496090993285,
          "degree": 308,
          "lon": 100.747001647949,
          "wccid": 2357352929951779,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "2709",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 65.4809036254883,
          "elev": 49,
          "longest": 8711,
          "city": "Nadym",
          "type": "airport",
          "region": "RU-YAN",
          "desc": "Nadym Airport",
          "code": "NYM",
          "prscore": 0.00016044313088059425,
          "degree": 18,
          "lon": 72.6988983154297,
          "wccid": 2357352929951779,
          "country": "RU",
          "icao": "USMM",
          "runways": 1
        }
      },
      "distance": 3812
    },
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "prscore": 0.002498496090993285,
          "degree": 308,
          "lon": 100.747001647949,
          "wccid": 2357352929951779,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "2667",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 56.8567008972168,
          "elev": 2188,
          "longest": 6562,
          "city": "Ust-Kut",
          "type": "airport",
          "region": "RU-IRK",
          "desc": "Ust-Kut Airport",
          "code": "UKX",
          "prscore": 0.00005827549999999997,
          "degree": 4,
          "lon": 105.730003356934,
          "wccid": 2357352929951779,
          "country": "RU",
          "icao": "UITT",
          "runways": 1
        }
      },
      "distance": 2993
    }
  ]
}
```

Delta-stepping single source shortest path (SSSP) parents algorithm

The `.sssp.deltaStepping.parents` algorithm computes the shortest path distances from a single source vertex to all other vertices in the graph using a delta-stepping algorithm.

Neptune Analytics implements the algorithm such that:

- Positive edge weights must be provided using the `edgeWeightProperty` field
- Negative edge weights are not supported.
- The traversal direction cannot be set to `both`.

`.sssp.deltaStepping.parents` syntax

```
CALL neptune.algo.sssp.deltaStepping.parents(
  [source-node list (required)],
  {
    edgeWeightProperty: edge weight predicate for traversal (required)
    edgeWeightType: numeric type of the edge weight property (required)
    delta: the stepping delta (optional)
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    traversalDirection: traversal direction (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD the outputs to generate (source and/or node)
RETURN the outputs to return
```

`.sssp.deltaStepping.parents` inputs

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes to use as the starting location(s) for the algorithm.

  - Each starting node triggers its own execution of the algorithm.
  - If the source-node list is empty then the query result is also empty.
  - If the algorithm is called following a `MATCH` clause (this is known as query-algorithm integration), the output of the `MATCH` clause is used as the source-node list for the algorithm.
- **a configuration object that contains:**
  - **edgeWeightProperty** *(required)* – *type:* `string`; *default: none*.

The edge weight predicate for traversal.

- **edgeWeightType** *(required)* – *type:* `string`; *valid values:* `"int"`, `"long"`, `"float"`, `"double"`.

  The numeric data type of the values in the property specified by `edgeWeightProperty`.

- **delta** *(optional)* – *type:* float; *example:* `3.0`; *default:* `2.0`.

  The delta stepping value.

- **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

  To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **vertexLabel** *(optional)* – *type:* `string`; *example:* `"airport"`; *default:* no node filtering.

  A node label for node filtering. If a node label is provided, vertices matching the label are the only vertices that are included, including vertices in the input list.

- **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

  The direction of edge to follow. Must be one of: `"inbound"` or `"oubound"`.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `sssp.deltaStepping.parents` algorithm

For every node that can be reached from the specified source list, the algorithm returns:

- **source** – The source node.
- **node** – A node found traversing from the source.
- **distance** – The distance between the source node and the found node.
- **parent** – The parent of the found node. Note that the parent of the source vertex is itself.

`.sssp.deltaStepping.parents` query examples

This is a standalone query, where a source node (or nodes) is explicitly provided:

```
CALL neptune.algo.sssp.deltaStepping.parents(
  ["101"],
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    edgeWeightType: "int"
  }
)
```

This is a query integration example, where where `.sssp.deltaStepping.parents` follows a `MATCH` clause and uses the output of the `MATCH` clause as its source node list:

```
MATCH (source:airport {code: 'ANC'})
CALL neptune.algo.sssp.deltaStepping.parents(
  source,
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    edgeWeightType: "int",
    vertexLabel: "airport",
    traversalDirection: "outbound",
    concurrency: 1
  }
)
YIELD node, parent, distance
RETURN source, node, parent, distance
```

Warning

It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.sssp.deltaStepping.parents` output

Here is an example of the output returned by .sssp.deltaStepping.parents when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.sssp.deltaStepping.parents(['101'],
       {edgeWeightProperty: 'dist', edgeWeightType: 'int'})
     yield source, node, distance
     return source, node, distance
     limit 2" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "prscore": 0.002498496090993285,
          "degree": 308,
          "lon": 100.747001647949,
          "wccid": 2357352929951779,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "2709",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 65.4809036254883,
          "elev": 49,
          "longest": 8711,
          "city": "Nadym",
          "type": "airport",
          "region": "RU-YAN",
          "desc": "Nadym Airport",
          "code": "NYM",
          "prscore": 0.00016044313088059425,
          "degree": 18,
          "lon": 72.6988983154297,
          "wccid": 2357352929951779,
          "country": "RU",
          "icao": "USMM",
          "runways": 1
        }
      },
      "parent": {
        "~id": "810",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 55.0125999450684,
          "elev": 365,
          "longest": 11818,
          "city": "Novosibirsk",
          "type": "airport",
          "region": "RU-NVS",
          "desc": "Tolmachevo Airport",
          "code": "OVB",
          "prscore": 0.0012910010991618038,
          "degree": 162,
          "lon": 82.6507034301758,
          "wccid": 2357352929951779,
          "country": "RU",
          "icao": "UNNT",
          "runways": 2
        }
      }
    },
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "prscore": 0.002498496090993285,
          "degree": 308,
          "lon": 100.747001647949,
          "wccid": 2357352929951779,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "2667",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 56.8567008972168,
          "elev": 2188,
          "longest": 6562,
          "city": "Ust-Kut",
          "type": "airport",
          "region": "RU-IRK",
          "desc": "Ust-Kut Airport",
          "code": "UKX",
          "prscore": 0.000058275499999999997,
          "degree": 4,
          "lon": 105.730003356934,
          "wccid": 2357352929951779,
          "country": "RU",
          "icao": "UITT",
          "runways": 1
        }
      },
      "parent": {
        "~id": "1038",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 52.2680015563965,
          "elev": 1675,
          "longest": 10384,
          "city": "Irkutsk",
          "type": "airport",
          "region": "RU-IRK",
          "desc": "Irkutsk Airport",
          "code": "IKT",
          "prscore": 0.0008466026629321277,
          "degree": 84,
          "lon": 104.388999938965,
          "wccid": 2357352929951779,
          "country": "RU",
          "icao": "UIII",
          "runways": 1
        }
      }
    }
  ]
}
```

DeltaStepping single source single target shortest path algorithm

The `.sssp.deltaStepping.path` algorithm uses the deltaStepping algorithm to find the shortest path along with the shortest path distances from a source node to a target node in the graph. If there are multiple shortest paths between the source node and the target node, only one will be returned. The algorithm can run only weighted, with `edgeWeightProperty` provided.

Neptune Analytics implements the algorithm such that:

- Positive edge weights must be provided using the `edgeWeightProperty` field.
- Negative edge weights are not supported.
- The traversal direction cannot be set to `both`.

`.sssp.deltaStepping.path` syntax

```
CALL neptune.algo.sssp.deltaStepping.path(
  [source node(s) (required)], [target node(s) (required)]
  {
    edgeWeightProperty: edge weight predicate for traversal (required)
    edgeWeightType: numeric type of the edge weight property (required),
    delta: the stepping delta (optional)
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    traversalDirection: traversal direction (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD the outputs to generate (source and/or node)
RETURN the outputs to return
```

`.sssp.deltaStepping.path` inputs

- **source node(s)** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes to use as the starting location(s) for the algorithm.

  - Each starting node triggers its own execution of the algorithm.
  - If the source node(s) is empty then the query result is also empty.
  - If the algorithm is called following a `MATCH` clause (this is known as query-algorithm integration), the output of the `MATCH` clause is used as the source node(s) for the algorithm.

- **target node(s)** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes to use as the ending location(s) for the algorithm.

  - Each source-target node pair produces an output of the algorithm.
  - If the algorithm is called following a `MATCH` clause (this is known as query-algorithm integration), the output of the `MATCH` clause is used as the target node(s) for the algorithm.

- **a configuration object that contains:**

  - **edgeWeightProperty** *(required)* – *type:* `string`; *default: none*.

    The edge weight predicate for traversal.

  - **edgeWeightType** *(required)* – *type:* `string`; *valid values:* `"int"`, `"long"`, `"float"`, `"double"`.

    The numeric data type of the values in the property specified by `edgeWeightProperty`.

  - **delta** *(optional)* – *type:* `float`; *example:* `3.0`; *default:* 2.0

    The delta stepping value.

  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

  - **vertexLabel** *(optional)* – *type:* `string`; *example:* `"airport"`; *default:* no node filtering.

    A node label for node filtering. If a node label is provided, vertices matching the label are the only vertices that are included, including vertices in the input list.

  - **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

    The direction of edge to follow. Must be one of: `"inbound"` or `"oubound"`.

  - **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

    Controls the number of concurrent threads used to run the algorithm.

    If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `.sssp.deltaStepping.path` algorithm

For every pair of source and target nodes, the algorithm returns:

- **source** – The source vertex.
- **target** – The target vertex.
- **distance** – The total shortest path distance from source to target.
- **vertexPath** – A list of vertices in the path in visit order (including the source and the target).
- **allDistances** – A list of cumulative distances to the vertices in the traversal path (including the source and the target).
- **path** – An openCypher path object representing the shortest path between the source and the target. (A list of vertices from the source vertex to the target vertex, interleaved with the corresponding edges, representing the shortest path. Sequence of vertex id (source), edge id, vertex id, edge id, ..., vertex id (target)).
  - Starts and ends with a vertex, and has edges in between each vertex.
  - Includes the source and the target vertices.

`.sssp.deltaStepping.path` query examples

This is a standalone query, where a source node and target node are explicitly provided:

```
CALL neptune.algo.sssp.deltaStepping.path(
  "9", "37",
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    edgeWeightType: "int",
    delta: 2.0
  }
)
```

This is a query integration example, where `.sssp.deltaStepping.path` follows a `MATCH` clause and uses the output of the `MATCH` clause as its source node(s) and target node(s):

```
MATCH (source:airport {code: 'FLL'})
MATCH (target:airport {code: 'HNL'})
CALL neptune.algo.sssp.deltaStepping.path(
  source, target,
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    delta: 2.0,
    edgeWeightType: "int",
    vertexLabel: "airport",
    traversalDirection: "outbound",
    concurrency: 1
  }
)
YIELD distance, vertexPath, allDistances, path
RETURN source, target, distance, vertexPath, allDistances, path
```

> **Warning**
>
> It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes are returned; and that every source-target node pair produces an output, which can result in a very large query output if a large number of nodes are returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.sssp.deltaStepping.path` output

Here is an example of the output returned by .sssp.deltaStepping.path when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "MATCH (n:airport {code: 'FLL'})
  MATCH (m:airport {code: 'HNL'})
  CALL neptune.algo.sssp.deltaStepping.path(n, m, {
  edgeWeightProperty: "dist",
  edgeWeightType: "int",
  delta: 2.0,
  edgeLabels: ["route"],
  vertexLabel: "airport",
  traversalDirection: "outbound",
  concurrency: 1
  })
  YIELD source, target, distance, vertexPath, allDistances, path
  RETURN source, target, distance, vertexPath, allDistances, path" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [{
      "source": {
        "~id": "9",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "region": "US-FL",
          "runways": 2,
          "country": "US",
          "city": "Fort Lauderdale",
          "type": "airport",
          "icao": "KFLL",
          "lon": -80.152702331542997,
          "code": "FLL",
          "lat": 26.0725994110107,
          "longest": 9000,
          "elev": 64,
          "desc": "Fort Lauderdale/Hollywood International Airport"
        }
      },
      "target": {
        "~id": "37",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "region": "US-HI",
          "runways": 4,
          "country": "US",
          "city": "Honolulu",
          "type": "airport",
          "icao": "PHNL",
          "lon": -157.92199707031199,
          "code": "HNL",
          "lat": 21.318700790405298,
          "longest": 12312,
          "elev": 13,
          "desc": "Honolulu International Airport"
        }
      },
      "distance": 4854,
      "vertexPath": [{
          "~id": "9",
          "~entityType": "node",
          "~labels": ["airport"],
          "~properties": {
            "region": "US-FL",
            "runways": 2,
            "country": "US",
            "city": "Fort Lauderdale",
            "type": "airport",
            "icao": "KFLL",
            "lon": -80.152702331542997,
            "code": "FLL",
            "lat": 26.0725994110107,
            "longest": 9000,
            "elev": 64,
            "desc": "Fort Lauderdale/Hollywood International Airport"
          }
        }, {
          "~id": "11",
          "~entityType": "node",
          "~labels": ["airport"],
          "~properties": {
            "region": "US-TX",
            "runways": 5,
            "country": "US",
            "city": "Houston",
            "type": "airport",
            "icao": "KIAH",
            "lon": -95.341400146484403,
            "code": "IAH",
            "lat": 29.984399795532202,
            "longest": 12001,
            "elev": 96,
            "desc": "George Bush Intercontinental"
          }
        }, {
          "~id": "37",
          "~entityType": "node",
          "~labels": ["airport"],
          "~properties": {
            "region": "US-HI",
            "runways": 4,
            "country": "US",
            "city": "Honolulu",
            "type": "airport",
            "icao": "PHNL",
            "lon": -157.92199707031199,
            "code": "HNL",
            "lat": 21.318700790405298,
            "longest": 12312,
            "elev": 13,
            "desc": "Honolulu International Airport"
          }
        }],
      "allDistances": [0, 964, 4854],
      "path": [{
          "~id": "9",
          "~entityType": "node",
          "~labels": ["airport"],
          "~properties": {
            "region": "US-FL",
            "runways": 2,
            "country": "US",
            "city": "Fort Lauderdale",
            "type": "airport",
            "icao": "KFLL",
            "lon": -80.152702331542997,
            "code": "FLL",
            "lat": 26.0725994110107,
            "longest": 9000,
            "elev": 64,
            "desc": "Fort Lauderdale/Hollywood International Airport"
          }
        }, {
          "~id": "neptune_reserved_1_1152921504607567884",
          "~entityType": "relationship",
          "~start": "9",
          "~end": "11",
          "~type": "route",
          "~properties": {
            "dist": 964
          }
        }, {
          "~id": "11",
          "~entityType": "node",
          "~labels": ["airport"],
          "~properties": {
            "region": "US-TX",
            "runways": 5,
            "country": "US",
            "city": "Houston",
            "type": "airport",
            "icao": "KIAH",
            "lon": -95.341400146484403,
            "code": "IAH",
            "lat": 29.984399795532202,
            "longest": 12001,
            "elev": 96,
            "desc": "George Bush Intercontinental"
          }
        }, {
          "~id": "neptune_reserved_1_1152921508902600717",
          "~entityType": "relationship",
          "~start": "11",
          "~end": "37",
          "~type": "route",
          "~properties": {
            "dist": 3890
          }
        }, {
          "~id": "37",
          "~entityType": "node",
          "~labels": ["airport"],
          "~properties": {
            "region": "US-HI",
            "runways": 4,
            "country": "US",
            "city": "Honolulu",
            "type": "airport",
            "icao": "PHNL",
            "lon": -157.92199707031199,
            "code": "HNL",
            "lat": 21.318700790405298,
            "longest": 12312,
            "elev": 13,
            "desc": "Honolulu International Airport"
          }
        }]
    }]
}
```

TopK hop-limited single source (weighted) shortest path algorithm

The `.topksssp`algorithm finds the single-source weighted shortest paths from a source node to its neighbors out to the distance specified by `maxDepth`. It accumulates the path lengths using the edge weights along the paths and then returns a sorted list of the shortest paths.

`.topksssp` syntax

```
CALL neptune.algo.topksssp(
  [source-node list (required)],
  {
    hopCount: maximum hops on the shortest path (required),
    perHopLimits: [a list of the maximum number of nodes to carry forward at each hop
 (required)],
    edgeLabels: [list of edge labels for filtering (optional)],
    edgeWeightProperty: a numeric edge property to weight the traversal (optional),
    edgeWeightType: numeric type of the specified edgeWeightProperty (optional),
    vertexLabel: a node label for filtering (optional),
    traversalDirection: traversal direction (optional, default: outbound),
    costFunction: determines whether the topK distances are in ascending or descending
 order (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD source, node, distance
RETURN source, node, distance
```

Inputs for the `topksssp` algorithm

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes to use as the starting location(s) for the algorithm.

  - Each starting node triggers its own execution of the algorithm.
  - If the source-node list is empty then the query result is also empty.
  - If the algorithm is called following a `MATCH` clause (this is known as query-algorithm integration), the output of the `MATCH` clause is used as the source-node list for the algorithm.

- **a configuration object that contains:**

  - **hopCount** *(required)* – *type:* positive integer; *default:* none.

Restricts the number of hops on a shortest path, which restricts the number of iterations of the SSSP algorithm to be used.

- **perHopLimits** *(required)* – *type: a list of integers*; *valid values:* positive integers, or `-1` meaning unlimited; *default:* none.

  Each integer represents the maximum number of candidate vertices to carry to the next hop.
- **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

  To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.
- **edgeWeightProperty** *(optional)* – *type:* `string`; *default: none*.

  The edge weight predicate to for traversal. If no property is specified then the algorithm runs unweighted. If multiple properties exist on an edge having the specified name, then one of them is selected at random for the weight value.
- **edgeWeightType** *(optional)* – *type:* `string`; *valid values:* `"int"`, `"long"`, `"float"`, `"double"`.

  The numeric data type of the values in the property specified by `edgeWeightProperty`. If the `edgeWeightProperty` is not present, `edgeWeightType` is ignored and the algorithm runs unweighted. If an edge contains a property specified by `edgeWeightProperty` that has a numeric type different from what is specified in `edgeWeightType`, the property value is typecast to the type specified by `edgeWeightType`.
- **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

  A node label for node filtering. If a node label is provided, vertices matching the label are the only vertices that are included, including vertices in the input list.
- **costFunction** *(optional)* – *type:* `string`; *valid values:* `"min"`, `"max"`; *default: `"min"`*.

  Specifies the ordering of the topK distances returned. A `"min"` value indicates that the topK distances between the source and target vertices should be returned in descending order, whereas a `"max"` value indicates that they should be returned in ascending order.
- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

If set to 0, uses all available threads to complete execution of the individual algorithm invocation. If set to 1, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `topksssp` algorithm

For every node that can be reached from the specified source list, the algorithm returns:

- **source** – The source node.
- **node** – A node found traversing from the source.
- **distance** – The distance between the source node and the found node.

`.topksssp` query examples

This ia a standalone query, where the source node list is explicitly provided in the query:

```
CALL neptune.algo.topksssp(
  ["101"],
  {
    edgeLabels: ["route"],
    hopCount: 3,
    perHopLimits: [10, 100, 1000],
    edgeWeightProperty: "dist",
    edgeWeightType: "int"
  }
)
```

This is a query integration example, where `.topksssp` follows a `MATCH` clause and uses the output of the `MATCH` clause as its source node list:

```
MATCH (n) WHERE id(n) IN ["108","109"]
CALL neptune.algo.topksssp(
  n,
  {
    edgeLabels: ["route"],
    edgeWeightProperty: "dist",
    edgeWeightType: "int",
    hopCount: 5,
    perHopLimits: [5,10,15,20,25]
  }
)
YIELD distance
RETURN n, collect(distance) AS distances'
```

> **Warning**
>
> It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.topksssp` output

Here is an example of the output returned by .topksssp when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier}
  --query-string "CALL neptune.algo.topksssp(['101'], {hopCount: 2, perHopLimits: [3,
 5]})
      YIELD source, node, distance
      RETURN source, node, distance limit 2" \
  --language open_cypher
  /tmp/out.txt

  cat /tmp/out.txt
  {
  "results": [
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "prscore": 0.002498496090993285,
          "degree": 308,
          "lon": 100.747001647949,
          "wccid": 2357352929951779,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "170",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 40.8860015869,
          "elev": 294,
          "longest": 8622,
          "city": "Naples",
          "type": "airport",
          "region": "IT-72",
          "desc": "Naples International Airport",
          "code": "NAP",
          "prscore": 0.001119577675126493,
          "degree": 222,
          "lon": 14.2908000946,
          "wccid": 2357352929951779,
          "country": "IT",
          "icao": "LIRN",
          "runways": 1
        }
      },
      "distance": 2.0
    },
    {
      "source": {
        "~id": "101",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 13.6810998916626,
          "elev": 5,
          "longest": 13123,
          "city": "Bangkok",
          "type": "airport",
          "region": "TH-10",
          "desc": "Suvarnabhumi Bangkok International Airport",
          "code": "BKK",
          "prscore": 0.002498496090993285,
          "degree": 308,
          "lon": 100.747001647949,
          "wccid": 2357352929951779,
          "country": "TH",
          "icao": "VTBS",
          "runways": 2
        }
      },
      "node": {
        "~id": "12",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 40.63980103,
          "elev": 12,
          "longest": 14511,
          "city": "New York",
          "type": "airport",
          "region": "US-NY",
          "desc": "New York John F. Kennedy International Airport",
          "code": "JFK",
          "prscore": 0.002885053399950266,
          "degree": 403,
          "lon": -73.77890015,
          "wccid": 2357352929951779,
          "country": "US",
          "icao": "KJFK",
          "runways": 4
        }
      },
      "distance": 2.0
    }
  ]
}
```

Egonet algorithms

This EgoNet algorithm finds the (filtered) EgoNet of a vertex to its hopCount-neighbors. An EgoNet, also known as the egocentric network, is a subgraph of a social network that encapsulates the connections of a single individual, known as the ego, and all the people they are socially connected to, known as alters. EgoNet can be used for further analysis in social networks.

Neptune Analytics supports the following EgoNet algorithms:

- .egonet – The EgoNet algorithm finds the (filtered) EgoNet of a vertex to its hopCount-neighbors. An EgoNet, also known as the egocentric network, is a subgraph of a social network that encapsulates the connections of a single individual, known as the ego, and all the people they are socially connected to, known as alters.
- .egonet.edgeList – This algorithm has a different output schema than egonet.

.egonet

This EgoNet algorithm finds the (filtered) EgoNet of a vertex to its hopCount-neighbors. An EgoNet, also known as the egocentric network, is a subgraph of a social network that encapsulates the connections of a single individual, known as the ego, and all the people they are socially connected to, known as alters.

For each hop, the algorithm gets the topK (K is specified per hop by the user via perHopMaxNeighbor) neighbors those have the highest/lowest (based on the costFunction) edge weights, and these neighbors become the source vertices for the next hop. The algorithm assumes the graph is an edge weighted graph.

.egonet syntax

```
CALL neptune.algo.egonet(
  [source/ego-node list (required)],
  {
    hopCount: fixed hops of traversal (required),
    perHopMaxNeighbor: [list of the max number of top neighor vertices at each hop
 (required)],
    perHopEdgeWeightProperty: [list of edge weight predicates at each hop (required)],
    edgeWeightType: numeric type of the specified edgeWeightProperty (required),
    edgeLabels: [list of edge labels for filtering (optional)],
    perHopVertexLabel: [list of node labels for filtering at each hop(optional)],
    perHopTraversalDirection: [list of traversal directions at each hop (optional,
 default: outbound)],
    costFunction: determines whether the edges having the maximum weights or the
 minimum weight will be included in the EgoNet (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD egoNode, nodeList, edgeList
RETURN egoNode, nodeList, edgeList
```

Inputs for the .egonet algorithm

- **a source/ego node list** *(required)* – *type:* Node[] or NodeId[]; *default: none*.

  The node or nodes to use as the starting location(s) for the algorithm.

  - Each starting node triggers its own execution of the algorithm.
  - If the source-node list is empty then the query result is also empty.

  - If the algorithm is called following a MATCH clause (this is known as query-algorithm integration), the output of the MATCH clause is used as the source-node list for the algorithm.

- **a configuration object that contains:**

  - **hopCount** *(required)* – *type:* positive integer; *valid values:* 1, 2 or 3; other values will be rejected. *default:* none.

    Restricts the number of hops during traversal.

  - **perHopMaxNeighbor** *(required)* – *type: a list of integers*; *valid values:* positive integers, or `-1` meaning unlimited; *default:* none.

    Each integer represents the maximum number of candidate vertices to carry to the next hop. It should have the same size as the value of `hopCount`.

  - **perHopEdgeWeightProperty** *(required)* – *type: a list of strings*; *default:* none.

    The edge weight predicate for traversal at each hop. If multiple properties exist on an edge having the specified name, then one of them is selected at random for the weight value. It should have the same size as the value of `hopCount`.

  - **edgeWeightType** *(required)* – *type: string*; *valid values:* "int", "long", "float", "double"; *default:* none.

    The numeric data type of the values in the property specified by `perHopEdgeWeightProperty`. If an edge contains a property specified by `perHopEdgeWeightProperty` that has a numeric type different from what is specified in `edgeWeightType`, the property value is typecast to the type specified by `edgeWeightType`.

  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

  - **perHopVertexLabel** *(optional)* – *type:* a list of vertex label strings; *default: none*.

    A list of node labels for node filtering at each hop. At each hop, if a node label is provided, vertices matching the label are the only vertices that are included, including vertices in the input list. It should have the same size as the value of `hopCount`.

  - **perHopTraversalDirection** *(optional)* – *type:* a list of strings; *valid values:* "inbound","outbound", or "both"; *default:* outbound.

The direction of edge to follow at each hop. It should have the same size as the value of `hopCount`.

- **costFunction** *(optional)* – *type:* `string`; *valid values:* `"min"`, `"max"`; *default: `"max"`*.

  This determines whether the edges having the maximum weights or the minimum weight will be included in the EgoNet adhering the `perHopMaxNeigbor` limits. A `min` value indicates that the edge with minimum weights will be included in the EgoNet, whereas a `max` value indicates that the edge with maximum weights will be included in the EgoNet.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to 1, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `.egonet` algorithm

The .egonet algorithm returns:

- **egoNode** – The ego vertex for the egonet.
- **nodeList** – A list of traversed vertices from the ego vertex.
- **edgeList** – A list of traversed edges from the ego vertex.

`.egonet` query examples

This ia a standalone query, where the source node list is explicitly provided in the query:

```
CALL neptune.algo.egonet(["101"], {
hopCount: 2,
perHopMaxNeighbor: [-1,-1],
edgeLabels: ["route"],
perHopEdgeWeightProperty: ["dist", "dist"],
edgeWeightType: "int",
perHopVertexLabel: ["airport", "airport"],
perHopTraversalDirection: ["outbound", "outbound"],
costFunction: "max",
concurrency: 1
})
YIELD egoNode, nodeList, edgeList
RETURN egoNode, nodeList, edgeList
```

This is a query integration example, where `.egonet` follows a `MATCH` clause and uses the output of the `MATCH` clause as its source node list:

```
MATCH (n:airport {code: 'ANC'})
CALL neptune.algo.egonet(n, {
hopCount: 2,
perHopMaxNeighbor: [-1,-1],
edgeLabels: ["route"],
perHopEdgeWeightProperty: ["dist", "dist"],
edgeWeightType: "int",
perHopVertexLabel: ["airport", "airport"],
perHopTraversalDirection: ["outbound", "outbound"],
costFunction: "max",
concurrency: 1
})
YIELD nodeList, edgeList
RETURN n, nodeList, edgeList
```

Warning

It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.egonet` output

Here is an example of the output returned by .egonet when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier}
  --query-string "CALL neptune.algo.egonet(["1"], \
  {perHopEdgeWeightProperty: ["dist"], \
  edgeWeightType: "int", \
  hopCount: 1, \
  perHopMaxNeighbor: [3], \
  perHopTraversalDirection: ["both"]}) \
  yield egoNode, edgeList, nodeList \
  return egoNode, edgeList, nodeList" \
  --language open_cypher
  /tmp/out.txt

  cat /tmp/out.txt
{
 "results": [{
 "egoNode": {
 "~id": "1",
 "~entityType": "node",
 "~labels": ["airport"],
 "~properties": {
 "region": "US-GA",
 "runways": 5,
 "country": "US",
 "city": "Atlanta",
 "type": "airport",
 "icao": "KATL",
 "lon": -84.4281005859375,
 "code": "ATL",
 "lat": 33.6366996765137,
 "longest": 12390,
 "elev": 1026,
 "desc": "Hartsfield - Jackson Atlanta International Airport"
 }
},
 "edgeList": [{
 "~id": "neptune_reserved_1_1152921770894950415",
 "~entityType": "relationship",
 "~start": "67",
 "~end": "1",
 "~type": "route",
 "~properties": {
 "dist": 7640
 }
 }, {
 "~id": "neptune_reserved_1_1152922020003053583",
 "~entityType": "relationship",
 "~start": "126",
 "~end": "1",
 "~type": "route",
 "~properties": {
 "dist": 8434
 }
 }, {
 "~id": "neptune_reserved_1_1152921521787699214",
 "~entityType": "relationship",
 "~start": "1",
 "~end": "58",
 "~type": "route",
 "~properties": {
 "dist": 7581
 }
 }],
 "nodeList": [{
 "~id": "126",
 "~entityType": "node",
 "~labels": ["airport"],
 "~properties": {
 "region": "ZA-GT",
 "runways": 2,
 "country": "ZA",
 "city": "Johannesburg",
 "type": "airport",
 "icao": "FAJS",
 "lon": 28.2460002899,
 "code": "JNB",
 "lat": -26.139200210599999,
 "longest": 14495,
 "elev": 5558,
 "desc": "Johannesburg, OR Tambo International Airport"
 }
 }, {
 "~id": "67",
 "~entityType": "node",
 "~labels": ["airport"],
 "~properties": {
 "region": "CN-31",
 "runways": 2,
 "country": "CN",
 "city": "Shanghai",
 "type": "airport",
 "icao": "ZSPD",
 "lon": 121.80500030517599,
 "code": "PVG",
 "lat": 31.1434001922607,
 "longest": 13123,
 "elev": 13,
 "desc": "Shanghai - Pudong International Airport"
 }
 }, {
 "~id": "58",
 "~entityType": "node",
 "~labels": ["airport"],
 "~properties": {
 "region": "AE-DU",
 "runways": 2,
 "country": "AE",
 "city": "Dubai",
 "type": "airport",
 "icao": "OMDB",
 "lon": 55.364398956300001,
 "code": "DXB",
 "lat": 25.2527999878,
 "longest": 13124,
 "elev": 62,
 "desc": "Dubai International Airport"
 }
 }, {
 "~id": "1",
 "~entityType": "node",
 "~labels": ["airport"],
 "~properties": {
 "region": "US-GA",
 "runways": 5,
 "country": "US",
 "city": "Atlanta",
 "type": "airport",
 "icao": "KATL",
 "lon": -84.4281005859375,
 "code": "ATL",
 "lat": 33.6366996765137,
 "longest": 12390,
 "elev": 1026,
 "desc": "Hartsfield - Jackson Atlanta International Airport"
 }
 }]
 }]
```

```
}
```

.egonet.edgelist

This EgoNet EdgeList algorithm is the same as the standard EgoNet algorithm, except that this variant has a different output schema, which returns the EgoNet in an edge list form.

.egonet.edgelist syntax

```
CALL neptune.algo.egonet.edgelist(
  [source/ego-node list (required)],
  {
    hopCount: fixed hops of traversal (required),
    perHopMaxNeighbor: [list of the max number of top neighor vertices at each hop (required)],
    perHopEdgeWeightProperty: [list of edge weight predicates at each hop (required)],
    edgeWeightType: numeric type of the specified edgeWeightProperty (required),
    edgeLabels: [list of edge labels for filtering (optional)],
    perHopVertexLabel: [list of node labels for filtering at each hop(optional)],
    perHopTraversalDirection: [list of traversal directions at each hop (optional, default: outbound)],
    costFunction: determines whether the edges having the maximum weights or the minimum weight will be included in the EgoNet (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD egoNode, source, target, weight
RETURN egoNode, source, target, weight
```

Inputs for the .egonet.edgelist algorithm

- **a source/ego node list** *(required)* – *type:* Node[] or NodeId[]; *default: none*.

  The node or nodes to use as the starting location(s) for the algorithm.

  - Each starting node triggers its own execution of the algorithm.
  - If the source-node list is empty then the query result is also empty.
  - If the algorithm is called following a MATCH clause (this is known as query-algorithm integration), the output of the MATCH clause is used as the source-node list for the algorithm.
- **a configuration object that contains:**
  - **hopCount** *(required)* – *type:* positive integer; *valid values:* 1, 2 or 3; other values will be rejected. *default:* none.

Restricts the number of hops during traversal.

- **perHopMaxNeighbor** *(required)* – *type: a list of integers*; *valid values:* positive integers, or `-1` meaning unlimited; *default:* none.

  Each integer represents the maximum number of candidate vertices to carry to the next hop. It should have the same size as the value of `hopCount`.

- **perHopEdgeWeightProperty** *(required)* – *type: a list of strings*; *default:* none.

  The edge weight predicate for traversal at each hop. If multiple properties exist on an edge having the specified name, then one of them is selected at random for the weight value. It should have the same size as the value of `hopCount`.

- **edgeWeightType** *(required)* – *type: string*; *valid values:* "int", "long", "float", "double"; *default:* none.

  The numeric data type of the values in the property specified by `perHopEdgeWeightProperty`. If an edge contains a property specified by `perHopEdgeWeightProperty` that has a numeric type different from what is specified in `edgeWeightType`, the property value is typecast to the type specified by `edgeWeightType`.

- **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

  To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **perHopVertexLabel** *(optional)* – *type:* a list of vertex label strings; *default: none*.

  A list of node labels for node filtering at each hop. At each hop, if a node label is provided, vertices matching the label are the only vertices that are included, including vertices in the input list. It should have the same size as the value of `hopCount`.

- **perHopTraversalDirection** *(optional)* – *type:* a list of strings; *valid values:* "inbound","outbound", or "both"; *default:* outbound.

  The direction of edge to follow at each hop. It should have the same size as the value of `hopCount`.

- **costFunction** *(optional)* – *type:* `string`; *valid values:* `"min"`, `"max"`; *default: `"max"`*.

  This determines whether the edges having the maximum weights or the minimum weight will be included in the EgoNet adhering the `perHopMaxNeigbor` limits. A `min` value indicates that the edge with minimum weights will be included in the EgoNet, whereas a `max` value indicates that the edge with maximum weights will be included in the EgoNet.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `.egonet.edgelist` algorithm

The .egonet.edgelist algorithm returns:

- **egoNode** – The ego vertex of the egonet.
- **source** – The source vertex of an edge in the weighted edge list.
- **target** – The source vertex of an edge in the weighted edge list.
- **weight** – The edge weight of the source-target edge in the weighted edge list.

`.egonet.edgelist` query examples

This ia a standalone query, where the source node list is explicitly provided in the query:

```
CALL neptune.algo.egonet.edgeList(["101"], {
hopCount: 2,
perHopMaxNeighbor: [-1,-1],
edgeLabels: ["route"],
perHopEdgeWeightProperty: ["dist", "dist"],
edgeWeightType: "int",
perHopVertexLabel: ["airport", "airport"],
perHopTraversalDirection: ["outbound", "outbound"],
costFunction: "max",
concurrency: 1
})
YIELD egoNode, source, target, weight
RETURN egoNode, source, target, weight
```

This is a query integration example, where `.egonet.edgelist` follows a `MATCH` clause and uses the output of the `MATCH` clause as its source node list:

```
MATCH (n:airport {code: 'ANC'})
CALL neptune.algo.egonet.edgeList(n, {
hopCount: 2,
perHopMaxNeighbor: [-1,-1],
edgeLabels: ["route"],
perHopEdgeWeightProperty: ["dist", "dist"],
edgeWeightType: "int",
perHopVertexLabel: ["airport", "airport"],
perHopTraversalDirection: ["outbound", "outbound"],
costFunction: "max",
concurrency: 1
})
YIELD source, target, weight
RETURN n, source, target, weight
```

> **Warning**
>
> It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.egonet.edgelist` output

Here is an example of the output returned by .egonet.edgelist when run against the [sample air-routes dataset [nodes]](), and [sample air-routes dataset [edges]](), when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier}
  --query-string "CALL neptune.algo.egonet(["1"], \
  {perHopEdgeWeightProperty: ["dist", "dist"], \
  edgeWeightType: "int", \
  hopCount: 2, \
  perHopMaxNeighbor: [30, 30], \
  perHopTraversalDirection: ["both", "both"]}) \
  YIELD egoNode, source, target, weight \
  RETURN egoNode, source, target, weight limit 2" \
  --language open_cypher
  /tmp/out.txt
  cat /tmp/out.txt
{
 "results": [{
 "egoNode": {
 "~id": "1",
 "~entityType": "node",
 "~labels": ["airport"],
 "~properties": {
 "region": "US-GA",
 "runways": 5,
 "country": "US",
 "city": "Atlanta",
 "type": "airport",
 "icao": "KATL",
 "lon": -84.4281005859375,
 "code": "ATL",
 "lat": 33.6366996765137,
 "longest": 12390,
 "elev": 1026,
 "desc": "Hartsfield - Jackson Atlanta International Airport"
 }
 },
 "source": {
 "~id": "1",
 "~entityType": "node",
 "~labels": ["airport"],
 "~properties": {
 "region": "US-GA",
 "runways": 5,
 "country": "US",
 "city": "Atlanta",
 "type": "airport",
 "icao": "KATL",
 "lon": -84.4281005859375,
 "code": "ATL",
 "lat": 33.6366996765137,
 "longest": 12390,
 "elev": 1026,
 "desc": "Hartsfield - Jackson Atlanta International Airport"
 }
 },
 "target": {
 "~id": "27",
 "~entityType": "node",
 "~labels": ["airport"],
 "~properties": {
 "region": "US-CA",
 "runways": 3,
 "country": "US",
 "city": "Long Beach",
 "type": "airport",
 "icao": "KLGB",
 "lon": -118.15200040000001,
 "code": "LGB",
 "lat": 33.817699429999998,
 "longest": 10003,
 "elev": 60,
 "desc": "Long Beach Airport"
 }
 },
 "weight": 2460
 }, {
 "egoNode": {
 "~id": "1",
 "~entityType": "node",
 "~labels": ["airport"],
 "~properties": {
 "region": "US-GA",
 "runways": 5,
 "country": "US",
 "city": "Atlanta",
 "type": "airport",
 "icao": "KATL",
 "lon": -84.4281005859375,
 "code": "ATL",
 "lat": 33.6366996765137,
 "longest": 12390,
 "elev": 1026,
 "desc": "Hartsfield - Jackson Atlanta International Airport"
 }
 },
 "source": {
 "~id": "134",
 "~entityType": "node",
 "~labels": ["airport"],
 "~properties": {
 "region": "PE-LIM",
 "runways": 1,
 "country": "PE",
 "city": "Lima",
 "type": "airport",
 "icao": "SPIM",
 "lon": -77.1143035889,
 "code": "LIM",
 "lat": -12.021900176999999,
 "longest": 11506,
 "elev": 113,
 "desc": "Lima, Jorge Chavez International Airport"
 }
 },
 "target": {
 "~id": "1",
 "~entityType": "node",
 "~labels": ["airport"],
 "~properties": {
 "region": "US-GA",
 "runways": 5,
 "country": "US",
 "city": "Atlanta",
 "type": "airport",
 "icao": "KATL",
 "lon": -84.4281005859375,
 "code": "ATL",
 "lat": 33.6366996765137,
 "longest": 12390,
 "elev": 1026,
 "desc": "Hartsfield - Jackson Atlanta International Airport"
 }
 },
 "weight": 3189
 }]
}
```

Centrality algorithms in Neptune Analytics

Centrality algorithms utilize the topology of a network to determine the relative importance or influence of a specific node within the graph. By measuring the relative importance of a node or edge within a network, centrality values can indicate which elements in a graph play a critical role in that network.

By identifying the most influential or important nodes within a network, centrality algorithms can provide insights about key players or critical points of interaction. This is valuable in social network analysis, where it helps pinpoint influential individuals, and in transportation networks, where it aids in identifying crucial hubs for efficient routing and resource allocation.

Different types of centrality algorithms use different techniques to measure the importance of a node. Understanding how an algorithm calculates centrality is important to understanding the meaning of its outputs.

In addition to returning centrality data to the client, Neptune Analytics provides mutate variations of the centrality algorithms which store the calculated centrality values as vertex properties in the graph.

Neptune Analytics supports three centrality algorithms along with their mutate variants:

- `degree` – This measures a nodes's centrality by the number of edges connected to it, and can therefore be used to find the most connected nodes in a network.
- `degree.mutate` – The degree centrality mutate algorithm measures the number of incident edges of each vertex it traverses and writes that calculated degree value as a property of the vertex.
- `pageRank` – This is an iterative algorithm that measures a nodes's centrality by the number and quality of incident edges and adjacent vertices. The centrality of a node connected to a few important nodes may therefore be higher than that of a node connected to many less important nodes. The output of this algorithm is a value that indicates the importance of a given node relative to the other nodes in the graph.
- `pageRank.mutate` – This algorithm stores the calculated PageRank of a given node as a property of the node.
- `closenessCentrality` – This algorithm computes the closeness centrality (CC) metric of nodes in a graph. The closeness centrality metric of a vertex is a positive measure of how close it is to all other vertices, or how central it is in the graph. Because it indicates how quickly all other

nodes in a network can be reached from a given node, it can be used in transportation networks to identify key hub locations, and in disease-spread modeling to pinpoint central locations for targeted intervention efforts.

- `closenessCentrality.mutate` – This algorithm computes the closeness centrality (CC) metric of vertices in a graph and writes them as a property of each vertex.

Degree centrality algorithm

The `.degree` centrality algorithm counts the number of incident edges at each node that it traverses. This measure of how connected the node is can in turn indicate the node's importance and level of influence in the network.

The `.degree` algorithm is used in social networks to identify popular individuals with many connections, in transportation networks to locate central hubs with numerous roads leading to and from them, and in web analysis to find influential web pages with many incoming links.

The time complexity of `.degree` is `O(|E|)`, where `|E|` is the number of edges in the graph. The space complexity is `O(|V|)`, where `|V|` is the number of vertices in the graph.

`.degree` syntax

```
CALL neptune.algo.degree(
  [node list (required)],
  {
    edgeLabels: [a list of edge labels for filtering (optional)],
    vertexLabel: "a node label for filtering (optional)",
    traversalDirection: traversal direction (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD node, degree
RETURN node, degree
```

Inputs for the `.degree` algorithm

- **a node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes for which to return the edge count (degree). If an empty list is provided, the query result is also empty.

  If the algorithm is called following a `MATCH` clause (query integration), the result returned by the `MATCH` clause is taken as the node list.

- **a configuration object that contains:**
  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

  A vertex label for vertex filtering. If a vertex label is provided, vertices matching the label are the only vertices that will be included in the calculation. Furthermore, vertices in the input that do not satisfy this constraint will not have results returned.

- **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

  The direction of edge to follow. Must be one of: `"inbound"`, `"oubound"`, or `"both"`.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

`.degree` outputs

- **node** – A list of the requested nodes. If `vertexLabel` is present, only the requested nodes that match the `vertexLabel` value are included.
- **degree** – A list of corresponding degree values for the nodes with respect to edges with a label specified in `edgeLabels`.

If the input vertex list is empty, the output is empty.

Query examples for `.degree`

This is a standalone example, where the source node list is explicitly specified in the query:

```
CALL neptune.algo.degree(["101"], {edgeLabel: "route"})
```

This is a more complicated standalone query submitted using the AWS CLI:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string 'CALL neptune.algo.degree(
        ["101", "102", "103"],
        {
          edgeLabels: ["route"],
          vertexLabel: "airport",
          traversalDirection: "inbound",
          concurrency: 2
        }
      )
      YIELD node, degree
      RETURN node, degree' \
  --language open_cypher \
  /tmp/out.txt
```

This is a query integration example with frontier injection, where `.degree` follows a `MATCH` clause and finds the degree value for all vertices returned by `MATCH(n:airport)`:

```
MATCH(n:airport)
CALL neptune.algo.degree(n, {edgeLabels: ["route"]})
YIELD degree
RETURN n, degree'
```

This is an example of multiple `.degree` invocations chained together, where the output of one invocation serves as the input of another:

```
CALL neptune.algo.degree(
  ["108"],
  {
    edgeLabels: ["route"],
    vertexLabel: "airport"
  }
)
YIELD node
CALL neptune.algo.degree(
  node,
  {
    edgeLabels: ["route"],
    vertexLabel: "airport"
  }
)
YIELD node AS node2 WITH id(node2) AS id
RETURN id
```

> **Warning**
>
> It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.degree` output

Here is an example of the output returned by .degree when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string 'MATCH (n)
      CALL neptune.algo.degree(n)
      YIELD node, degree
      RETURN node, degree
      LIMIT 2' \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "node": {
        "~id": "10",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 38.94449997,
          "elev": 313,
          "longest": 11500,
          "city": "Washington D.C.",
          "type": "airport",
          "region": "US-VA",
          "desc": "Washington Dulles International Airport",
          "code": "IAD",
          "prscore": 0.002264724113047123,
          "lon": -77.45580292,
          "wccid": 2357352929951779,
          "country": "US",
          "icao": "KIAD",
          "runways": 4
        }
      },
      "degree": 312
    },
    {
      "node": {
        "~id": "12",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 40.63980103,
          "elev": 12,
          "longest": 14511,
          "city": "New York",
          "type": "airport",
          "region": "US-NY",
          "desc": "New York John F. Kennedy International Airport",
          "code": "JFK",
          "prscore": 0.002885053399950266,
          "lon": -73.77890015,
          "wccid": 2357352929951779,
          "country": "US",
          "icao": "KJFK",
          "runways": 4
        }
      },
      "degree": 403
    }
  ]
}
```

Degree mutate centrality algorithm

The `.degree.mutate` centrality algorithm counts the number of incident edges of every node in the graph. This measure of how connected the node is can in turn indicate the node's importance and level of influence in the network. The `.degree.mutate` algorithm then stores each node's calculated degree value as a property of the node.

The algorithm returns a single success flag (`true` or `false`), which indicates whether the writes succeeded or failed.

`.degree.mutate` syntax

```
CALL neptune.algo.degree.mutate(
  {
    writeProperty: A name for the new node property where the degree values will be
 written,
    edgeLabels: [a list of edge labels for filtering (optional)],
    vertexLabel: "a node label for filtering (optional)",
    traversalDirection: traversal direction (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD success
RETURN success
```

`.degree.mutate` inputs

a configuration object that contains:

- **writeProperty** *(required)* – *type:* `string`; *default: none*.

  A name for the new node property that will contain the computed degree values.

- **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

  To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

  A node label for node filtering. If `vertexLabel` is provided, vertices matching the label are the only vertices that are processed, including vertices in the input list.

- **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

  The direction of edge to follow. Must be one of: `"inbound"`, `"oubound"`, or `"both"`.
- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Output of the `.degree.mutate` algorithm

The computed degree values are written to a new vertex property using the property name specified by the `writeProperty` input parameter.

A single Boolean `success` value (`true` or `false`) is returned, which indicates whether or not the writes succeeded.

`.degree.mutate` query examples

The example below is a standalone example, where the source vertex list is explicitly provided in the query.

This query writes the degree values of all nodes in the graph to a new vertex property called `DEGREE`:

```
CALL neptune.algo.degree.mutate({writeProperty: "DEGREE", edgeLabels: ["route]})
```

After using the mutate algorithm, the newly written properties can then be accessed in subsequent queries. For example, after the mutate algorithm call above, you could use the following query to retrieve the `.degree` property of specific nodes:

```
MATCH (n) WHERE id(n) IN ["101", "102", "103"]
RETURN n.DEGREE'
```

Sample output from `.degree.mutate`

Here is an example of the output returned by .degree.mutate when run against the [sample air-routes dataset [nodes]](), and [sample air-routes dataset [edges]](), when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.degree.mutate({writeProperty: 'degree'}) YIELD
 success RETURN success" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    { "success": true }
  ]
}
```

PageRank centrality algorithm

PageRank is an algorithm originally developed by Larry Page and Sergey Brin, co-founders of Google. It was originally developed to rank web pages in search engine results. The PageRank score for a given node is calculated based on the number and quality of the edges pointing to that node, as well as the importance of the nodes that are connected to it. The PageRank algorithm assigns a higher score to nodes that are linked to other high-scoring nodes, and a lower score to nodes that are linked to low-scoring nodes.

The output of PageRank can be visualized as a ranking metric for the importance of a node within a given graph, with the most important nodes having the highest score, and the least important node having the lowest score. PageRank is used in search engines to rank web pages based on their importance and influence, in citation networks to identify highly cited scientific papers, and in recommendation systems to suggest popular and relevant content to users.

The space complexity of the .pageRank algorithm is:

```
O(|V|) + [storage of data O(|V|+|E|)]
```

In more detail, this is:

```
|V| * (3 * sizeof(local id type) +
       2 * sizeof(double) +
       sizeof(global id type)) +
       O(|1DData Parts|)
```

.pageRank syntax

```
CALL neptune.algo.pageRank(
  [list of source nodes (required)],
  {
    numOfIterations: a small positive integer like 20 (optional),
    dampingFactor: a positive float less than 1.0, like 0.85 (optional)
    edgeLabels: [a list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD node, rank
RETURN node, rank
```

`.pageRank` inputs

- **a source vertex list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes to use as the starting locations for the algorithm. Each node in the list triggers an execution of the algorithm. If an empty list is provided, the query result is also empty.

  If the algorithm is called following a `MATCH` clause (query algo integration), the source query list is the result returned by the `MATCH` clause.

- **a configuration object that contains:**

  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

  - **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

    A vertex label for vertex filtering. If a vertex label is provided, vertices matching the label are the only vertices that are included, including vertices in the input list.

  - **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

    The direction of edge to follow. Must be one of: `"oubound"` or `"inbound"`.

  - **numOfIterations** *(optional)* – *type:* a positive integer greater than zero; *default: 20*.

    The number of iterations to perform to reach convergence. A number between 10 and 20 is recommended.

  - `dampingFactor` *(optional)* – *type:* a positive floating-point number less than `1.0`; *default:* `0.85`.

    A positive floating-point damping factor between 0.0 and 1.0 that expresses the probability, at any step, that the node will continue.

  - **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

    Controls the number of concurrent threads used to run the algorithm.

    If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the .pageRank algorithm

- **node** – A key column of the input nodes.
- **rank** – A key column of the corresponding page-rank scores for those nodes.

If the input nodes list is empty, the output is empty.

Query examples for .pageRank

This is a standalone example, where the source vertex list is explicitly specified in the query.

```
CALL neptune.algo.pageRank(
  ["101"],
  {
    numOfIterations: 1,
    dampingFactor: 0.85,
    edgeLabels: ["route"]
  }
)
```

This is a query integration examples, where `.pageRank` follows a `MATCH` clause and uses frontier injection to take the output of the `MATCH` clause as its list of source nodes:

```
MATCH (n)
CALL neptune.algo.pageRank(
  n,
  {
    dampingFactor: 0.85,
    numOfIterations: 1,
    edgeLabels: ["route"]
  }
)
YIELD rank
RETURN n, rank
```

This query is an example of constraining the results of `.pageRank` based on the PageRank values, and returning them in ascending order:

```
MATCH (n)
CALL neptune.algo.pageRank(
  n,
  {
    numOfIterations: 10,
    dampingFactor: 0.85,
    vertexLabel: "airport",
    edgeLabels: ["route"]
  }
)
YIELD rank WHERE rank > 0.004
RETURN n, rank ORDER BY rank
```

Sample `.pageRank` output

Here is an example of the output returned by .pageRank when run against the [sample air-routes dataset [nodes]](), and [sample air-routes dataset [edges]](), when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.pageRank(n) YIELD node, rank RETURN node, rank
 LIMIT" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "node": {
        "~id": "2709",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 65.4809036254883,
          "elev": 49,
          "longest": 8711,
          "city": "Nadym",
          "type": "airport",
          "region": "RU-YAN",
          "desc": "Nadym Airport",
          "code": "NYM",
          "lon": 72.6988983154297,
          "country": "RU",
          "icao": "USMM",
          "runways": 1
        }
      },
      "rank": 0.00016044313088059425
    },
    {
      "node": {
        "~id": "3747",
        "~entityType": "node",
        "~labels": ["continent"],
        "~properties": {
          "code": "AN",
          "type": "continent",
          "desc": "Antarctica"
        }
      },
      "rank": 0.0000404242
    }
  ]
}
```

PageRank mutate centrality algorithm

The ranking metric computed by `.pageRank.mutate` can indicate the importance of a node within a given graph, with the most important nodes having the highest score, and the least important node having the lowest score. PageRank is used in search engines to rank web pages based on their importance and influence, in citation networks to identify highly cited scientific papers, and in recommendation systems to suggest popular and relevant content to users.

The mutate variant of the PageRank algorithm performs the PageRank calculation over the entire graph unless the configuration parameters establish a filter, and each traversed node's calculated PageRank value is stored on that node as a property.

`pageRank.mutate` inputs

Inputs for the `pageRank.mutate` algorithm are passed in a configuration object parameter that contains:

- **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

  To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.
- **writeProperty** *(required)* – *type:* `string`; *default: none*.

  A name for the new vertex property that will contain the computed PageRank values. If a property of that name already exists, it is overwritten.
- **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

  A vertex label for vertex filtering. If a vertex label is provided, vertices matching the label are the only vertices that are included, including vertices in the input list.
- **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

  The direction of edge to follow. Must be one of: `"oubound"` or `"inbound"`.
- **numOfIterations** *(optional)* – *type:* a positive integer greater than zero; *default: 20*.

  The number of iterations to perform to reach convergence. A number between 10 and 20 is recommended.
- `dampingFactor` *(optional)* – *type:* a positive floating-point number less than `1.0`; *default:* `0.85`.

A positive floating-point damping factor between 0.0 and 1.0 that expresses the probability, at any step, that the node will continue.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `pageRank.mutate` algorithm

The computed PageRank values are written to a new vertex property on each node using the property name specified by the `writeProperty` input parameter.

A single Boolean `success` value (`true` or `false`) is returned, which indicates whether or not the writes succeeded.

Query example for `pageRank.mutate`

The example below computes the PageRank score of every vertex in the graph, and writes that score to a new vertex property named P_RANK:

```
CALL neptune.algo.pageRank.mutate(
  {
    writeProperty:"P_RANK",
    dampingFactor: 0.85,
    numOfIterations: 1,
    edgeLabels: ["route"]
  }
)
```

This query illustrates how you could then access the PageRank values in the P_RANK vertex property. It counts how many nodes have a P_RANK property value greater than the "SEA" node's P_RANK property value:

```
MATCH (n) WHERE n.code = "SEA" WITH n.P_RANK AS lowerBound
MATCH (m) WHERE m.P_RANK > lowerBound
RETURN count(m)
```

Sample `.pageRank.mutate` output

Here is an example of the output returned by .pageRank.mutate when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.pageRank.mutate({writeProperty: 'prscore'}) YIELD
 success RETURN success" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    { "success": true }
  ]
}
```

Closeness centrality algorithm

The closeness centrality algorithm computes a Closeness Centrality (CC) metric for specified nodes in a graph. The CC metric of a node can be used as a positive measure of how close it is to all other nodes or how central it is in the graph.

The CC metric can be interpreted to show how quickly all other nodes in a network can be reached from a given node, and how important it is as a central hub for rapid information flow. It can be used in transportation networks to identify key hub locations, and in disease-spread modeling to pinpoint central points for targeted intervention efforts.

The closeness centrality (CC) score of a node is calculated based on the sum of its distances to all other nodes. The CC score itself is the inverse of that number; in other words, one divided by that sum. In practice, the calculation is commonly normalized to use the average length of the shortest paths rather than the actual sum of their lengths.

`.closenessCentrality` syntax

```
CALL neptune.algo.closenessCentrality(
  [node list (required)],
  {
    numSources: the number of BFS sources to use for computing the CC (required)
    edgeLabels: [a list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    traversalDirection: traversal direction (optional),
    normalize: Boolean, set to false to prevent normalization (optional)
    concurrency: number of threads to use (optional)
  }
)
YIELD node, score
RETURN node, score
```

Inputs for the `.closenessCentrality` algorithm

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none.*

  The node or nodes to use as the starting locations for the algorithm. Each node in the list triggers an execution of the algorithm. If an empty list is provided, the query result is also empty.

  If the algorithm is called following a `MATCH` clause (query algo integration), the source query list is the result returned by the `MATCH` clause.

- **a configuration object that contains:**
  - **numSources** *(required)* – *type:* `unsigned long`; *default: none*.

    The number of BFS sources for computing approximate Closeness Centrality (CC). To compute exact closeness centrality, set `numSources` to a number larger than number of nodes, such as `maxInt`.

    Because of the computational complexity of the algorithm for large graphs, it's generally best to specify a number in the order of thousands to ten thousands, such as 8,192.
  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.
  - **normalize** *(optional)* – *type:* `Boolean`; *default:* `true`.

    You can use this field to turn off normalization, which is on by default. Without normalization, only centrality scores of nodes within the same component can be meaningfully compared. Normalized scores can be compared across different connected components.

    The CC is normalizd using the Wasserman-Faust normalization formula for unconnected graphs. If there are n vertices reachable from vertex u (including vertex u itself), the Wasserman-Faust normalized closeness centrality score of vertex u is calculated as follows:

    ```
    (n-1)^2 / (|V| - 1) * sum(distance from u to these n vertices)
    ```

    Without normalization, the centrality score of vertex u is calculated as:

    ```
    (|V| - 1) / sum(distance from u to all other vertices in the graph)
    ```
  - **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

    A node label for node filtering. If a node label is provided, nodes matching the label are the only nodes that are included, including nodes in the input list.
  - **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

    The direction of edge to follow. Must be one of: `"inbound"`, `"oubound"`, or `"both"`.
  - **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

Controls the number of concurrent threads used to run the algorithm.

If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to 1, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `.closenessCentrality` algorithm

- **node** – A key column of the input nodes.
- **score** – A key column of the corresponding closeness-centrality (CC) scores for those nodes.

If the input node list is empty, the output is empty.

`.closenessCentrality` query examples

This is a standalone example, where the source node list is explicitly provided in the query:

```
CALL neptune.algo.closenessCentrality(
  ["101"],
  {
    numSources: 10,
    edgeLabels: ["route"],
    vertexLabel: "airport",
    traversalDirection: "outbound",
    normalize: true,
    concurrency: 1
  }
)
YIELD node, score
RETURN node, score
```

This is a query integration example, where `.closenessCentrality.mutate` follows a `MATCH` clause and uses the output of the `MATCH` clause as its list of source nodes:

```
Match (n)
CALL neptune.algo.closenessCentrality(
  n,
  {
    numSources: 10,
    edgeLabels: ["route"],
    vertexLabel: "airport",
    traversalDirection: "outbound",
    normalize: true,
    concurrency: 1
  }
)
YIELD score
RETURN n, score
```

This is a query integration examples that returns the nodes with the 10 highest CC scores:

```
CALL neptune.algo.closenessCentrality(
  n,
  {
    edgeLabels: ["route"],
    numSources: 10
  }
)
YIELD score
RETURN n, score
ORDER BY score DESC
LIMIT 10"
```

Warning

It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.closenessCentrality` output

Here is an example of the output returned by .closenessCentrality when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.closenessCentrality(n, {numSources: 10}) YIELD
 node, score RETURN node, score limit 2" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "node": {
        "~id": "10",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 38.94449997,
          "elev": 313,
          "longest": 11500,
          "city": "Washington D.C.",
          "type": "airport",
          "region": "US-VA",
          "desc": "Washington Dulles International Airport",
          "code": "IAD",
          "prscore": 0.002264724113047123,
          "degree": 312,
          "lon": -77.45580292,
          "wccid": 2357352929951779,
          "country": "US",
          "icao": "KIAD",
          "runways": 4
        }
      },
      "score": 0.20877772569656373
    },
    {
      "node": {
        "~id": "12",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 40.63980103,
          "elev": 12,
          "longest": 14511,
          "city": "New York",
          "type": "airport",
          "region": "US-NY",
          "desc": "New York John F. Kennedy International Airport",
          "code": "JFK",
          "prscore": 0.002885053399950266,
          "degree": 403,
          "lon": -73.77890015,
          "wccid": 2357352929951779,
          "country": "US",
          "icao": "KJFK",
          "runways": 4
        }
      },
      "score": 0.2199712097644806
    }
  ]
}
```

Closeness centrality mutatealgorithm

The closeness centrality mutate algorithm computes a Closeness Centrality (CC) metric for specified nodes in a graph. The CC metric of a node can be used as a positive measure of how close it is to all other nodes or how central it is in the graph.

The CC metric can be interpreted to show how quickly all other nodes in a network can be reached from a given node, and how important it is as a central hub for rapid information flow. It can be used in transportation networks to identify key hub locations, and in disease-spread modeling to pinpoint central points for targeted intervention efforts.

The closeness centrality (CC) score of a node is calculated based on the sum of its distances to all other vertices. The CC score itself is the inverse of that number; in other words, one divided by that sum. In practice, the calculation is commonly normalized to use the average length of the shortest paths rather than the actual sum of their lengths.

`.closenessCentrality.mutate` syntax

```
CALL neptune.algo.closenessCentrality.mutate(
  [node list (required)],
  {
    numSources: the number of BFS sources to use for computing the CC (required)
    writeProperty: name of the node property to write the CC score to (required)
    edgeLabels: [a list of edge labels for filtering (optional)],
    vertexLabel: "a node label for filtering (optional)",
    traversalDirection: traversal direction (optional),
    normalize: Boolean, set to false to prevent normalization (optional)
    concurrency: number of threads to use (optional)
  }
)
YIELD success
RETURN success
```

`.closenessCentrality.mutate` inputs

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes to use as the starting locations for the algorithm. Each node in the list triggers an execution of the algorithm. If an empty list is provided, the query result is also empty.

If the algorithm is called following a MATCH clause (query algo integration), the source node list is the result returned by the MATCH clause.

- **a configuration object that contains:**
  - **numSources** *(required)* – *type:* `uint64_t`; *default: none*.

    The number of BFS sources for computing approximate Closeness Centrality (CC). To compute exact closeness centrality, set `numSources` to a number larger than number of vertices, such as `maxInt`.

    Because of the computational complexity of the algorithm for large graphs, it's generally best to specify a number in the order of thousands to ten thousands, such as 8,192.
  - **writeProperty** *(required)* – *type:* `string`; *default: none*.

    A name for the new node property that will contain the computed CC score of each node.
  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.
  - **normalize** *(optional)* – *type:* `Boolean`; *default:* `true`.

    You can use this field to turn off normalization, which is on by default. Without normalization, only centrality scores of nodes within the same component can be meaningfully compared. Normalized scores can be compared across different connected components.

    The CC is normalizd using the Wasserman-Faust normalization formula for unconnected graphs. If there are n vertices reachable from vertex u (including vertex u itself), the Wasserman-Faust normalized closeness centrality score of vertex u is calculated as follows:

    ```
    (n-1)^2 / (|V| - 1) * sum(distance from u to these n vertices)
    ```

    Without normalization, the centrality score of vertex u is calculated as:

    ```
    (|V| - 1) / sum(distance from u to all other vertices in the graph)
    ```
  - **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

A node label for node filtering. If a node label is provided, vertices matching the label are the only vertices that are included, including vertices in the input list.

- **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

  The direction of edge to follow. Must be one of: `"inbound"`, `"oubound"`, or `"both"`.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

`.closenessCentrality.mutate` outputs

The closeness centrality score of each source node in the input list is written as a new node property using the property name specified in `writeProperty`.

If the algorithm is invoked as a standalone query, there is no other output.

If the algorithm is invoked following a `MATCH` clause that provides its source node list (query integration), the algorithm outputs a key column of the source vertices from the `MATCH` clause and a value column of Booleans (true or false) that indicate whether the CC value was successfully written to the node in question.

Query examples for `.closenessCentrality.mutate`

This example computes closeness centrality scores and writes them as a new node property called `ccScore`:

```
CALL neptune.algo.closenessCentrality.mutate(
  {
    numSources: 10,
    writeProperty: "ccScore",
    edgeLabels: ["route"],
    vertexLabel: "airport",
    traversalDirection: "outbound",
    normalize: true,
    concurrency: 1
  }
)
```

Then you can query the `ccScore` property in a subsequent query:

```
MATCH (n) RETURN id(n), n.ccScore limit 5
```

Sample `.closenessCentrality.mutate` output

Here is an example of the output returned by .closenessCentrality.mutate when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.closenessCentrality.mutate(
       {
         writeProperty: 'ccscore',
         numSources: 10
       }
     )
     YIELD success
     RETURN success"
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "success": true
    }
  ]
}
```

Similarity algorithms in Neptune Analytics

Graph similarity algorithms allow you to compare and analyze the similarities and dissimilarities between different graph structures, which can provide insight into relationships, patterns, and commonalities across diverse datasets. This is invaluable in various fields, such as biology, for comparing molecular structures, such as social networks, for identifying similar communities, and such as recommendation systems, for suggesting similar items based on user preferences.

Neptune Analytics supports the following similarity algorithms:

- `neighbors.common` – This algorithm counts the number of common neighbors of two input vertices, which is the intersection of the neighborhoods of those vertices.

  By counting how many neighboring nodes are shared by two nodes, it provides a measure of their potential interaction or similarity within the network. It's used in social network analysis to identify individuals with mutual connections, in citation networks to find influential papers referenced by multiple sources, and in transportation networks to locate critical hubs with many direct connections to other nodes.
- `neighbors.total` – This algorithm counts the number of total unique neighbors among two input vertices, which is the union of the neighborhoods of those vertices.
- `jaccardSimilarity` – This algorithm measures the similarity between two sets by dividing the size of their intersection by the size of their union.

  By measuring the proportion of shared neighbors relative to the total number of unique neighbors, it provides a metric for understanding the degree of overlap or commonality between different parts of a network. Jaccard similarity is applied in recommendation systems to suggest products or content to users based on their shared preferences and in biology to compare genetic sequences for identifying similarities in DNA fragments.
- `overlapSimilarity` – This algorithm measures the overlap between the neighbors of two vertices.

  It quantifies the similarity between nodes by calculating the ratio of common neighbors they share to the total number of neighbors they collectively have, providing a measure of their closeness or similarity within the network. Overlap similarity is applied in social network analysis to identify communities of individuals with shared interests or interactions, and in biological networks to detect common functionalities among proteins in molecular pathways.

Common neighbors algorithm

Common neighbors is an algorithm that counts the number of common neighbors of two input nodes, which is the intersection of their neighborhoods. This provides a measure of their potential interaction or similarity within the network. The common neighbors algorithm is used in social network analysis to identify individuals with mutual connections, in citation networks to find influential papers referenced by multiple sources, and in transportation networks to locate critical hubs with many direct connections to other nodes.

`.neighbors.common` syntax

```
CALL neptune.algo.neighbors.common(
  [first node(s)],
  [second node(s)],
  {
    edgeLabels: [a list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
  }
)
YIELD common
RETURN firstNodes, secondNodes, common
```

`.neighbors.common` inputs

- **first node(s)** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  One or more nodes of which to find the common neighbors with the corresponding second node(s).

- **second node(s)** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  One or more nodes of which to find the common neighbors with the corresponding first node(s).

- **a configuration object that contains:**
  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

  - **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

A node label for node filtering. If a node label is provided, nodes matching the label are the only nodes that are considered neighbors. This does not filter the nodes in the first or second node lists.

`.neighbors.common` outputs

common: A row for each node in the first node list and corresponding node in the second node list, and the number of neighboring nodes they have in common.

If either input node list is empty, the output is empty.

`.neighbors.common` query examples

This example specifies only two nodes:

```
MATCH (sydairport:airport {code: 'SYD'})
MATCH (jfkairport:airport {code: 'JFK'})
CALL neptune.algo.neighbors.common( sydairport, jfkairport, { edgeLabels: ['route'] })
YIELD common
RETURN sydairport, jfkairport, common
```

This example specifies multiple nodes. It returns a row for each combination of a US airport and a UK airport, and the number of destinations we could reach from both of those two airports:

```
MATCH (usairports:airport {country: 'US'})
MATCH (ukairports:airport {country: 'UK'})
CALL neptune.algo.neighbors.common(usairports, ukairports, {edgeLabels: ['route']})
YIELD common
RETURN usairports, ukairports, common
```

Warning

It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample .neighbors.common output

Here is an example of the output returned by .neighbors.common when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "MATCH (sydairport:airport {code: 'SYD'})
                        MATCH (jfkairport:airport {code: 'JFK'})
                        CALL neptune.algo.neighbors.common(sydairport, jfkairport,
 {edgeLabels: ['route']})
                        YIELD common
                        RETURN sydairport, jfkairport, common" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "sydairport": {
        "~id": "55",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": -33.9460983276367,
          "elev": 21,
          "type": "airport",
          "code": "SYD",
          "lon": 151.177001953125,
          "runways": 3,
          "longest": 12999,
          "communityId": 2357352929951971,
          "city": "Sydney",
          "region": "AU-NSW",
          "desc": "Sydney Kingsford Smith",
          "prscore": 0.0028037719894200565,
          "degree": 206,
          "wccid": 2357352929951779,
          "ccscore": 0.19631840288639069,
          "country": "AU",
          "icao": "YSSY"
        }
      },
      "jfkairport": {
        "~id": "12",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 40.63980103,
          "elev": 12,
          "type": "airport",
          "code": "JFK",
          "lon": -73.77890015,
          "runways": 4,
          "longest": 14511,
          "communityId": 2357352929951971,
          "city": "New York",
          "region": "US-NY",
          "desc": "New York John F. Kennedy International Airport",
          "prscore": 0.002885053399950266,
          "degree": 403,
          "wccid": 2357352929951779,
          "ccscore": 0.2199712097644806,
          "country": "US",
          "icao": "KJFK"
        }
      },
      "common": 24
    }
  ]
}
```

Total neighbors algorithm

Total neighbors is an algorithm that counts the total number of unique neighbors of two input vertices, which is the union of the neighborhoods of those vertices.

`.neighbors.total` syntax

```
CALL neptune.algo.neighbors.total(
  [first node(s)],
  [second node(s)],
  {
    edgeLabels: [a list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
  }
)
YIELD total
RETURN firstNodes, secondNodes, total
```

Inputs for the `.neighbors.total` algorithm

- **first node(s)** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  One or more nodes of which to find the common neighbors with the corresponding second nodes.

- **second node(s)** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  One or more nodes of which to find the common neighbors with the corresponding first nodes.

- **a configuration object that contains:**

  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

  - **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

    A node label for node filtering. If a node label is provided, nodes matching the label are the only nodes that are considered neighbors. This does not filter the nodes in the first or second node lists.

`.neighbors.total` outputs

total: A row for each node in the first node list and corresponding node in the second node list, and the total number of neighboring nodes they have.

If either input node list is empty, the output is empty.

`.neighbors.total` query examples

This example returns a row for each combination of a US airport and a UK airport, and the total number of destinations we could reach if we could fly out of either of the two airports.

```
MATCH (usairports:airport {country: 'US'})
MATCH (ukairports:airport {country: 'UK'})
CALL neptune.algo.neighbors.total(usairports, ukairports, {edgeLabels: ['route']})
YIELD total
RETURN usairports, ukairports, total"
```

> ⚠ **Warning**
>
> It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.neighbors.total` output

Here is an example of the output returned by .neighbors.total when run against the [sample air-routes dataset [nodes]](), and [sample air-routes dataset [edges]](), when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "MATCH (sydairport:airport {code: 'SYD'})
                       MATCH (jfkairport:airport {code: 'JFK'})
                       CALL neptune.algo.neighbors.total(sydairport, jfkairport,
 {edgeLabels: ['route']})
                       YIELD total
                       RETURN sydairport, jfkairport, total"
  --language open_cypher \
  /tmp/out.txt
```

```
cat /tmp/out.txt
{
  "results": [
    {
      "sydairport": {
        "~id": "55",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": -33.9460983276367,
          "elev": 21,
          "type": "airport",
          "code": "SYD",
          "lon": 151.177001953125,
          "runways": 3,
          "longest": 12999,
          "communityId": 2357352929951971,
          "city": "Sydney",
          "region": "AU-NSW",
          "desc": "Sydney Kingsford Smith",
          "prscore": 0.0028037719894200565,
          "degree": 206,
          "wccid": 2357352929951779,
          "ccscore": 0.19631840288639069,
          "country": "AU",
          "icao": "YSSY"
        }
      },
      "jfkairport": {
        "~id": "12",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 40.63980103,
          "elev": 12,
          "type": "airport",
          "code": "JFK",
          "lon": -73.77890015,
          "runways": 4,
          "longest": 14511,
          "communityId": 2357352929951971,
          "city": "New York",
          "region": "US-NY",
          "desc": "New York John F. Kennedy International Airport",
          "prscore": 0.002885053399950266,
          "degree": 403,
          "wccid": 2357352929951779,
          "ccscore": 0.2199712097644806,
          "country": "US",
          "icao": "KJFK"
        }
      },
      "total": 279
    }
  ]
}
```

Jaccard similarity algorithm

The Jaccard similarity algorithm measures the similarity between two sets. It is calculated by dividing the size of the intersection of the two sets by the size of their union.

By measuring the proportion of shared neighbors relative to the total number of unique neighbors, this algorithm provides a metric for the degree of overlap or commonality between different parts of a network. It can be useful in recommendation systems to suggest products or content to users based on their shared preferences and in biology to compare genetic sequences for identifying similarities in DNA fragments.

`.jaccardSimilarity` syntax

```
CALL neptune.algo.jaccardSimilarity(
  [first node(s)],
  [second node(s)],
  {
    edgeLabels: [a list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
  }
)
YIELD score
RETURN firstNodes, secondNodes, score
```

`.jaccardSimilarity` inputs

- **first node(s)** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  One or more nodes for which to find the Jaccard similarity score with respect to the corresponding second node(s).

- **second node(s)** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  One or more nodes for which to find the Jaccard similarity score with respect to the corresponding first node(s).

- **a configuration object that contains:**

  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

  A node label for node filtering. If a node label is provided, nodes matching the label are the only nodes that are considered neighbors. This does not filter the nodes in the first or second node lists.

Outputs for the `.jaccardSimilarity` algorithm

score: A row for each node in the first node list and corresponding node in the second node list, and the Jaccard similarity score for the two.

If either input node list is empty, the output is empty.

`.jaccardSimilarity` query examples

The example below is a query integration examples, where the node list inputs for `.jaccardSimilarity` come from a preceding `MATCH` clause:

```
MATCH (n1:Person {name: "Alice"}), (n2:Person {name: "Bob"})
CALL neptune.algo.jaccardSimilarity(n1, n2, {edgeLabels: ['knows']})
YIELD score
RETURN n1, n2, score
```

Another example:

```
MATCH (n {code: "AUS"})
MATCH (m {code: "FLL"})
CALL neptune.algo.jaccardSimilarity(
  n,
  m,
  {
    edgeLabels: ["route"],
    vertexLabel: "airport"
  }
)
YIELD score
RETURN n, m, score
```

Warning

It is not good practice to use MATCH(n) without restriction in query integrations. Keep in mind that every node returned by the MATCH(n) clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use LIMIT or put conditions on the MATCH clause to restrict its output appropriately.

Sample .jaccardSimilarity output

Here is an example of the output returned by .jaccardSimilarity when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "MATCH (n {code: 'AUS'})
                       MATCH (m {code: "FLL"})
                       CALL neptune.algo.jaccardSimilarity(n, m,
                           {edgeLabels: [\"route\"], vertexLabel: \"airport\"})
                       YIELD score
                       RETURN n, m, score"
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "n": {
        "~id": "3",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 30.1944999694824,
          "elev": 542,
          "type": "airport",
          "code": "AUS",
          "lon": -97.6698989868164,
          "runways": 2,
          "longest": 12250,
          "communityId": 2357352929951971,
          "city": "Austin",
          "region": "US-TX",
          "desc": "Austin Bergstrom International Airport",
          "prscore": 0.0012390684569254518,
          "degree": 188,
          "wccid": 2357352929951779,
          "ccscore": 0.1833982616662979,
          "country": "US",
          "icao": "KAUS"
        }
      },
      "m": {
        "~id": "9",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 26.0725994110107,
          "elev": 64,
          "type": "airport",
          "code": "FLL",
          "lon": -80.152702331543,
          "runways": 2,
          "longest": 9000,
          "communityId": 2357352929951971,
          "city": "Fort Lauderdale",
          "region": "US-FL",
          "desc": "Fort Lauderdale/Hollywood International Airport",
          "prscore": 0.0024497462436556818,
          "degree": 316,
          "wccid": 2357352929951779,
          "ccscore": 0.19741515815258027,
          "country": "US",
          "icao": "KFLL"
        }
      },
      "score": 0.2953367829322815
    }
  ]
}
```

Overlap similarity algorithm

Overlap Similarity is an algorithm that measures the overlap between the neighbors of two nodes. It does this by dividing the intersection of the two neighborhoods by the neighbor with minimum degree.

By calculating the ratio of common neighbors shared by two nodes to the total number of neighbors they collectively have, it provides a measure of their closeness or similarity within the network. Overlap similarity is applied in social network analysis to identify communities of individuals with shared interests or interactions, and in biological networks to detect common functionalities among proteins in molecular pathways.

`.overlapSimilarity` syntax

```
CALL neptune.algo.overlapSimilarity(
  [first node(s)],
  [second node(s)],
  {
    edgeLabels: [a list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
  }
)
YIELD score
RETURN firstNodes, secondNodes, score
```

`.overlapSimilarity` inputs

- **first node(s)** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  One or more nodes for which to find the overlap similarity score with respect to the corresponding second node(s).

- **second node(s)** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  One or more nodes for which to find the overlap similarity score with respect to the corresponding first node(s).

- **a configuration object that contains:**

  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

  A node label for node filtering. If a node label is provided, nodes matching the label are the only nodes that are considered neighbors. This does not filter the nodes in the first or second node lists.

`.overlapSimilarity` outputs

score: A row for each node in the first node list and corresponding node in the second node list, and the overlap similarity score for the two.

If either input node list is empty, the output is empty.

`.overlapSimilarity` query examples

This is a query integration examples, where `.overlapSimilarity` takes its input node lists from the output of a `MATCH` clause:

```
MATCH (n1:Person {name: "Alice"}), (n2:Person {name: "Bob"})
CALL neptune.algo.overlapSimilarity(n1, n2, {edgeLabel: 'knows'})
YIELD score
RETURN n1, n2, score
```

Another example:

```
MATCH (n {code: "AUS"})
MATCH (m {code: "FLL"})
CALL neptune.algo.overlapSimilarity(
  n,
  m,
  {
    edgeLabels: ["route"],
    vertexLabel: "airport"
  }
)
YIELD score
RETURN n, m, score'
```

Warning

It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.overlapSimilarity` output

Here is an example of the output returned by .overlapSimilarity when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string 'MATCH (n {code: "AUS"})
                       MATCH (m {code: "FLL"})
                       CALL neptune.algo.overlapSimilarity(
                         n,
                         m,
                         {
                           edgeLabels: ["route"],
                           vertexLabel: "airport"
                         }
                       )
                       YIELD score
                       RETURN n, m, score' \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "n": {
        "~id": "3",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 30.1944999694824,
          "elev": 542,
          "type": "airport",
          "code": "AUS",
          "lon": -97.6698989868164,
          "runways": 2,
          "longest": 12250,
          "communityId": 2357352929951971,
          "city": "Austin",
          "region": "US-TX",
          "desc": "Austin Bergstrom International Airport",
          "prscore": 0.0012390684569254518,
          "degree": 188,
          "wccid": 2357352929951779,
          "ccscore": 0.1833982616662979,
          "country": "US",
          "icao": "KAUS"
        }
      },
      "m": {
        "~id": "9",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 26.0725994110107,
          "elev": 64,
          "type": "airport",
          "code": "FLL",
          "lon": -80.152702331543,
          "runways": 2,
          "longest": 9000,
          "communityId": 2357352929951971,
          "city": "Fort Lauderdale",
          "region": "US-FL",
          "desc": "Fort Lauderdale/Hollywood International Airport",
          "prscore": 0.0024497462436556818,
          "degree": 316,
          "wccid": 2357352929951779,
          "ccscore": 0.19741515815258027,
          "country": "US",
          "icao": "KFLL"
        }
      },
      "score": 0.6129032373428345
    }
  ]
```

```
}
```

Clustering and community detection algorithms in Neptune Analytics

Clustering algorithms evaluate how nodes are clustered in communities, in closely-knit sets, or in highly or loosely interconnected groups.

These algorithms can identify meaningful groups or clusters of nodes in a network, revealing hidden patterns and structures that can provide insights into the organization and dynamics of complex systems. This is valuable in social network analysis and in biology, for identifying functional modules in protein-protein interaction networks, and more generally for understanding information flow and influence propagation in many different domains.

Neptune Analytics supports these community detection algorithms:

- `wcc` – The Weakly Connected Components (WCC) algorithm finds weakly-connected components in a directed graph. A weakly-connected component is a group of nodes where every node in the group is reachable from every other node in the group if edge direction is ignored.

  Identifying weakly-conected components helps in understanding the overall connectivity and structure of the graph. Weakly-connected components can be used in transportation networks to identify disconnected regions that may require improved connectivity, and in social networks to find isolated groups of users with limited interactions, and in webpage analysis to pinpoint sections with low accessibility.
- `wcc.mutate` – This algorithm stores the calculated component value of each given node as a property of the node.
- `labelPropagation` – Label Propagation Algorithm (LPA) is an algorithm for community detection that is also used in semi-supervised machine learning for data classification.
- `labelPropagation.mutate` – Label Propagation Algorithm (LPA) is an algorithm tha assigns labels to nodes based on the consensus of their neighboring nodes, making it useful for identifying groups. Label propagation can be applied in social networks to find groups, and in identity management to identify households, and in recommendation systems to group similar products for personalized suggestions. It can also be used in semi-supervised machine learning for data classification.
- `scc` – The Strongly Connected Components (SCC) algorithm identifies maximally connected subgraphs of a directed graph, where every node is reachable from every other node. This can provide insights into the tightly interconnected portions of a graph and highlight key structures

within it. Strongly connected components are valuable in computer programming for detecting loops or cycles in code, in social networks to find tightly connected groups of users who interact frequently, and in web crawling to identify clusters of interlinked pages for efficient indexing.

- `scc.mutate` – This algorithm finds the maximally connected subgraphs of a directed graph and writes their component IDs as a new property of each subgraph node.

Weakly connected components algorithm

The Weakly Connected Components (WCC) algorithm finds the weakly-connected components in a directed graph. A weakly-connected component is a group of nodes in which every node is reachable from every other node when edge directions are ignored. Weakly connected components are the maximal connected subgraphs of an undirected graph.

Identifying weakly-conected components helps in understanding the overall connectivity and structure of the graph. Weakly-connected components can be used in transportation networks to identify disconnected regions that may require improved connectivity, and in social networks to find isolated groups of users with limited interactions, and in webpage analysis to pinpoint sections with low accessibility.

The time complexity of the WCC algorithm is `O(|E|logD)`, where `|E|` is the number of edges in the graph, and `D` is the diameter (the length of the longest path from one node to any other node) of the graph.

The memory used by the WCC algorithm is approximately |V| * 20 bytes.

`.wcc` syntax

```
CALL neptune.algo.wcc(
  [source-node list (required)],
  {
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD node, component
RETURN node, component
```

`.wcc` inputs

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes to use as the starting locations for the algorithm. Each node in the list triggers an execution of the algorithm. If an empty list is provided, the query result is also empty.

  If the algorithm is called following a `MATCH` clause (query algo integration), the source query list is the result returned by the `MATCH` clause.

- **a configuration object that contains:**
  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.
  - **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

    A node label for node filtering. If a node label is provided, only nodes matching the label are considered. This includes the nodes in the source node lists.
  - **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

    Controls the number of concurrent threads used to run the algorithm.

    If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

`.wcc` outputs

For each source node:

- **node** – The source node.
- **component** – The component ID associated with the source node.

If the input node list is empty, the output is empty.

`.wcc` query examples

This is a standalone example, where the source node list is explicitly provided in the query:

```
CALL neptune.algo.wcc(
  ["101"],
  {
    edgeLabels: ["route"],
    vertexLabel: "airport",
    concurrency: 2
  }
)
```

```
YIELD node, component
RETURN node, component
```

This is a query integration examples, where `.wcc` follows a `MATCH` clause and uses the output of the MATCH clause as its source node list:

```
MATCH (n) WHERE n.region = 'US-WA'
CALL neptune.algo.wcc(
  n,
  {
    edgeLabels: ["route"],
    vertexLabel: "airport"
  }
)
YIELD component
RETURN n, component
```

Warning

It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.wcc` output

Here is an example of the output returned by .wcc when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "MATCH (n)
                       CALL neptune.algo.wcc(n)
                       YIELD node, component
                       RETURN node, component
                       LIMIT 2"
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
```

```
{
  "results": [
    {
      "node": {
        "~id": "10",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 38.94449997,
          "elev": 313,
          "longest": 11500,
          "city": "Washington D.C.",
          "type": "airport",
          "region": "US-VA",
          "desc": "Washington Dulles International Airport",
          "code": "IAD",
          "prscore": 0.002264724113047123,
          "lon": -77.45580292,
          "wccid": 2357352929951779,
          "country": "US",
          "icao": "KIAD",
          "runways": 4
        }
      },
      "component": 2357352929951779
    },
    {
      "node": {
        "~id": "12",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 40.63980103,
          "elev": 12,
          "longest": 14511,
          "city": "New York",
          "type": "airport",
          "region": "US-NY",
          "desc": "New York John F. Kennedy International Airport",
          "code": "JFK",
          "prscore": 0.002885053399950266,
          "lon": -73.77890015,
          "wccid": 2357352929951779,
          "country": "US",
     "icao": "KJFK",
     "runways": 4
    }
   },
   "component": 2357352929951779
  }
 ]
}
```

Weakly connected components mutate algorithm

The mutate variant of the weakly connected components (WCC) algorithm performs the weakly connected components calculation over the entire graph unless the configuration parameters establish a filter, and each traversed node's calculated WCC value is stored as a property on the node.

`.wcc.mutate` syntax

```
CALL neptune.algo.wcc.mutate(
  {
    writeProperty: the name for the node property to which to write component IDs
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD success
RETURN success
```

`.wcc.mutate` inputs

Inputs for `.wcc.mutate` are passed in a configuration object that contains:

- **`writeProperty`** *(required)* – *type:* `string`; *default: none*.

  A name for the new node property where the component IDs will be written.

- **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

  To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **`vertexLabel`** *(optional)* – *type:* `string`; *default: none*.

  The node label to filter on for traversing. Only nodes matching this label will be traversed. For example: `"airport"`.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to 1, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

`.wcc.mutate` outputs

success: The computed component IDs are written as a new property on each node using the property name specified by `writeProperty`, and a single `success` flag (`true` or `false`) is returned to indicate whether or not the writes succeeded.

`.wcc.mutate` query examples

This query writes the calculated component ID of each vertex in the graph to a new property of the vertex named `CCID`:

```
CALL neptune.algo.wcc.mutate(
  {
    writeProperty: "CCID",
    edgeLabels: ["route"],
    vertexLabel: "airport",
    concurrency: 2
  }
)
```

After the mutate algorithm call above, the following query can retrieve the CCID property of a specific node:

```
MATCH (n: airport {code: "SEA"})
RETURN n.CCID
```

Sample `.wcc.mutate` output

Here is an example of the output returned by .wcc.mutate when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.wcc.mutate({writeProperty: 'wccid'}) YIELD success
 RETURN success"
```

```
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "success": true
    }]
}
```

Label propagation algorithm (LPA)

Label Propagation Algorithm (LPA) is an algorithm for community detection that is also used in semi-supervised machine learning for data classification.

A community structure is loosely defined as a tightly knit group of entities in social networks. LPA can be enhanced by providing a set of seed nodes, the quality of which can dramatically influence the solution quality of the found communities. If the seeds are well-selected, the quality of the solution can be good, but if not, the quality of the solution can be very bad.

See Xu T. Liu *et al*, Direction-optimizing label propagation and its application to community detection, and Xu T. Liu *et al*, Direction-optimizing Label Propagation Framework for Structure Detection in Graphs: Design, Implementation, and Experimental Analysis, and the Neo4j Label Propagation API.

The time complexity of the algorithm is `O(k|E|)`, where `|E|` is the number of edges in the graph, and `k` is the number of iterations for the algorithm to converge. Its space complexity is `O(|V|)`, where `|V|` is the number of nodes in the graph.

`.labelPropagation` syntax

```
CALL neptune.algo.labelPropagation(
  [source-node list (required)],
  {
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    vertexWeightProperty: a numeric node property used to weight the community ID
 (optional),
    vertexWeightType: numeric type of the specified vertexWeightProperty (optional),
    edgeWeightProperty: a numeric edge property used to weight the community ID
 (optional),
    edgeWeightType: numeric type of the specified edgeWeightProperty (optional),
    maxIterations: the maximum number of iterations to run (optional, default: 10),
    traversalDirection: traversal direction (optional, default: outbound),
    concurrency: number of threads to use (optional)
  }
)
Yield node, community
Return node, community
```

.labelPropagation inputs

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The node or nodes to use as the starting locations for the algorithm. Each node in the list triggers an execution of the algorithm. If an empty list is provided, the query result is also empty.

  If the algorithm is called following a `MATCH` clause (query algo integration), the source query list is the result returned by the `MATCH` clause.

- **a configuration object that contains:**
  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

  - **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

    A node label for node filtering. If a node label is provided, nodes matching the label are the only nodes that are included in the calculation, including nodes in the input list.

  - **vertexWeightProperty** *(optional)* – *type:* `string`; *default: none*.

    The node weight used in Label Propagation. When `vertexWeightProperty` is not specified, each node's `communityId` is treated equally, as if the node weight were 1.0. When the `vertexWeightProperty` is specified without an `edgeWeightProperty`, the weight of the `communityId` for each node is the value of the node weight property. When both `vertexWeightProperty` and `edgeWeightProperty` are specified, the weight of the `communityId` is the product of the node property value and edge property value.

    Note that if multiple properties exist on the node with the name specified by `vertexWeightProperty`, one of those property values will be sampled at random.

  - **vertexWeightType** *(required if `vertexWeightProperty` is present)* – *type:* `string`; *valid values:* `"int"`, `"long"`, `"float"`, `"double"`; *default: empty*.

    The type of the numeric values in the node property specified by `vertexWeightProperty`.

    If `vertexWeightProperty` is not provided, `vertexWeightType` is ignored. If a node contains a numeric property with the name specified by `vertexWeightProperty` but its value is a different numeric type than is specified by `vertexWeightType`, the value is typecast to the type specified by `vertexWeightType`. If both `vertexWeightType` and `edgeWeightType` are given, the type specified by `edgeWeightType` is used for both node and edge properties.

- **edgeWeightProperty** *(optional)* – *type:* `string`; *default: none*.

  The numeric edge property used as a weight in Label Propagation. When `vertexWeightProperty` is not specified, the default edge weight is 1.0, so each edge is treated equally. When only `edgeWeightProperty` is provided, the weight of the communityId is the value of that edge property. When both `vertexWeightProperty` and `edgeWeightProperty` are present, the weight of a `communityId` is the product of the edge property value and the node property value.

  Note that if multiple properties exist on the edge with the name specified by `edgeWeightProperty`, one of those property values will be sampled at random.

- **edgeWeightType** *(required if `edgeWeightProperty` is present)* – *type:* `string`; *valid values:* `"int"`, `"unsignedInt"`, `"long"`, `"unsignedLong"`, `"float"`, `"double"`; *default: none*.

  The type of the numeric values in the edge property specified by `edgeWeightProperty`.

  If `edgeWeightProperty` is not provided, `edgeWeightType` is ignored. If a node contains a numeric property with the name specified by `edgeWeightProperty` but its value is a different numeric type than is specified by `edgeWeightType`, the value is typecast to the type specified by `edgeWeightType`. If both `vertexWeightType` and `edgeWeightType` are given, the type specified by `edgeWeightType` is used for both node and edge properties.

- **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

  The direction of edge to follow. Must be one of: `"inbound"`, `"oubound"`, or `"both"`.

- **maxIterations** *(optional)* – *type:* integer; *default:* `10`.

  The maximum number of iterations to run.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

.labelPropagation outputs

- **node** – A key column of the input nodes.
- **community** – A key column of the corresponding `communityId` values for those nodes. All the nodes with the same `communityId` are in the same weakly-connected component.

If the input node list is empty, the output is empty.

.labelPropagation query examples

This is a standalone example, where the source node list is explicitly provided in the query. It runs the algorithm over the whole graph, but only queries the component ID of one node:

```
CALL neptune.algo.labelPropagation(
  ["101"],
  {
    edgeLabels: ["route"],
    maxIterations: 10,
    vertexLabel: "airport",
    vertexWeightProperty: "runways",
    vertexWeightType: "int",
    edgeWeightProperty: "dist",
    edgeWeightType: "int",
    traversalDirection: "both",
    concurrency: 2
  }
)
YIELD node, community
RETURN node, community
```

This is a query integration example, where `.labelPropagation` uses the output of a preceding `MATCH` clause as its source node list:

```
Match (n)
CALL neptune.algo.labelPropagation(
  n,
  {
    edgeLabels: ["route"],
    maxIterations: 10,
    vertexLabel: "airport",
    vertexWeightProperty: "runways",
    vertexWeightType: "int",
    edgeWeightProperty: "dist",
    edgeWeightType: "int",
    traversalDirection: "both",
    concurrency: 2
  }
)
YIELD community
RETURN n, community
```

Warning

It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.labelPropagation` output

Here is an example of the output returned by .labelPropagation when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
"results": [{
      "node": {
        "~id": "8",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "region": "US-TX",
          "runways": 7,
          "country": "US",
          "city": "Dallas",
          "type": "airport",
          "icao": "KDFW",
          "lon": -97.038002014160199,
          "code": "DFW",
          "lat": 32.896800994872997,
          "longest": 13401,
          "elev": 607,
          "desc": "Dallas/Fort Worth International Airport"
        }
      },
      "community": 2357352929952311
    }, {
      "node": {
        "~id": "24",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "region": "US-CA",
          "runways": 3,
          "country": "US",
          "city": "San Jose",
          "type": "airport",
          "icao": "KSJC",
          "lon": -121.929000854492,
          "code": "SJC",
          "lat": 37.362598419189503,
          "longest": 11000,
          "elev": 62,
          "desc": "Norman Y. Mineta San Jose International Airport"
        }
      },
      "community": 2357352929952311
    }]
```

Label propagation mutate algorithm

Label Propagation Algorithm (LPA) is an algorithm for community detection that is also used in semi-supervised machine learning for data classification.

The `.labelPropagation.mutate` variant of the algorithm writes the derived community component ID of each node in the source list to a new property of that node.

`.labelPropagation.mutate` syntax

```
CALL neptune.algo.labelPropagation(
  {
    writeProperty: the name for the node property to which to write component IDs
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    vertexWeightProperty: a numeric node property used to weight the community ID
 (optional),
    vertexWeightType: numeric type of the specified vertexWeightProperty (optional),
    edgeWeightProperty: a numeric edge property used to weight the community ID
 (optional),
    edgeWeightType: numeric type of the specified edgeWeightProperty (optional),
    maxIterations: the maximum number of iterations to run (optional, default: 10),
    traversalDirection: traversal direction (optional, default: outbound),
    concurrency: number of threads to use (optional)
  }
)
```

`.labelPropagation.mutate` inputs

Inputs for `.wcc.mutate` are passed in a configuration object that contains:

- **`writeProperty`** *(required)* – *type:* `string`; *default: none*.

  A name for the new node property that will contain the computed community component ID of the node.

- **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

  To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **vertexWeightProperty** *(optional)* – *type:* `string`; *default: none*.

The node weight used in Label Propagation. When `vertexWeightProperty` is not specified, each node's `communityId` is treated equally, as if the node weight were 1.0. When the `vertexWeightProperty` is specified without an `edgeWeightProperty`, the weight of the `communityId` for each node is the value of the node weight property. When both `vertexWeightProperty` and `edgeWeightProperty` are specified, the weight of the `communityId` is the product of the node property value and edge property value.

Note that if multiple properties exist on the node with the name specified by `vertexWeightProperty`, one of those property values will be sampled at random.

- **vertexWeightType** *(required if `vertexWeightProperty` is presnt)* – *type:* `string`; *valid values:* `"int"`, `"long"`, `"float"`, `"double"`; *default: empty*.

  The type of the numeric values in the node property specified by `vertexWeightProperty`.

  If `vertexWeightProperty` is not provided, `vertexWeightType` is ignored. If a node contains a numeric property with the name specified by `vertexWeightProperty` but its value is a different numeric type than is specified by `vertexWeightType`, the value is typecast to the type specified by `vertexWeightType`. If both `vertexWeightType` and `edgeWeightType` are given, the type specified by `edgeWeightType` is used for both node and edge properties.

- **edgeWeightProperty** *(optional)* – *type:* `string`; *default: none*.

  The numeric edge property used as a weight in Label Propagation. When `vertexWeightProperty` is not specified, the default edge weight is 1.0, so each edge is treated equally. When only `edgeWeightProperty` is provided, the weight of the communityId is the value of that edge property. When both `vertexWeightProperty` and `edgeWeightProperty` are present, the weight of a `communityId` is the product of the edge property value and the node property value.

  Note that if multiple properties exist on the edge with the name specified by `edgeWeightProperty`, one of those property values will be sampled at random.

- **edgeWeightType** *(required if `edgeWeightProperty` is presnt)* – *type:* `string`; *valid values:* `"int"`, `"unsignedInt"`, `"long"`, `"unsignedLong"`, `"float"`, `"double"`; *default: none*.

  The type of the numeric values in the edge property specified by `edgeWeightProperty`.

  If `edgeWeightProperty` is not provided, `edgeWeightType` is ignored. If a node contains a numeric property with the name specified by `edgeWeightProperty` but its value is a different numeric type than is specified by `edgeWeightType`, the value is typecast to the type specified by `edgeWeightType`. If both `vertexWeightType` and `edgeWeightType` are given, the type specified by `edgeWeightType` is used for both node and edge properties.

- **traversalDirection** *(optional)* – *type:* `string`; *default:* `"outbound"`.

  The direction of edge to follow. Must be one of: `"inbound"`, `"oubound"`, or `"both"`.

- **maxIterations** *(optional)* – *type:* integer; *default:* `10`.

  The maximum number of iterations to run.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `.labelPropagation.mutate` algorithm

The community component IDs are written as a new node property of each source node using the property name specified by `writeProperty`.

If the algorithm is invoked as a standalone query, there is no other output.

If the algorithm is invoked immediately after a `MATCH` clause that supplies its source node list, the algorithm outputs a key column of the source nodes from the `MATCH` clause and a value column of success flags (true or false) to indicate whether or not the write to the new node property of that node succeeded.

`.labelPropagation.mutate` query example

```
CALL neptune.algo.labelPropagation.mutate(
  {
    writeProperty: "COMM_ID",
    edgeLabels: ["route"],
    maxIterations: 10,
    vertexLabel: "airport",
    vertexWeightProperty: "runways",
    vertexWeightType: "int",
    edgeWeightProperty: "dist",
    edgeWeightType: "int",
    traversalDirection: "both",
    concurrency: 2
  }
)
```

Sample .labelPropagation.mutate output

Here is an example of the output returned by .labelPropagation.mutate when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.labelPropagation.mutate({writeProperty:
 'communityId'}) YIELD success RETURN success" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "success": true
    }
  ]
}
```

Strongly connected components algorithm

Strongly connected components (SCC) are the maximally connected subgraphs of a directed graph where every node is reachable from every other node (in other words, there exists a path between every node in the subgraph).

Neptune Analytics implements this algorithm using a modified multi-step approach (see BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems, by George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri, IPDPS 2014).

The time complexity of the `.scc` algorithm in the worst case is `O(|V|+|E|*D)`, where `|V|` is the number of nodes in the graph, `|E|` is the number of edges in the graph, and `D` is the diameter, defined as the length of the longest path from one node to another in the graph.

The algorith's space complexity is `O(|V|) + [storage of data O(|V|+|E|)]`. This can be expressed in more detail as: `|V| * (7 * sizeof(local id type) + 4 * sizeof(global id type)) + O(|1DData Parts|)`.

`.scc` syntax

```
CALL neptune.algo.scc(
  [source-node list (required)],
  {
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD node, component
RETURN node, component
```

`.scc` inputs

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.
- **a configuration object that contains:**
  - **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

    To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

  A node label to filter on.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

`.scc` outputs

For each source node:

- **node** – The source node.
- **component** – The component ID associated with the source node.

If the input node list is empty, the output is empty.

`.scc` query examples

This openCypher query has an empty input list, and so will have no output:

```
Match (n)
CALL neptune.algo.scc(n, {edgeLabels: ["route", "contains"]})
YIELD component
RETURN n, component
```

This is a query integration example, where `.scc` follows a `MATCH` clause that generates its input node list:

```
Match (n)
CALL neptune.algo.scc(n, {})
Yield component
Return n, component
```

Warning

> It is not good practice to use `MATCH(n)` without restriction in query integrations. Keep in mind that every node returned by the `MATCH(n)` clause invokes the algorithm once, which can result a very long-running query if a large number of nodes is returned. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.scc` output

Here is an example of the output returned by .scc when run against the [sample air-routes dataset [nodes]](), and [sample air-routes dataset [edges]](), when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.scc({writeProperty: 'sccid'}) YIELD success RETURN
 success" \
  --language open_cypher \
  /tmp/out.txt

cat /tmp/out.txt
{
  "results": [
    {
      "node": {
        "~id": "10",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 38.94449997,
          "elev": 313,
          "type": "airport",
          "code": "IAD",
          "lon": -77.45580292,
          "runways": 4,
          "longest": 11500,
          "communityId": 2357352929951971,
          "city": "Washington D.C.",
          "region": "US-VA",
          "desc": "Washington Dulles International Airport",
          "prscore": 0.002264724113047123,
          "degree": 312,
          "wccid": 2357352929951779,
          "ccscore": 0.20877772569656373,
          "country": "US",
          "icao": "KIAD"
        }
      },
      "component": 2357352929966149
    },
    {
      "node": {
        "~id": "12",
        "~entityType": "node",
        "~labels": ["airport"],
        "~properties": {
          "lat": 40.63980103,
          "elev": 12,
          "type": "airport",
          "code": "JFK",
          "lon": -73.77890015,
          "runways": 4,
          "longest": 14511,
          "communityId": 2357352929951971,
          "city": "New York",
          "region": "US-NY",
          "desc": "New York John F. Kennedy International Airport",
          "prscore": 0.002885053399950266,
          "degree": 403,
          "wccid": 2357352929951779,
          "ccscore": 0.2199712097644806,
          "country": "US",
          "icao": "KJFK"
        }
      },
      "component": 2357352929966149
    }
  ]
}
```

Strongly connected components mutate algorithm

Strongly connected components (SCC) are the maximally connected subgraphs of a directed graph, where every node is reachable from every other node (in other words, there exists a path between every node in the subgraph).

The time complexity of the `.scc-mutate` algorithm in the worst case is `O(|V|+|E|*D)`, where `|V|` is the number of nodes in the graph, `|E|` is the number of edges in the graph, and `D` is the diameter, the length of the longest path from one node to another in the graph.

The algorith's space complexity is `O(|V|) + [storage of data O(|V|+|E|)]`. This can be expressed in more detail as: `|V| * (7 * sizeof(local id type) + 4 * sizeof(global id type)) + O(|1DData Parts|)`.

`.scc.mutate` syntax

```
CALL neptune.algo.scc.mutate(
  {
    writeProperty: the name for the node property to which to write component IDs
    edgeLabels: [list of edge labels for filtering (optional)],
    vertexLabel: a node label for filtering (optional),
    concurrency: number of threads to use (optional)
  }
)
YIELD success
RETURN success
```

Inputs for the `.scc.mutate` algorithm

Inputs for `.scc.mutate` are passed in a configuration object that contains:

- **writeProperty** *(required)* – *type:* `string`; *default: none*.

  A name for the new node property where the component IDs will be written.

- **edgeLabels** *(optional)* – *type:* a list of edge label strings; *example:* `["route", ...]`; *default:* no edge filtering.

  To filter on one more edge labels, provide a list of the ones to filter on. If no `edgeLabels` field is provided then all edge labels are processed during traversal.

- **vertexLabel** *(optional)* – *type:* `string`; *default: none*.

The node label to filter on for traversing. Only nodes matching this label will be traversed. For example: `"airport"`.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to 1, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

Outputs for the `.scc.mutate` algorithm

The computed strongly connected component IDs are written as a new node property using the specified property name. A single success flag (true or false) is returned to indicate whether the computation and writes succeeded or failed.

`.scc.mutate` query example

```
CALL neptune.algo.scc.mutate(
  {
    writeProperty: "SCOMM_ID",
    edgeLabels: ["route", ..],
    vertexLabel: "airport",
    concurrency: 2
  }
)
```

Sample `.scc.mutate` output

Here is an example of the output returned by .scc.mutate when run against the sample air-routes dataset [nodes], and sample air-routes dataset [edges], when using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "CALL neptune.algo.scc.mutate({writeProperty: 'sccid'}) YIELD success
 RETURN success" \
  --language open_cypher \
  /tmp/out.txt
```

```
cat /tmp/out.txt
{
  "results": [
    {
      "success": true
    }
  ]
}
```

Custom algorithms

Property graph information

Property Graph Information (graph.pg_info) summarizes some of the basic metrics of the graph, such as the number of vertices, the number of edges, the number of edge properties, the number of vertex properties, the number of edge labels, and the number of vertex labels.

Inputs for graph.pg_info

There are no inputs for graph.pg_info.

Outputs for graph.pg_info

There are two columns in the output relation: the first column is the metric name and the second column is the count.

metric: the metrics that graph.pg_info will return, which include:

- numVertices: the number of vertices in the graph.
- numEdges: the number of edges in the graph.
- numVertexProperties: the number of node properties in the graph.
- numEdgeProperties: the number of edge properties in the graph.
- numVertexLabels: the number of unique vertex labels in the graph.
- numEdgeLabels: the number of unique edge labels in the graph.

count

- count: the value of the above metrics.

graph.pg_info query example

```
## Syntax
CALL neptune.graph.pg_info()
YIELD metric, count
RETURN metric, count
```

graph.pg_info query integration

```
# sample query integration
CALL neptune.graph.pg_info()
YIELD metric, count
WHERE metric = 'numVertices'
RETURN count
```

Sample graph.pg_info output

```
# sample output of graph.pg_info
aws neptune-graph execute-query \

     --graph-identifier ${graphIdentifier} \
     --query-string "CALL neptune.graph.pg_info()
     YIELD metric, count
     RETURN metric, count " \
     --language open_cypher \
     /tmp/out.txt
cat /tmp/out.txt
{
  "results": [{
      "metric": "numVertices",
      "count": 3748
    }, {
      "metric": "numEdges",
      "count": 57538
    }, {
      "metric": "numVertexProperties",
      "count": 42773
    }, {
      "metric": "numEdgeProperties",
      "count": 50532
    }, {
      "metric": "numVertexLabels",
      "count": 4
    }, {
      "metric": "numEdgeLabels",
      "count": 2
    }]
}
```

Working with vector similarity in Neptune Analytics

You can answer complex questions about your data by transforming data shapes into embeddings (that is, vectors). Using a vector search index lets you answer questions about the your data's context and its similarity and connection to other data.

For example, a support agent could translate a question that they receive into a vector and use it to search the support knowledge base for articles that are similar to the words in the question (implicit similarity). For the most applicable articles, they could then collect metadata about the author, previous cases, runbooks, and so on so as to provide additional context when answering the question (explicit data).

Vector similarity search in Neptune Analytics makes it easy for you to build machine learning (ML) augmented search experiences and generative artificial intelligence (GenAI) applications. It also gives you an overall lower total cost of ownership and simpler management overhead because you no longer need to manage separate data stores, build pipelines, or worry about keep the data stores in sync. You can use vector similarity search in Neptune Analytics to augment your LLMs by integrating graph queries for domain-specific context with the results from low-latency, nearest-neighbor similarity search on embeddings imported from LLMs hosted in Amazon Bedrock, Graph Neural Networks (GNNs) in GraphStorm, or other sources.

As an example, Bioinformatics researchers who are interested in re-purposing existing blood pressure drugs for other treatable diseases, want to use vector similarity search over in-house knowledge graphs to find patterns in protein interaction networks.

For another example, a large online book retailer may need to use known pirated material to quickly identify similar media in conjunction with a knowledge graph to identify patterns of deceptive listing behaviours and find malicious sellers.

In both cases, vector search over a knowledge graph increases accuracy and speed when building the solution. It reduces the operational overhead and complexity using the tools available today.

You can create a vector index for your graph to try out this feature. Neptune Analytics supports associating embeddings generated from LLMs with the nodes of your graphs.

Contents

Vector indexing in Neptune Analytics

You can only create a vector search index for a Neptune Analytics graph at the time the graph is created. Neptune Analytics lets you create only one vector index for a graph, with a fixed dimension between 1 and 65,535 inclusive.

When you create a Neptune Analytics graph in the console, you specify the index dimension under **Vector search settings** near the end of the process.

Vector index transaction support

When using Neptune Analytics with a vector search index, it is important to understand that any updates performed on the vector index are not ACID compliant - specifically, any updates to the vector index are not atomic in nature. Atomicity in a database defines that when updates are performed, either all or none of them succeed. There are situations with the vector index where updating the embeddings may succeed, even when the remainder of the transaction fails:

- When one or more concurrent queries are executed against different vertices, then atomicity is guaranteed.
- When one or more concurrent queries are executed against the same vertex, then there is no serializable guarantee of the resulting stored data.
- If one or more queries, including `neptune.load()` updates, fail to complete then the resulting index may contain partial updates.

To minimize the potential for this issue to occur, it is recommended that you either run a single query on a single vertex at a time, or if you are running concurrent queries, that the set of vertices being updated are distinct.

Loading vectors into a Neptune Analytics graph vector index

Note that the nodes in your graph must have at least one user property or label in order to associate them with embeddings. Also, Neptune Analytics does not support the special positive and negative infinity (`INF`, `-INF`) and not-a-number (`NaN`) floating-point values.

Neptune Analytics supports optional embeddings in the CSV file when the vector index is enabled. This means that not every node needs to be associated with an embedding.

Neptune Analytics does not currently support loading vectors from Neptune Database or a snapshot.

There are two ways you can load vectors associated with nodes in your graph:

Load the vectors from graph data files Amazon S3

When you're loading graph data from files in Amazon S3 using the console or the `neptune.load{}` openCypher integration, you can add a column to your CSV data with an `embedding:vector` header. This column should contain a list of integer or floating-point values separated by semicolons (`;`) that forms a vector of the required dimension and is the embedding for the node in question.

For example, associating a 4-dimensional vector with nodes in your graph in the openCypher CSV format would look like this:

```
:ID, name:String, embedding:Vector, :LABEL
v1,"ABC",0.1;0.5;0.8;-1.32,person
v2,"DEF",8.1;-0.2;0.432;-1.02,person
v3,"GHI",12323343;24324;2433554;-4343434,person
v4,"JKL",121.12213;3223.212;265;-1.32,person
```

In the Gremlin CSV format, the same thing would look like this:

```
~id, name, embedding:vector, ~label
v1,"ABC",0.1;0.5;0.8;-1.32,person
v2,"DEF",8.1;-0.2;0.432;-1.02,person
v3,"GHI",12323343;24324;2433554;-4343434,person
v4,"JKL",121.12213;3223.212;265;-1.32,person
```

Using the `vectors.upsert` algorithm to load vectors for your graph

You can also use the vectors.upsert algorithm to insert or update embeddings in a Neptune Analytics graph that has a vector search index. For example, in openCypher you can call the algorithm like this:

```
CALL neptune.algo.vectors.upsert(
  "person933",
  [0.1, 0.2, 0.3, ..]
)
YIELD node, embedding, success
RETURN node, embedding, success
```

Another example is:

```
UNWIND [
  {id: "933", embedding: [1,2,3,4]},
  {id: "934", embedding: [-1,-2,-3,-4]}
] as entry
MATCH (n:person) WHERE id(n)=entry.id WITH n, entry.embedding as embedding
CALL neptune.algo.vectors.upsert(n, embedding)
YIELD success
RETURN n, embedding, success
```

Common errors you may encounter when loading embeddings

- If the embeddings you are trying to load have a different dimension than is expected by the vector index, the load fails with parsing exception and a message like the following:

  ```
  An error occurred (ParsingException) when calling the
            ExecuteOpenCypherQuery operation: Could not load vector embedding: (the
            embedding in question). Please check the dimensionality for this vector
            when parsing line [(line number)] in [(file
            name)]
  ```

- If the embeddings in a file are not properly formatted, Neptune Analytics reports a Parsing Exception before starting the load. For example, if the column header for the embedding column is not `embedding:vector`, Neptune Analytics would report an error like this:

  ```
  An error occurred (ParsingException) when calling the
            ExecuteOpenCypherQuery operation: Invalid data type encountered for header
            embedding:Vectttor when parsing line
            [~id, name:string, embedding:Vectttor, ~label] in [(file name)]
  ```

- If embeddings are present in a file to be loaded but no vector index is present, Neptune Analytics simply ignores the embeddings and loads the graph data without them.

Vector-search algorithms in Neptune Analytics

Neptune Analytics supports a variety of vector-search algorithms that are listed in the VSS algorithms section.

Vector-similarity search (VSS) algorithms in Neptune Analytics

Vector simlarity search algorithms identify similar vectors based on the vector distance between them.

Neptune Analytics supports the following vector-similarity search algorithms:

> **Note**
>
> The following special floating-point values are not supported in Neptune Analytics vector-similarity search algorithms:
>
> - **INF** (infinity)
> - **-INF** (negative infinity)
> - **NaN** (not-a-number)

Contents

The `.vectors.distance` algorithm

The `.vectors.distance` algorithm computes the distance between two vectors based on their embeddings. The distance is the L2 norm of the vectors.

`.vectors.distance` syntax

```
MATCH( n {`~id`: "the ID of the source node(s)"} )
MATCH( m {`~id`: "the ID of the target node(s)" })
CALL neptune.algo.vectors.distance(n, m)
YIELD distance
RETURN n, m, distance
```

`.vectors.distance` inputs

- **a source node list** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The result of a `MATCH` statement from which you want to source distance computations.

- **a target node** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The result of a `MATCH` statement from which you want to source distance computations.

> ⚠ **Warning**
>
> Be careful to limit `MATCH(n)` and `MATCH(m)` so that they don't return a large number of nodes. Keep in mind that every pair of `n` and `m` in the join result invokes `.vectors.distance` once. Too many inputs can therefore result in very long runtimes. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

`.vectors.distance` outputs

For every pair of source node and target node:

- **source** – The starting point for the distance call.
- **target** – The target for the distance computation.
- **distance** – The distance between source and target.

`.vectors.distance` query example

```
MATCH ( n {`~id`: "106"} )
MATCH ( m {`~id`: "110" } )
CALL neptune.algo.vectors.distance( n, m )
YIELD distance
RETURN n, m, distance
```

Sample `.vectors.distance` output

Here is an example of the output returned by `.vectors.distance` when run against a sample Wikipedia dataset using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "MATCH (n{`~id`: '0'})
                       MATCH (m{`~id`: '1'})
                       CALL neptune.algo.vectors.distance(n, m)
                       YIELD distance
                       RETURN n, m, distance" \
  --language open_cypher \
  /tmp/out.txt

{
  "results": [
    {
      "n": {
        "~id": "0",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 0,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "The 24-hour clock is a way of telling the time in which the day
 runs from midnight to midnight and is divided into 24 hours\\, numbered from 0 to
 23. It does not use a.m. or p.m. This system is also referred to (only in the US and
 the English speaking parts of Canada) as military time or (only in the United Kingdom
 and now very rarely) as continental time. In some parts of the world\\, it is called
 railway time. Also\\, the international standard notation of time (ISO 8601) is based
 on this format."
        }
      },
      "m": {
        "~id": "1",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 1,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "A time in the 24-hour clock is written in the form hours:minutes
 (for example\\, 01:23)\\, or hours:minutes:seconds (01:23:45). Numbers under 10 have
 a zero in front (called a leading zero); e.g. 09:07. Under the 24-hour clock system\\,
 the day begins at midnight\\, 00:00\\, and the last minute of the day begins at 23:59
 and ends at 24:00\\, which is identical to 00:00 of the following day. 12:00 can only
 be mid-day. Midnight is called 24:00 and is used to mean the end of the day and 00:00
 is used to mean the beginning of the day. For example\\, you would say \"Tuesday at
 24:00\" and \"Wednesday at 00:00\" to mean exactly the same time."
        }
      },
      "distance": 27.762847900390626
    }
  ]
}
```

The `.vectors.get` algorithm

The `.vectors.get` algorithm retrieves the embedding for a node.

`.vectors.get` syntax

```
MATCH( n {`~id`: "the ID of the node"} )
CALL neptune.algo.vectors.get(n)
YIELD embedding
RETURN n, embedding
```

`.vectors.get` input

- **a source node or nodes** *(required)* – *type:* `Node[]` or `NodeId[]`; *default: none*.

  The result of a `MATCH` statement that produces the node(s) for which you want to retrieve the embedding.

> **Warning**
>
> Be careful to limit `MATCH(n)` so that it doesn't return a large number of nodes. Keep in mind that every source node in the `n` result invokes `.vectors.get` once. Too many inputs can therefore result in very long runtimes. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

`.vectors.get` outputs

For each source node provided:

- **node** – The source node.
- **embedding** – The embedding of that source node.

`.vectors.get` query example

```
MATCH ( n {`~id`: "0"} )
CALL neptune.algo.vectors.get(n)
YIELD embedding
```

```
RETURN n, embedding
```

Sample .vectors.get output

Here is an example of the output returned by `.vectors.get` when run against the sample Wikipedia dataset using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "MATCH ( n {`~id`: '0'} )
                       CALL neptune.algo.vectors.get(n)
                       YIELD embedding
                       RETURN n, embedding" \
  --language open_cypher \
  /tmp/out.txt
{
  "results": [
    {
      "n": {
        "~id": "0",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 0,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "The 24-hour clock is a way of telling the time in which the day
 runs from midnight to midnight and is divided into 24 hours\\, numbered from 0 to
 23. It does not use a.m. or p.m. This system is also referred to (only in the US and
 the English speaking parts of Canada) as military time or (only in the United Kingdom
 and now very rarely) as continental time. In some parts of the world\\, it is called
 railway time. Also\\, the international standard notation of time (ISO 8601) is based
 on this format."
        }
      },
      "embedding": [
        0.07711287587881088,
        0.3197174072265625,
       -0.2051590085029602,
        0.6302579045295715,
         0.032093219459056857,
         0.200703963637352,
         0.16665680706501008,
        -0.31295087933540347,
         0.17575109004974366,
         0.5308129191398621,
        -0.37528499960899355,
         0.3338659405708313,
        -0.046272162348032,
         0.07841536402702332,
        -0.3490406274795532,
         0.27182886004447939,
         0.3073517680168152,
        -0.08306130766868592,
         0.5035958886146545,
         0.254621684551239,
        -0.40407684445381167,
         0.28878292441368105,
        -0.22588828206062318,
        -0.13185778260231019,
        -0.21559733152389527,
         0.4900434613227844,
         0.03866531699895859,
         0.507415771484375,
        -0.3067346513271332,
         0.10740984976291657,
         0.08998646587133408,
        -0.2652775049209595,
        -0.28492602705955508,
         0.33600345253944399,
        -0.27227747440338137,
         0.3691731095314026,
        -0.2815995514392853,
         0.0856710895895958,
        -0.13187488913536073,
         0.4753035008907318,
        -0.2241700142621994,
         0.20263174176216126,
         0.4390721619129181,
         0.06424559652805329,
         0.2463042289018631,
        -0.39631763100624087,
         0.2971232533454895,
         0.2415716052055359,
        -0.02803819440305233,
         0.32105034589767458,
        -0.02222033031284809,
        -0.008510420098900795,
        -0.00032598740654066205,
         0.031057516112923623,
        -0.5332233309745789,
         0.45022767782211306,
        -0.6829474568367004,
         1.3313145637512208,
         0.19445496797561646,
        -0.15697629749774934,
        -0.09996363520622254,
        -0.2786232829093933,
        -0.09833164513111115,
        -0.17644722759723664,
         0.11717787384986878,
         0.2820119559764862,
         0.029635537415742875,
         0.5247654914855957,
         0.5323811173439026,
        -0.06254086643457413,
        -0.05274389684200287,
         0.3877565860748291,
         0.43260684609413149,
         0.5207982063293457,
        -0.27160540223121645,
        -0.06000519543886185,
        -0.032806672155857089,
        -0.3594319522380829,
         0.4218965470790863,
        -0.3766363263130188,
         0.44727250933647158,
        -0.04586323723196983,
         0.06902860850095749,
         0.3030509352684021,
         0.18945887684822083,
         0.21681705117225648,
        -0.014492596499621868,
        -0.38649576902389529,
        -0.1129651814699173,
         0.050081491470336917,
        -0.01697717048227787,
         0.1415158063173294,
        -0.3284287750720978,
        -0.02309800498187542,
        -0.2051207274198532,
        -0.017861712723970414,
        -0.07372242212295532,
        -0.12263767421245575,
         0.21828559041023255,
        -0.36898064613342287,
         0.3558262288570404,
        -0.16924124956130982,
        -0.31757786870002749,
         0.5452765226364136,
         0.24666202068328858,
        -0.08289600908756256,
        -0.14674079418182374,
        -0.18049933016300202,
         0.3646247982978821,
         0.42489132285118105,
         0.0909421369433403,
        -0.1764664500951767,
         0.22471413016319276,
         0.049531541764736179,
        -0.022898104041814805,
         0.08607156574726105,
         0.14532636106014253,
        -0.205774188041687,
        -0.3457978069782257,
        -1.2771626710891724,
         0.2826114892959595,
         0.2066900134086609,
        -0.3884444832801819,
        -0.3564482629299164,
        -0.25118574500083926,
        -0.728326141834259,
         0.5217206478118897,
        -0.43305152654647829,
         0.3510914444923401,
         0.5106240510940552,
        -0.11594267934560776,
         0.43993058800697329,
         0.25412991642951968,
         0.4275965392589569,
         0.1463870108127594,
         0.3510439395904541,
         0.1619710624217987,
         0.11160195618867874,
        -0.22760489583015443,
        -0.23652249574661256,
         0.05374380201101303,
         0.7251803278923035,
        -0.13991153240203858,
         0.9363659024238586,
        -0.05858418717980385,
         0.5233941674232483,
         0.12388131022453308,
         0.6248424649238586,
        -0.11751417070627213,
         0.09689709544181824,
         0.7467237710952759,
         0.2247271090745926,
        -0.6747357845306397,
        -0.16039365530014039,
        -0.41555172204971316,
        -0.04566565155982971,
         0.21260707080364228,
         0.2549103796482086,
         0.24795542657375337,
         0.5625612735748291,
         0.8036459684371948,
         0.15800043940544129,
         0.04797195643186569,
        -0.15839435160160066,
        -0.06506697088479996,
        -0.2577322721481323,
         0.3262946903705597,
         0.5458049178123474,
         0.616370439529419,
        -0.35092639923095705,
         0.048758912831544879,
         0.11522434651851654,
         0.04175107553601265,
        -0.12269306182861328,
         0.1227836161851883,
         0.4020257890224457,
         0.07093577086925507,
        -0.1880340874195099,
         0.5334663391113281,
         0.46888044476509097,
         0.18104688823223115,
         0.30756646394729617,
         0.29316428303718569,
        -0.10604366660118103,
         0.44999250769615176,
         0.18227706849575044,
         0.5962150692939758,
         0.38278165459632876,
        -0.40461188554763796,
         0.17775404453277589,
        -0.16349074244499207,
         0.06950787454843521,
         0.7547341585159302,
        -0.4842711389064789,
         0.4062837064266205,
         0.09000574052333832,
         0.03859427571296692,
         0.24143263697624207,
        -0.3383118510246277,
         0.3363209366798401,
         0.10778547078371048,
         0.3429640233516693,
        -0.20395530760288239,
         0.011477324180305004,
         0.6145590543746948,
        -0.5488739609718323,
        -0.26194247603416445,
        -0.09723474085330963,
        -0.19020821154117585,
        -0.18068274855613709,
         0.1601778119802475,
         0.038950759917497638,
         0.6372026205062866,
        -0.12897184491157533,
         0.10720998793840409,
         0.13482464849948884,
        -0.07540713250637055,
        -0.0881727784872055,
         0.5626690983772278,
        -0.31975486874580386,
        -0.029084375128149987,
         0.43618619441986086,
         0.32975345849990847,
        -0.4053913652896881,
        0.15788795053958894,
       -0.3212168216705322,
       -0.20272433757781983,
       -0.8973743319511414,
        0.060059018433094028,
       -0.014103145338594914,
       -0.3387225568294525,
       -0.49839726090431216,
       -0.011007139459252358,
       -0.16101065278053285,
       -0.20850643515586854,
        0.4891682267189026,
        0.33551496267318728,
       -0.23595896363258363,
       -0.4257577359676361,
       -0.48884832859039309,
        0.48760101199150088,
        0.34031161665916445,
        0.1722799688577652,
       -0.35575979948043826,
        0.629051923751831,
       -0.8014369010925293,
        0.575096607208252,
        0.421142578125,
       -0.2668846547603607,
       -0.046029768884181979,
        0.2791147530078888,
       -0.22112232446670533,
        0.02008579671382904,
        0.22087614238262177,
       -0.17961964011192323,
        0.4235396981239319,
        0.295818567276001,
       -0.18260923027992249,
        0.3227207660675049,
        0.11412205547094345,
        0.04591478034853935,
        0.5127033591270447,
        0.428005576133728,
        0.20718106627464295,
        0.18405631184577943,
       -0.22416146099567414,
        0.4277373254299164,
        0.5384698510169983,
        0.04109276458621025,
        0.5105301141738892,
        0.473961740732193,
       -0.6853302717208862,
       -0.16557902097702027,
       -0.12704522907733918,
        0.0026600745040923359,
        0.5272349715232849,
        0.12121742218732834,
        0.427141010761261,
       -0.3047095239162445,
        0.5948843359947205,
        0.335798442363739,
        0.35749775171279909,
       -0.18497343361377717,
        0.26501506567001345,
        0.1564970314502716,
        0.4210122525691986,
       -0.1915784478187561,
        0.057152874767780307,
       -0.28498271107673647,
        0.04969947412610054,
        0.7697478532791138,
        0.5546697974205017,
        0.0958070456981659,
       -0.3533228933811188,
        0.4784282147884369,
        0.624963104724884,
        0.2151053100824356,
        0.17361000180244447,
        0.22527147829532624,
       -0.12481484562158585,
        0.4212929904460907,
       -0.2926572859287262,
        0.2562543749809265,
        0.38751208782196047,
        0.1340814083814621,
        0.0680900365114212,
        0.2952287793159485,
        0.12217980623245239,
       -0.2869758605957031,
        0.15682946145534516,
       -0.022066200152039529,
       -0.09002991020679474,
       -0.2826828360557556,
        0.84619140625,
        0.7544476985931397,
        0.5953861474990845,
        0.6517565250396729,
       -0.07932830601930618,
        0.22802823781967164,
       -0.135965958237648,
       -0.8263510465621948,
       -0.6325801610946655,
       -0.5928561091423035,
        0.4108763635158539,
        0.0964483916759491,
       -0.5045000910758972,
       -0.06772734969854355,
       -0.79107666015625,
        0.060380879789590839,
        0.015578197315335274,
        0.32540079951286318,
       -0.044692762196063998,
       -0.17132098972797395,
       -0.19123415648937226,
        0.17911623418331147,
        0.3269428014755249,
       -0.22874118387699128,
        0.4686919152736664,
       -0.15749554336071015,
       -0.25185921788215639,
       -0.21561351418495179,
       -0.10132477432489395,
       -0.057977184653282168,
        0.09759098291397095,
        0.16202516853809358,
        0.01888692006468773,
        0.1724688857793808,
       -0.3449697196483612,
        0.4449881315231323,
        0.10185430943965912,
       -0.2976726293563843,
        0.06075461208820343,
        0.21909406781196595,
       -0.07409229874610901,
        0.6881160140037537,
        0.17447273433208466,
       -0.048471711575984958,
        0.5318611264228821,
        0.30954766273498537,
       -0.24350836873054505,
        0.14386573433876038,
       -0.10827953368425369,
        0.08575868606567383,
        0.14200334250926972,
        0.5095603466033936,
       -0.025056177750229837,
        0.24901045858860017,
       -0.23696841299533845,
       -0.03630203381180763,
        0.45206722617149355,
        0.5019969344139099,
       -0.21705971658229829,
       -0.08452687412500382,
       -0.10376924276351929,
       -0.3200875520706177,
       -0.2048267275094986,
       -0.2703971266746521,
        0.2925371825695038,
        0.3755778670310974,
        0.2522588074207306,
        0.22964833676815034,
        0.7995960116386414,
        0.12206973880529404,
        0.2896155118942261,
        0.04163726791739464,
       -0.12602514028549195,
        0.004978220444172621,
        0.3399927020072937,
        0.09124521911144257,
       -0.5452605485916138,
        0.2247130423784256,
        0.23503662645816804,
        0.06750215590000153,
       -0.2884872257709503,
       -0.2791622579097748,
       -0.1780446618795395,
       -0.44350507855415347,
       -0.1840016394853592,
        0.8970789909362793,
       -0.3687478303909302,
        0.36603569984436037,
        0.23560358583927155,
        0.020292289555072786,
        0.2446030080318451,
        4.3314642906188969,
        0.194863960146904,
       -0.10218192636966706,
        0.5695234537124634,
        0.016988292336463929,
       -0.15768325328826905,
        0.050476688891649249,
        0.09948820620775223,
       -0.06554386019706726,
        0.22301962971687318,
       -0.05468735471367836,
        0.29051196575164797,
        0.12100572139024735,
        0.4127441644668579,
        0.1667146235704422,
        0.0587792843580246,
       -0.09758614003658295,
       -0.20510408282279969,
       -0.21746976673603059,
        0.43335747718811037,
       -0.32159093022346499,
        0.6942153573036194,
        0.6173154711723328,
        0.3104712665081024,
        0.5751503109931946,
        0.4174514412879944,
       -0.2948107421398163,
        0.3532458245754242,
        0.4869029223918915,
        0.3115881681442261,
        0.28135108947753909,
        0.38450825214385989,
        0.016915690153837205,
       -0.11598393321037293,
       -0.32250434160232546,
       -0.06988134980201721,
        0.22417351603507996,
       -0.35582518577575686,
        0.2677224576473236,
        0.008019124157726765,
       -0.19177919626235963,
        0.5731900334358215,
       -0.03540642186999321,
        0.43302130699157717,
        0.1796148121356964,
       -0.005056577268987894,
        0.37953320145606997,
        0.13488957285881043,
        0.7240068912506104,
       -0.3088097870349884,
        0.5610846281051636,
       -0.29582735896110537,
       -0.20909856259822846,
       -0.2881403863430023,
        0.10329002141952515,
        0.49255961179733279,
        0.14558906853199006,
        0.41020694375038149,
        0.04002099484205246,
       -0.24476903676986695,
       -0.389543354511261,
        0.3901459574699402,
        0.6170359253883362,
        0.18917717039585114,
       -0.41235554218292239,
       -0.19313344359397889,
       -0.10294703394174576,
        0.5560699105262756,
        0.5773581266403198,
       -0.17282086610794068,
        0.28679269552230837,
        0.34322652220726015,
       -0.07227988541126251,
       -0.5244243741035461,
       -0.26529040932655337,
       -0.11131077259778977,
       -0.19524210691452027,
        0.4082769453525543,
       -0.009217939339578152,
       -0.1462743580341339,
        0.7264918684959412,
       -0.09149657934904099,
       -0.3374916911125183,
       -0.05742226541042328,
       -0.3913151025772095,
        0.7185215950012207,
       -0.3785516619682312,
       -0.00010882654169108719,
        0.6655824780464172,
        0.4194306433200836,
        0.3726831376552582,
       -0.014721312560141087,
        0.5345744490623474,
        0.33022087812423708,
       -0.06344814598560333,
       -0.1560882031917572,
        0.22698232531547547,
       -3.8697707653045656,
        0.06812435388565064,
       -0.4368731677532196,
       -0.07041455805301666,
       -0.015291529707610608,
       -0.41140303015708926,
        0.31612321734428408,
        0.2914712429046631,
       -0.3867192566394806,
       -0.026363473385572435,
       -0.08788029104471207,
       -0.10701339691877365,
       -0.2673511505126953,
        0.27538666129112246,
       -0.3661351501941681,
        0.5879861116409302,
        0.06352981925010681,
        0.15547777712345124,
        0.0863194614648819,
       -0.021183960139751436,
        0.428565114736557,
        0.04859453812241554,
        0.35721391439437868,
       -0.3864029347896576,
       -0.20986808836460114,
        0.15433000028133393,
        0.25567296147346499,
        0.25359275937080386,
       -0.4783596396446228,
       -0.010366495698690415,
        0.4777776598930359,
       -0.029405448585748674,
        0.3631121814250946,
       -0.18738743662834168,
        0.2193489819765091,
        0.7861229777336121,
       -0.01961355283856392,
        0.16653983294963838,
       -0.4193624258041382,
        0.3085209131240845,
       -0.03517897054553032,
       -0.035910699516534808,
        0.37241387367248537,
       -0.13769084215164185,
       -0.08015040308237076,
        0.4384872615337372,
       -0.12396809458732605,
        0.15661391615867616,
       -0.3919837176799774,
       -0.6586825251579285,
        0.5687432885169983,
        0.0396936871111393,
       -0.09660491347312927,
        0.05788198113441467,
        0.48911261558532717,
        0.5213083028793335,
        0.3355415165424347,
       -0.006735790055245161,
       -0.11381038278341294,
        0.09182903915643692,
       -0.11055094748735428,
       -0.28275448083877566,
        0.24975340068340302,
        0.11746659129858017,
       -0.42452141642570498,
       -0.2323901206254959,
       -0.38694220781326296,
        0.015501483343541623,
        0.6440262198448181,
       -0.3121536672115326,
       -0.08778296411037445,
       -0.14549347758293153,
        0.01749151013791561,
       -0.5398207902908325,
        0.4124368131160736,
        0.5154116749763489,
       -0.34769660234451296,
        0.5662841796875,
        0.4989481270313263,
        0.06761053949594498,
        0.014184223487973214,
        0.601079523563385,
       -0.3859538435935974,
        0.3446619212627411,
        2.190366744995117,
        0.4051366150379181,
        2.288928508758545,
        0.5293960571289063,
       -0.3505767583847046,
        0.5397417545318604,
       -0.6520821452140808,
        0.4239364266395569,
        0.2618080675601959,
        0.20174439251422883,
        0.030146604403853418,
        0.0610184520483017,
        0.062213074415922168,
       -0.11276254057884217,
       -0.1301877349615097,
       -0.19404706358909608,
        0.5268515348434448,
       -0.7370991706848145,
        0.028712594881653787,
       -0.4024544954299927,
        0.18225152790546418,
        0.7267741560935974,
       -0.2734072208404541,
        0.1759040206670761,
       -0.2950340211391449,
        0.14166314899921418,
        0.6515365242958069,
       -0.29643580317497256,
       -0.06734377890825272,
        0.09662584215402603,
       -0.010966300964355469,
       -0.3204823136329651,
        0.6417866349220276,
       -0.051218003034591678,
       -0.008819818496704102,
        0.5098630785942078,
       -0.21459998190402986,
        4.437846660614014,
       -0.24779054522514344,
        0.018799694254994394,
       -0.01747281290590763,
       -0.0487254373729229,
        0.6121163964271545,
        0.4686623811721802,
       -0.22926479578018189,
       -0.03692511469125748,
       -0.4286654591560364,
        0.46073317527770998,
        0.16875289380550385,
       -0.014255600981414318,
       -0.07684683054685593,
        0.12223237752914429,
       -0.30599895119667055,
        0.39215049147605898,
        0.22453786432743073,
        0.5624862313270569,
       -0.011985340155661106,
        0.05180392041802406,
        0.030400553718209268,
        0.08391892164945603,
        0.10214067250490189,
       -0.4449590742588043,
        0.2225639522075653,
        0.3862999975681305,
        0.24732927978038789,
       -0.05571140721440315,
       -0.021564822643995286,
        0.28468334674835207,
        5.213898658752441,
        0.13289497792720796,
       -0.1400047093629837,
       -0.39865049719810488,
        0.12139834463596344,
        0.45539018511772158,
       -0.1865275651216507,
       -0.08270177245140076,
       -0.38520801067352297,
        0.08869948983192444,
       -0.05266271159052849,
         0.14364486932754517,
        -0.2860695719718933,
         0.4430652856826782,
         0.7777798771858215,
         0.21114271879196168,
        -0.358752578496933,
        -0.3664247989654541,
         0.6665846109390259,
        -0.40493687987327578,
         0.1747705042362213,
        -0.06670021265745163,
         0.20972059667110444,
        -0.19101694226264954,
         0.23892535269260407,
        -0.08149895817041397,
         0.018510373309254648,
         0.8112999796867371,
         0.07871513813734055,
         0.09570053964853287,
         0.5030911564826965,
         0.21463628113269807,
        -0.31457462906837466,
         0.3051794767379761,
        -0.39506298303604128,
         0.06605447828769684,
         0.6144300699234009,
        -0.4566810429096222,
         0.3146623373031616,
         0.1887989640235901,
         0.9544244408607483,
         0.5103438496589661,
        -0.4859951138496399,
        -0.32647767663002016,
        -0.07584235072135925,
         0.21474787592887879,
        -0.1920636147260666,
        -0.4472030997276306,
         0.08618132770061493,
        -0.17384092509746552,
        -0.20969024300575257,
        -0.1831870973110199,
         0.8782939314842224,
        -0.15720084309577943,
         0.37347128987312319,
         0.5088165998458862,
         0.29395583271980288,
        -0.3580363988876343,
        -0.17590023577213288,
        -0.508141279220581,
         0.4661521315574646,
         0.142064169049263,
        -0.05615571141242981,
         0.592810869216919,
         0.37807324528694155,
        -0.14052101969718934,
        -0.19951890408992768,
        -0.12800109386444093,
         0.748070478439331,
         0.13753947615623475,
        -0.08446942269802094,
         0.3747580945491791,
        -0.12847286462783814,
        -0.13892321288585664,
         0.08525972813367844,
         0.12516680359840394,
         0.5701874494552612,
        -0.24708901345729829,
         0.0679594948887825,
         0.10870008915662766,
         0.20561885833740235,
        -0.7872452139854431,
         0.07303950190544129,
         0.35694700479507449,
         0.245212584733963,
         0.3299793303012848,
        -0.010669616051018238,
        -0.12047348916530609,
         0.3540535271167755,
         0.32180890440940859,
         0.3066200911998749,
         0.021576205268502237,
         0.17679384350776673,
        -0.23050960898399354,
         0.1292697787284851,
         0.022921407595276834,
         0.5460971593856812,
         0.3612038493156433,
         0.1963733434677124,
         0.4622957706451416,
         0.16855642199516297,
         0.2564740478992462,
        -0.27637141942977908,
        -0.16345584392547608,
         0.08119463175535202,
         0.07851938903331757,
        -0.5181471109390259,
        -0.5290305614471436,
         0.5271350741386414,
         0.3391841650009155,
         0.501441240310669,
         0.740936279296875,
        -0.26713573932647707,
         0.030347898602485658,
         0.05174243822693825
       ]
     }
   ]
 }
```

.vectors.topKByEmbedding algorithm

The `.vectors.topKByEmbedding` algorithm finds the `topK` nearest neighbors of an embedding based on the distance of their vector embeddings.

`.vectors.topKByEmbedding` syntax

```
CALL neptune.algo.vectors.topKByEmbedding(
  [an embedding (required)],
  {
    topK: the number of result nodes to return (optional, default: 10),
    concurrency: the number of cores to use to run the algorithm (optional)
  }
)
YIELD embedding, node, score
RETURN embedding, node, score
```

`.vectors.topKByEmbedding` input

- **an embedding** *(required) type:* a list of floating-point values.

  The source input embedding to use to compute the distance to the embeddings of the candidate target nodes. The dimension of the embedding must match the declared dimension of the associated vector index.

  The embedding may or may not exist in the database. If not, it can be any vector of the same dimension as is declared in the associated vector index.

- **topK** *(optional)* *type:* a positive integer; *default:* 10.

  The number of result nodes to return.

- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to 1, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

`.vectors.topKByEmbedding` outputs

For each node returned:

- **embedding** – The input embedding.
- **node** – A node whose embedding is at one of the `topK` nearest distances from the input embedding.
- **score** – The distance between the input embedding and the embedding of this node.

`.vectors.topKByEmbedding` query example

You can provide the embedding explicitly in the query, although embeddings tend to be very large:

```
CALL neptune.algo.vectors.topKByEmbedding(
  [0.1, 0.2, 0.3, ...],
  {
    topK: 7,
    concurrency: 1
  }
)
YIELD embedding, node, score
RETURN embedding, node, score
```

Most often, you will by generating embeddings to pass to the algorithm. For example:

```
MATCH ( n:airport {code: 'ANC'} )
CALL neptune.algo.vectors.get(n) YIELD embedding AS vector WITH vector
CALL neptune.algo.vectors.topKByEmbedding(
  vector,
  {
    topK: 10,
    concurrency: 1
  }
)
YIELD node, score
RETURN vector, node, score
```

Warning

In queries like the one above, be careful to limit `MATCH(n)` so that it doesn't return a large number of nodes. Keep in mind that every node in `n` invokes a separate run of both `.vectors.get` and `.vectors.topKByEmbedding`. Too many inputs can therefore result in very long runtimes. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.vectors.topKByEmbedding` output

Here is an example of the output returned by `.vectors.topKByEmbedding` when run against the sample Wikipedia dataset using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "MATCH ( n {`~id`: '0'} )
                      CALL neptune.algo.vectors.get(n) YIELD embedding AS vector
                      CALL neptune.algo.vectors.topKByEmbedding( vector, { topK: 3 })
                      YIELD node, score
                      RETURN node, score"
  --language open_cypher \
  /tmp/out.txt
{
  "results": [
    {
      "node": {
        "~id": "0",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 0,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "The 24-hour clock is a way of telling the time in which the day
 runs from midnight to midnight and is divided into 24 hours\\, numbered from 0 to
 23. It does not use a.m. or p.m. This system is also referred to (only in the US and
 the English speaking parts of Canada) as military time or (only in the United Kingdom
 and now very rarely) as continental time. In some parts of the world\\, it is called
 railway time. Also\\, the international standard notation of time (ISO 8601) is based
 on this format."
        }
      },
      "score": 0.0
    },
    {
      "node": {
        "~id": "2",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 2,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "However\\, the US military prefers not to say 24:00 - they do not
 like to have two names for the same thing\\, so they always say \"23:59\"\\, which is
 one minute before midnight."
        }
      },
      "score": 24.000200271606447
    },
    {
      "node": {
        "~id": "3",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 3,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "24-hour clock time is used in computers\\, military\\, public safety
\\, and transport. In many Asian\\, European and Latin American countries people use it
 to write the time. Many European people use it in speaking."
        }
      },
      "score": 25.013729095458986
    }
```

```
  ]
}
```

.vectors.topKByNode algorithm

The `.vectors.topKByNode` algorithm finds the `topK` nearest neighbors of a node based on the distance of their vector embeddings from the node.

`.vectors.topKByNode` syntax

```
CALL neptune.algo.vectors.topKByNode(
  [a list of one or more nodes (required)],
  {
    topK: the number of result nodes to return (optional, default: 10),
    concurrency: the number of cores to use to run the algorithm (optional)
  }
)
YIELD node, score
RETURN node, score
```

`.vectors.topKByNode` input

- **a list of one or more source nodes** *(required)* – *type:* `Node[]` or `NodeId[]`.

  If the source-node list is empty then the query result is also empty.
- **topK** *(optional)* *type:* a positive integer; *default:* 10.

  The number of result nodes to return.
- **concurrency** *(optional)* – *type:* 0 or 1; *default:* 0.

  Controls the number of concurrent threads used to run the algorithm.

  If set to `0`, uses all available threads to complete execution of the individual algorithm invocation. If set to `1`, uses a single thread. This can be useful when requiring the invocation of many algorithms concurrently.

`.vectors.topKByNode` outputs

For each source node:

- **source** – The source node.
- **node** – A node whose embedding is at one of the `topK` nearest distances from the source node's embedding.

- **score** – The distance between the source node's embedding and the embedding of the close node.

`.vectors.topKByNode` query example

```
MATCH ( n:airport {code: 'ANC'} )
CALL neptune.algo.vectors.topKByNode(
  n,
  {
    topK: 10,
    concurrency: 1
  }
)
YIELD node, score
RETURN n, node, score
```

> ⚠ **Warning**
>
> In queries like the one above, be careful to limit `MATCH(n)` so that it doesn't return a large number of nodes. Keep in mind that every node in `n` invokes a separate run of `.vectors.topKByNode`. Too many inputs can therefore result in very long runtimes. Use `LIMIT` or put conditions on the `MATCH` clause to restrict its output appropriately.

Sample `.vectors.topKByNode` output

Here is an example of the output returned by `.vectors.topKByNode` when run against the sample Wikipedia dataset using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "MATCH ( n {`~id`: '0'} )
                       CALL neptune.algo.vectors.topKByNode(n, {topK: 3})
                       YIELD node, score
                       RETURN n, node, score" \
  --language open_cypher \
  /tmp/out.txt
{
  "results": [
    {
      "n": {
        "~id": "0",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 0,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "The 24-hour clock is a way of telling the time in which the day
 runs from midnight to midnight and is divided into 24 hours\\, numbered from 0 to
 23. It does not use a.m. or p.m. This system is also referred to (only in the US and
 the English speaking parts of Canada) as military time or (only in the United Kingdom
 and now very rarely) as continental time. In some parts of the world\\, it is called
 railway time. Also\\, the international standard notation of time (ISO 8601) is based
 on this format."
        }
      },
      "node": {
        "~id": "0",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 0,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "The 24-hour clock is a way of telling the time in which the day
 runs from midnight to midnight and is divided into 24 hours\\, numbered from 0 to
 23. It does not use a.m. or p.m. This system is also referred to (only in the US and
 the English speaking parts of Canada) as military time or (only in the United Kingdom
 and now very rarely) as continental time. In some parts of the world\\, it is called
 railway time. Also\\, the international standard notation of time (ISO 8601) is based
 on this format."
        }
      },
      "score": 0.0
    },
    {
      "n": {
        "~id": "0",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 0,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "The 24-hour clock is a way of telling the time in which the day
 runs from midnight to midnight and is divided into 24 hours\\, numbered from 0 to
 23. It does not use a.m. or p.m. This system is also referred to (only in the US and
 the English speaking parts of Canada) as military time or (only in the United Kingdom
 and now very rarely) as continental time. In some parts of the world\\, it is called
 railway time. Also\\, the international standard notation of time (ISO 8601) is based
 on this format."
        }
      },
      "node": {
        "~id": "2",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 2,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "However\\, the US military prefers not to say 24:00 - they do not
 like to have two names for the same thing\\, so they always say \"23:59\"\\, which is
 one minute before midnight."
        }
      },
      "score": 24.000200271606447
    },
    {
      "n": {
        "~id": "0",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 0,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "The 24-hour clock is a way of telling the time in which the day
 runs from midnight to midnight and is divided into 24 hours\\, numbered from 0 to
 23. It does not use a.m. or p.m. This system is also referred to (only in the US and
 the English speaking parts of Canada) as military time or (only in the United Kingdom
 and now very rarely) as continental time. In some parts of the world\\, it is called
 railway time. Also\\, the international standard notation of time (ISO 8601) is based
 on this format."
        }
      },
      "node": {
        "~id": "3",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 3,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "24-hour clock time is used in computers\\, military\\, public safety
\\, and transport. In many Asian\\, European and Latin American countries people use it
 to write the time. Many European people use it in speaking."
        }
      },
      "score": 25.013729095458986
    }
  ]
}
```

.vectors.upsert algorithm

The `.vectors.upsert` algorithm is used to add a new embedding or update an existing one for a node.

`.vectors.upsert` syntax

```
CALL neptune.algo.vectors.upsert(
  "a target node (required)",
  [the embedding to upsert for the target node (required)]
)
YIELD node, embedding, success
RETURN node, embedding, success
```

`.vectors.upsert` input

- **a target node** *(required)* – *type:* Node or `NodeId`.

  The node for which you want to upsert an embedding.
- **an embedding** *(required)* – *type:* a list of floating-point values.

  The embedding that you want to upsert for the target node.

  If the node has an existing embedding, this must match the dimension of the existing one or an exception is thrown.

`.vectors.upsert` outputs

If the target node already has an existing embedding then `.vectors.upsert` replaces it with the one supplied. Otherwise `.vectors.upsert` adds the supplied embedding for the target node.

- **node** – The target node.
- **embedding** – The embedding that was supplied to be upserted.
- **success** – A Boolean value: `true` indicates that the upsert succeded, and `false` that it failed.

`.vectors.upsert` query examples

```
CALL neptune.algo.vectors.upsert(
  "person933",
  [0.1, 0.2, 0.3, ..]
)
YIELD node, embedding, success
RETURN node, embedding, success
```

```
UNWIND [
  {id: "933", embedding: [1,2,3,4]},
  {id: "934", embedding: [-1,-2,-3,-4]}
] as entry
MATCH (n:person) WHERE id(n)=entry.id WITH n, entry.embedding as embedding
CALL neptune.algo.vectors.upsert(n, embedding)
YIELD success
RETURN n, embedding, success
```

Sample `.vectors.upsert` output

Here is an example of the output returned by `.vectors.upsert` when run against the sample Wikipedia dataset using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "MATCH (n{\`~id\`:\"0\"})
                       CALL neptune.algo.vectors.get(n)
                       YIELD embedding AS vector
                       MATCH (m{`~id`: '1'})
                       CALL neptune.algo.vectors.upsert(m, vector)
                       YIELD node, embedding, success
                       RETURN node, embedding, success"
  --language open_cypher \
  /tmp/out.txt
{
  "results": [
    {
      "node": {
        "~id": "1",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 1,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "A time in the 24-hour clock is written in the form hours:minutes
 (for example\\, 01:23)\\, or hours:minutes:seconds (01:23:45). Numbers under 10 have
 a zero in front (called a leading zero); e.g. 09:07. Under the 24-hour clock system\\,
 the day begins at midnight\\, 00:00\\, and the last minute of the day begins at 23:59
 and ends at 24:00\\, which is identical to 00:00 of the following day. 12:00 can only
 be mid-day. Midnight is called 24:00 and is used to mean the end of the day and 00:00
 is used to mean the beginning of the day. For example\\, you would say \"Tuesday at
 24:00\" and \"Wednesday at 00:00\" to mean exactly the same time."
        }
      },
      "embedding": [
        0.07711287587881088,
        0.3197174072265625,
       -0.2051590085029602,
        0.6302579045295715,
        0.032093219459056857,
        0.200703963637352,
        0.16665680706501008,
       -0.31295087933540347,
        0.17575109004974366,
        0.5308129191398621,
       -0.37528499960899355,
        0.3338659405708313,
       -0.046272162348032,
        0.07841536402702332,
       -0.3490406274795532,
        0.27182886004447939,
        0.3073517680168152,
       -0.08306130766868592,
        0.5035958886146545,
        0.254621684551239,
       -0.40407684445381167,
        0.28878292441368105,
       -0.22588828206062318,
       -0.13185778260231019,
       -0.21559733152389527,
        0.4900434613227844,
        0.03866531699895859,
        0.507415771484375,
       -0.3067346513271332,
        0.10740984976291657,
        0.08998646587133408,
       -0.2652775049209595,
       -0.28492602705955508,
        0.33600345253944399,
       -0.27227747440338137,
        0.3691731095314026,
       -0.2815995514392853,
        0.0856710895895958,
       -0.13187488913536073,
        0.4753035008907318,
       -0.2241700142621994,
        0.20263174176216126,
        0.4390721619129181,
        0.06424559652805329,
        0.2463042289018631,
       -0.39631763100624087,
        0.2971232533454895,
        0.2415716052055359,
       -0.02803819440305233,
        0.32105034589767458,
       -0.02222033031284809,
       -0.008510420098900795,
       -0.00032598740654066205,
        0.031057516112923623,
       -0.5332233309745789,
        0.45022767782211306,
       -0.6829474568367004,
        1.3313145637512208,
        0.19445496797561646,
       -0.15697629749774934,
       -0.09996363520622254,
       -0.2786232829093933,
       -0.09833164513111115,
       -0.17644722759723664,
        0.11717787384986878,
        0.2820119559764862,
        0.029635537415742875,
        0.5247654914855957,
        0.5323811173439026,
       -0.06254086643457413,
       -0.05274389684200287,
        0.3877565860748291,
        0.43260684609413149,
        0.5207982063293457,
       -0.27160540223121645,
       -0.06000519543886185,
       -0.032806672155857089,
       -0.3594319522380829,
        0.4218965470790863,
       -0.3766363263130188,
        0.44727250933647158,
       -0.04586323723196983,
        0.06902860850095749,
        0.3030509352684021,
        0.18945887684822083,
        0.21681705117225648,
       -0.014492596499621868,
       -0.38649576902389529,
       -0.1129651814699173,
        0.050081491470336917,
       -0.01697717048227787,
        0.1415158063173294,
       -0.3284287750720978,
       -0.02309800498187542,
       -0.2051207274198532,
       -0.017861712723970414,
       -0.07372242212295532,
       -0.12263767421245575,
        0.21828559041023255,
       -0.36898064613342287,
        0.3558262288570404,
       -0.16924124956130982,
       -0.31757786870002749,
        0.5452765226364136,
        0.24666202068328858,
       -0.08289600908756256,
       -0.14674079418182374,
       -0.18049933016300202,
        0.3646247982978821,
        0.42489132285118105,
        0.0909421369433403,
       -0.1764664500951767,
        0.22471413016319276,
        0.049531541764736179,
       -0.022898104041814805,
        0.08607156574726105,
        0.14532636106014253,
       -0.205774188041687,
       -0.3457978069782257,
       -1.2771626710891724,
        0.2826114892959595,
        0.2066900134086609,
       -0.3884444832801819,
       -0.3564482629299164,
       -0.25118574500083926,
       -0.728326141834259,
        0.5217206478118897,
       -0.43305152654647829,
        0.3510914444923401,
        0.5106240510940552,
       -0.11594267934560776,
        0.43993058800697329,
        0.25412991642951968,
        0.4275965392589569,
        0.1463870108127594,
        0.3510439395904541,
        0.1619710624217987,
        0.11160195618867874,
       -0.22760489583015443,
       -0.23652249574661256,
        0.05374380201101303,
        0.7251803278923035,
       -0.13991153240203858,
        0.9363659024238586,
       -0.05858418717980385,
        0.5233941674232483,
        0.12388131022453308,
        0.6248424649238586,
       -0.11751417070627213,
        0.09689709544181824,
        0.7467237710952759,
        0.2247271090745926,
       -0.6747357845306397,
       -0.16039365530014039,
       -0.41555172204971316,
       -0.04566565155982971,
        0.21260707080364228,
        0.2549103796482086,
        0.24795542657375337,
        0.5625612735748291,
        0.8036459684371948,
        0.15800043940544129,
        0.04797195643186569,
       -0.15839435160160066,
       -0.06506697088479996,
       -0.2577322721481323,
        0.3262946903705597,
        0.5458049178123474,
        0.616370439529419,
       -0.35092639923095705,
        0.048758912831544879,
        0.11522434651851654,
        0.04175107553601265,
       -0.12269306182861328,
        0.1227836161851883,
        0.4020257890224457,
        0.07093577086925507,
       -0.1880340874195099,
        0.5334663391113281,
        0.46888044476509097,
        0.18104688823223115,
        0.30756646394729617,
        0.29316428303718569,
       -0.10604366660118103,
        0.44999250769615176,
        0.18227706849575044,
        0.5962150692939758,
        0.38278165459632876,
       -0.40461188554763796,
        0.17775404453277589,
       -0.16349074244499207,
        0.06950787454843521,
        0.7547341585159302,
       -0.4842711389064789,
        0.4062837064266205,
        0.09000574052333832,
        0.03859427571296692,
        0.24143263697624207,
       -0.3383118510246277,
        0.3363209366798401,
        0.10778547078371048,
        0.3429640233516693,
       -0.20395530760288239,
        0.011477324180305004,
        0.6145590543746948,
       -0.5488739609718323,
       -0.26194247603416445,
       -0.09723474085330963,
       -0.19020821154117585,
       -0.18068274855613709,
        0.1601778119802475,
        0.038950759917497638,
        0.6372026205062866,
       -0.12897184491157533,
        0.10720998793840409,
        0.13482464849948884,
       -0.07540713250637055,
       -0.0881727784872055,
        0.5626690983772278,
       -0.31975486874580386,
       -0.029084375128149987,
        0.43618619441986086,
        0.32975345849990847,
       -0.4053913652896881,
        0.15788795053958894,
       -0.3212168216705322,
       -0.20272433757781983,
       -0.8973743319511414,
        0.060059018433094028,
       -0.014103145338594914,
       -0.3387225568294525,
       -0.49839726090431216,
       -0.011007139459252358,
       -0.16101065278053285,
       -0.20850643515586854,
        0.4891682267189026,
        0.33551496267318728,
       -0.23595896363258363,
       -0.4257577359676361,
       -0.48884832859039309,
        0.48760101199150088,
        0.34031161665916445,
        0.1722799688577652,
       -0.35575979948043826,
        0.629051923751831,
       -0.8014369010925293,
        0.575096607208252,
        0.421142578125,
       -0.2668846547603607,
       -0.046029768884181979,
        0.2791147530078888,
       -0.22112232446670533,
        0.02008579671382904,
        0.22087614238262177,
       -0.17961964011192323,
        0.4235396981239319,
        0.295818567276001,
       -0.18260923027992249,
        0.3227207660675049,
        0.11412205547094345,
        0.04591478034853935,
        0.5127033591270447,
        0.428005576133728,
        0.20718106627464295,
        0.18405631184577943,
       -0.22416146099567414,
        0.4277373254299164,
        0.5384698510169983,
        0.04109276458621025,
        0.5105301141738892,
        0.473961740732193,
       -0.6853302717208862,
       -0.16557902097702027,
       -0.12704522907733918,
        0.0026600745040923359,
        0.5272349715232849,
        0.12121742218732834,
        0.427141010761261,
       -0.3047095239162445,
        0.5948843359947205,
        0.335798442363739,
        0.35749775171279909,
       -0.18497343361377717,
        0.26501506567001345,
        0.1564970314502716,
        0.4210122525691986,
       -0.1915784478187561,
        0.05715287476778030 7,
       -0.28498271107673647,
        0.04969947412610054,
        0.7697478532791138,
        0.5546697974205017,
        0.0958070456981659,
       -0.3533228933811188,
        0.4784282147884369,
        0.624963104724884,
        0.2151053100824356,
        0.17361000180244447,
        0.22527147829532624,
       -0.12481484562158585,
        0.4212929904460907,
       -0.2926572859287262,
        0.2562543749809265,
        0.38751208782196047,
        0.1340814083814621,
        0.0680900365114212,
        0.2952287793159485,
        0.12217980623245239,
       -0.2869758605957031,
        0.15682946145534516,
       -0.022066200152039529,
       -0.09002991020679474,
       -0.2826828360557556,
        0.84619140625,
        0.7544476985931397,
        0.5953861474990845,
        0.6517565250396729,
       -0.07932830601930618,
        0.22802823781967164,
       -0.135965958237648,
       -0.8263510465621948,
       -0.6325801610946655,
       -0.5928561091423035,
        0.4108763635158539,
        0.0964483916759491,
       -0.5045000910758972,
       -0.06772734969854355,
       -0.79107666015625,
        0.060380879789590839,
        0.015578197315335274,
        0.32540079951286318,
       -0.044692762196063998,
       -0.17132098972797395,
       -0.19123415648937226,
        0.17911623418331147,
        0.3269428014755249,
       -0.22874118387699128,
        0.4686919152736664,
       -0.15749554336071015,
   -0.25185921788215639,
   -0.21561351418495179,
   -0.10132477432489395,
   -0.057977184653282168,
    0.09759098291397095,
    0.16202516853809358,
    0.01888692006468773,
    0.1724688857793808,
   -0.3449697196483612,
    0.4449881315231323,
    0.10185430943965912,
   -0.297672629356384З,
    0.06075461208820343,
    0.21909406781196595,
   -0.07409229874610901,
    0.6881160140037537,
    0.17447273433208466,
   -0.048471711575984958,
    0.5318611264228821,
    0.30954766273498537,
   -0.24350836873054505,
    0.14386573433876038,
   -0.10827953368425369,
    0.08575868606567383,
    0.14200334250926972,
    0.5095603466033936,
   -0.025056177750229837,
    0.24901045858860017,
   -0.23696841299533845,
   -0.03630203381180763,
    0.45206722617149355,
    0.5019969344139099,
   -0.21705971658229829,
   -0.08452687412500382,
   -0.10376924276351929,
   -0.3200875520706177,
   -0.2048267275094986,
   -0.2703971266746521,
    0.2925371825695038,
    0.3755778670310974,
    0.2522588074207306,
    0.22964833676815034,
    0.7995960116386414,
    0.12206973880529404,
    0.2896155118942261,
    0.04163726791739464,
   -0.12602514028549195,
    0.004978220444172621,
    0.3399927020072937,
    0.09124521911144257,
   -0.5452605485916138,
    0.2247130423784256,
    0.23503662645816804,
    0.06750215590000153,
   -0.2884872257709503,
   -0.2791622579097748,
   -0.1780446618795395,
   -0.44350507855415347,
   -0.1840016394853592,
    0.8970789909362793,
   -0.3687478303909302,
    0.36603569984436037,
    0.23560358583927155,
    0.020292289555072786,
    0.2446030080318451,
    4.3314642906188969,
    0.194863960146904,
   -0.10218192636966706,
    0.5695234537124634,
    0.016988292336463929,
   -0.15768325328826905,
    0.050476688891649249,
    0.09948820620775223,
   -0.06554386019706726,
    0.22301962971687318,
   -0.05468735471367836,
    0.29051196575164797,
    0.12100572139024735,
    0.4127441644668579,
    0.1667146235704422,
    0.0587792843580246,
   -0.09758614003658295,
   -0.20510408282279969,
   -0.21746976673603059,
    0.43335747718811037,
   -0.32159093022346499,
    0.6942153573036194,
    0.6173154711723328,
    0.3104712665081024,
    0.5751503109931946,
    0.4174514412879944,
   -0.2948107421398163,
    0.3532458245754242,
    0.4869029223918915,
    0.3115881681442261,
    0.28135108947753909,
    0.38450825214385989,
    0.016915690153837205,
   -0.11598393321037293,
   -0.32250434160232546,
   -0.06988134980201721,
    0.22417351603507996,
   -0.35582518577575686,
    0.2677224576473236,
    0.008019124157726765,
   -0.19177919626235963,
    0.5731900334358215,
   -0.03540642186999321,
    0.43302130699157717,
    0.1796148121356964,
   -0.005056577268987894,
    0.37953320145606997,
    0.13488957285881043,
    0.7240068912506104,
   -0.3088097870349884,
    0.5610846281051636,
   -0.29582735896110537,
   -0.20909856259822846,
   -0.2881403863430023,
    0.10329002141952515,
    0.49255961179733279,
    0.14558906853199006,
    0.41020694375038149,
    0.04002099484205246,
   -0.24476903676986695,
   -0.389543354511261,
    0.3901459574699402,
    0.6170359253883362,
    0.18917717039585114,
   -0.41235554218292239,
   -0.19313344359397889,
   -0.10294703394174576,
    0.5560699105262756,
    0.5773581266403198,
   -0.17282086610794068,
    0.28679269552230837,
    0.34322652220726015,
   -0.07227988541126251,
   -0.5244243741035461,
   -0.26529040932655337,
   -0.11131077259778977,
   -0.19524210691452027,
    0.4082769453525543,
   -0.009217939339578152,
   -0.1462743580341339,
    0.7264918684959412,
   -0.09149657934904099,
   -0.3374916911125183,
   -0.05742226541042328,
   -0.3913151025772095,
    0.7185215950012207,
   -0.3785516619682312,
   -0.00010882654169108719,
    0.6655824780464172,
    0.4194306433200836,
    0.3726831376552582,
   -0.014721312560141087,
    0.5345744490623474,
    0.33022087812423708,
   -0.06344814598560333,
   -0.1560882031917572,
    0.22698232531547547,
   -3.8697707653045656,
    0.06812435388565064,
   -0.4368731677532196,
   -0.07041455805301666,
   -0.015291529707610608,
   -0.41140303015708926,
    0.31612321734428408,
    0.2914712429046631,
   -0.3867192566394806,
   -0.026363473385572435,
   -0.08788029104471207,
   -0.10701339691877365,
   -0.2673511505126953,
    0.27538666129112246,
   -0.3661351501941681,
    0.5879861116409302,
    0.06352981925010681,
    0.15547777712345124,
    0.0863194614648819,
   -0.021183960139751436,
    0.428565114736557,
    0.04859453812241554,
    0.35721391439437868,
   -0.3864029347896576,
   -0.20986808836460114,
    0.15433000028133393,
    0.25567296147346499,
    0.25359275937080386,
   -0.4783596396446228,
   -0.010366495698690415,
    0.4777776598930359,
   -0.029405448585748674,
    0.3631121814250946,
   -0.18738743662834168,
    0.2193489819765091,
    0.7861229777336121,
   -0.01961355283856392,
    0.16653983294963838,
   -0.4193624258041382,
    0.3085209131240845,
   -0.03517897054553032,
   -0.035910699516534808,
    0.37241387367248537,
   -0.13769084215164185,
   -0.08015040308237076,
    0.4384872615337372,
   -0.12396809458732605,
    0.15661391615867616,
   -0.3919837176799774,
   -0.6586825251579285,
    0.5687432885169983,
    0.0396936871111393,
   -0.09660491347312927,
    0.05788198113441467,
    0.48911261558532717,
    0.5213083028793335,
    0.3355415165424347,
   -0.006735790055245161,
   -0.11381038278341294,
    0.09182903915643692,
   -0.11055094748735428,
   -0.28275448083877566,
    0.24975340068340302,
    0.11746659129858017,
   -0.42452141642570498,
   -0.2323901206254959,
   -0.38694220781326296,
    0.015501483343541623,
    0.6440262198448181,
   -0.3121536672115326,
   -0.08778296411037445,
   -0.14549347758293153,
    0.01749151013791561,
   -0.5398207902908325,
    0.4124368131160736,
    0.5154116749763489,
   -0.34769660234451296,
    0.5662841796875,
    0.4989481270313263,
    0.06761053949594498,
    0.014184223487973214,
    0.601079523563385,
   -0.3859538435935974,
    0.3446619212627411,
    2.190366744995117,
    0.4051366150379181,
    2.288928508758545,
    0.5293960571289063,
   -0.3505767583847046,
    0.5397417545318604,
   -0.6520821452140808,
    0.4239364266395569,
    0.2618080675601959,
    0.20174439251422883,
    0.030146604403853418,
    0.0610184520483017,
    0.062213074415922168,
   -0.11276254057884217,
   -0.1301877349615097,
   -0.19404706358909608,
    0.5268515348434448,
   -0.7370991706848145,
    0.028712594881653787,
   -0.4024544954299927,
    0.18225152790546418,
    0.7267741560935974,
   -0.2734072208404541,
    0.1759040206670761,
   -0.2950340211391449,
    0.14166314899921418,
    0.6515365242958069,
   -0.29643580317497256,
   -0.06734377890825272,
    0.09662584215402603,
   -0.010966300964355469,
   -0.3204823136329651,
    0.6417866349220276,
   -0.051218003034591678,
   -0.008819818496704102,
    0.5098630785942078,
   -0.21459998190402986,
    4.437846660614014,
   -0.24779054522514344,
    0.018799694254994394,
   -0.01747281290590763,
   -0.0487254373729229,
    0.6121163964271545,
    0.4686623811721802,
   -0.22926479578018189,
   -0.03692511469125748,
   -0.4286654591560364,
    0.46073317527770998,
    0.16875289380550385,
   -0.014255600981414318,
   -0.07684683054685593,
    0.12223237752914429,
   -0.30599895119667055,
    0.39215049147605898,
    0.22453786432743073,
    0.5624862313270569,
   -0.011985340155661106,
    0.05180392041802406,
    0.030400553718209268,
    0.08391892164945603,
    0.10214067250490189,
   -0.444959074258804 3,
    0.2225639522075653,
    0.3862999975681305,
    0.24732927978038789,
   -0.05571140721440315,
   -0.021564822643995286,
    0.28468334674835207,
    5.213898658752441,
    0.13289497792720796,
   -0.1400047093629837,
   -0.39865049719810488,
    0.12139834463596344,
    0.45539018511772158,
   -0.1865275651216507,
   -0.08270177245140076,
   -0.38520801067352297,
    0.08869948983192444,
   -0.05266271159052849,
    0.14364486932754517,
   -0.2860695719718933,
    0.4430652856826782,
    0.7777798771858215,
    0.21114271879196168,
   -0.358752578496933,
   -0.3664247989654541,
    0.6665846109390259,
   -0.40493687987327578,
    0.1747705042362213,
   -0.06670021265745163,
    0.20972059667110444,
   -0.19101694226264954,
    0.23892535269260407,
   -0.08149895817041397,
    0.018510373309254648,
    0.8112999796867371,
    0.07871513813734055,
    0.09570053964853287,
    0.5030911564826965,
    0.21463628113269807,
   -0.31457462906837466,
    0.3051794767379761,
   -0.39506298303604128,
    0.06605447828769684,
    0.6144300699234009,
   -0.4566810429096222,
     0.3146623373031616,
     0.1887989640235901,
     0.9544244408607483,
     0.5103438496589661,
    -0.4859951138496399,
    -0.32647767663002016,
    -0.07584235072135925,
     0.21474787592887879,
    -0.1920636147260666,
    -0.4472030997276306,
     0.08618132770061493,
    -0.17384092509746552,
    -0.20969024300575257,
    -0.1831870973110199,
     0.8782939314842224,
    -0.15720084309577943,
     0.37347128987312319,
     0.5088165998458862,
     0.29395583271980288,
    -0.3580363988876343,
    -0.17590023577213288,
    -0.508141279220581,
     0.4661521315574646,
     0.142064169049263,
    -0.05615571141242981,
     0.592810869216919,
     0.37807324528694155,
    -0.14052101969718934,
    -0.19951890408992768,
    -0.12800109386444093,
     0.748070478439331,
     0.13753947615623475,
    -0.08446942269802094,
     0.3747580945491791,
    -0.12847286462783814,
    -0.13892321288585664,
     0.08525972813367844,
     0.12516680359840394,
     0.5701874494552612,
    -0.24708901345729829,
     0.0679594948887825,
     0.10870008915662766,
     0.20561885833740235,
    -0.7872452139854431,
   0.07303950190544129,
   0.35694700479507449,
   0.245212584733963,
   0.3299793303012848,
  -0.010669616051018238,
  -0.12047348916530609,
   0.3540535271167755,
   0.32180890440940859,
   0.3066200911998749,
   0.021576205268502237,
   0.17679384350776673,
  -0.23050960898399354,
   0.1292697787284851,
   0.022921407595276834,
   0.5460971593856812,
   0.3612038493156433,
   0.1963733434677124,
   0.4622957706451416,
   0.16855642199516297,
   0.2564740478992462,
  -0.27637141942977908,
  -0.16345584392547608,
   0.08119463175535202,
   0.07851938903331757,
  -0.5181471109390259,
  -0.5290305614471436,
   0.5271350741386414,
   0.3391841650009155,
   0.501441240310669,
   0.740936279296875,
  -0.26713573932647707,
   0.030347898602485658,
   0.05174243822693825
 ],
 "success": true
}
]
}
```

.vectors.remove algorithm

The `.vectors.remove` algorithm is used to remove the embedding from a node.

`.vectors.remove` syntax

```
CALL neptune.algo.vectors.remove(
  [a list of one or more nodes]
)
YIELD node, success
RETURN node, success
```

`.vectors.remove` input

- **a target node list** *(required)* – *type:* `Node[]` or `NodeId[]`.

  The node(s) from which you want to remove the embedding. If an empty list is supplied, the result will be empty.

`.vectors.remove` outputs

The following outputs are returned for each target node, and if the node has an embedding, the embedding is removed:

- **node** – The target node.
- **success** – A Boolean value: `true` indicates that the removal succeded for the node, and `false` indicates that it failed.

`.vectors.remove` query examples

```
CALL neptune.algo.vectors.remove( ["person933"] )
YIELD node, success
RETURN node, success
```

```
MATCH (n: Student)
CALL neptune.algo.vectors.remove(n)
YIELD status
RETURN n, success
```

Sample .vectors.remove output

Here is an example of the output returned by `.vectors.remove` when run against the sample Wikipedia dataset using the following query:

```
aws neptune-graph execute-query \
  --graph-identifier ${graphIdentifier} \
  --query-string "MATCH (n {`~id`: '1'})
                       CALL neptune.algo.vectors.remove(n)
                       YIELD node, success
                       RETURN node, success" \
  --language open_cypher \
  /tmp/out.txt
{
  "results": [
    {
      "node": {
        "~id": "1",
        "~entityType": "node",
        "~labels": [],
        "~properties": {
          "title": "24-hour clock",
          "views": 2450.62548828125,
          "wiki_id": 9985,
          "paragraph_id": 1,
          "url": "https://simple.wikipedia.org/wiki?curid=9985",
          "langs": 30,
          "text": "A time in the 24-hour clock is written in the form hours:minutes
 (for example\\, 01:23)\\, or hours:minutes:seconds (01:23:45). Numbers under 10 have
 a zero in front (called a leading zero); e.g. 09:07. Under the 24-hour clock system\\,
 the day begins at midnight\\, 00:00\\, and the last minute of the day begins at 23:59
 and ends at 24:00\\, which is identical to 00:00 of the following day. 12:00 can only
 be mid-day. Midnight is called 24:00 and is used to mean the end of the day and 00:00
 is used to mean the beginning of the day. For example\\, you would say \"Tuesday at
 24:00\" and \"Wednesday at 00:00\" to mean exactly the same time."
        }
      },
      "success": true
    }
  ]
}
```

Best practices

The following are some general recommendations for working with Neptune Analytics. Use this information as a reference to quickly find recommendations for maximizing performance while using Neptune Analytics.

Contents

openCypher query best practices

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 Analytics, 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 Analytics is equivalent to removing properties. Neptune Analytics 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
```

You can determine if the query is using a cached plan by observing the `plan cache hits:` value in the output of the openCypher explain endpoint.

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())

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 Analytics, 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

Neptune Analytics allows users to explicitly assign IDs on nodes. 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'}]}
```

Neptune Analytics tools and utilities

Neptune Analytics provides tools and utilities that can simplify and automate your work with a graph. Among these are the following:

Neptune Analytics tools

- **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.

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.

Limits for Neptune Analytics

Regions

Neptune Analytics is available in the following AWS Regions:

- US East (N. Virginia): `us-east-1`
- US East (Ohio): `us-east-2`
- US West (Oregon): `us-west-2`
- Asia Pacific (Singapore): `ap-southeast-1`
- Asia Pacific (Tokyo): `ap-northeast-1`
- Europe (Ireland): `eu-west-1`
- Europe (London): `eu-west-2`
- Europe (Frankfurt): `eu-central-1`

Quotas

Your AWS account has default quotas, formerly referred to as limits, for each AWS service. Unless otherwise noted, each quota is Region-specific. You can request increases for some quotas, and other quotas cannot be increased.

To view the quotas for Neptune Analytics, open the Service Quotas console. In the navigation pane, choose **AWS services** and select **Neptune Analytics**.

To request a quota increase, see Requesting a Quota Increase in the *Service Quotas User Guide*. If the quota is not yet available in Service Quotas, use the limit increase form.

Vertex enumeration is not memory bounded

The following quotas and limits apply to Neptune Analytics:

The current implementation of vertex enumeration and counting is not memory bounded. As a consequence, queries such as `MATCH (n) RETURN count(n)` will require a significant amount of memory and, depending on the chosen capacity and dataset shape, may run into out-of-memory exceptions.

Where possible, we recommend replacing such queries with queries that operate on a per-label basis. For instance, a query such as `MATCH (n : Person) RETURN count(n)` will be significantly more efficient, both in terms of memory consumption and memory utilization.

Parameterized openCypher queries not supported for algorithms

Neptune Analytics supports parameterized openCypher queries with the limitation that parameters are not allowed inside algorithms.

For instance, a query such as `CALL neptune.algo.degree($id)` where `$id` is passed in as a parameter is currently not supported.

Size limits on properties, labels and strings

The maximum length of the strings supported is 1,048,062 bytes. The limit would be lower for strings with unicode characters since some unicode characters are represented using multiple bytes.

API reference

The Neptune Analytics API reference is available for more information.