Course Guide
v1.0

Neo4j Administration

Neo4j, Inc. is the graph company behind the leading platform for connected data. The Neo4j graph platform helps organizations make sense of their data by revealing how people, processes and digital systems are interrelated. This connections-first approach powers intelligent applications tackling challenges such as artificial intelligence, fraud detection, real-time recommendations and master data.

More than 250 commercial customers, including global enterprises like Walmart, Comcast, Cisco, eBay and UBS use Neo4j to create a competitive advantage from connections in their data.

Author: Elaine Rosenberg

Contributors: David Allen, Stefan Armbruster, Ryan Boyd, Andrew Bowman, Dana Canzano, David Fauth, Dave Gordon, Michael Hunger, William Lyon, Mark Needham, Jennifer Reif

© 2019 Neo4j. All rights reserved.

Introduction to Neo4j

Table of Contents

About this module

The Neo4j Graph Platform enables developers to create applications that are best architected as graph-powered systems that are built upon the rich connectedness of data.

At the end of this module, you should be able to:

- Describe the components and benefits of the Neo4j Graph Platform.

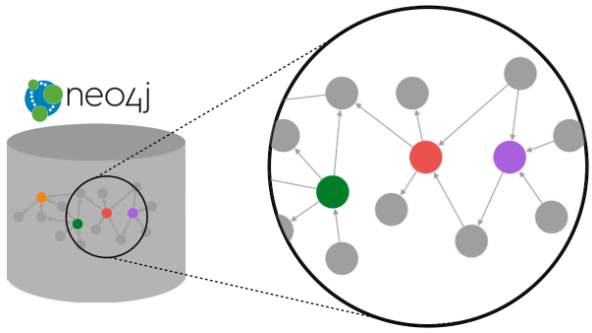
Neo4j Graph Platform

The Neo4j Graph Platform includes components that enable the development of graph-enabled applications. To better understand the Neo4j Graph Platform, you will learn about these components and the benefits they provide.

Neo4j Database

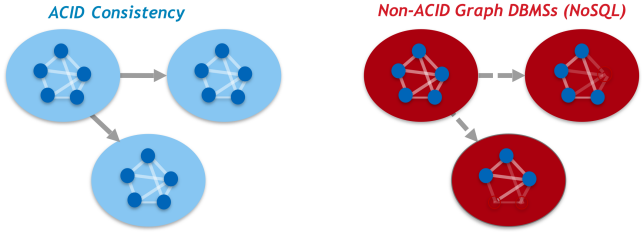
The heart of the Neo4j Graph Platform is the Neo4j Database. The Neo4j Graph Platform includes out-of-the-box tooling that enables you to access graphs in Neo4j Databases. In addition, Neo4j provides APIs and drivers that enable developers to create applications and custom tooling for accessing and visualizing graphs.

Neo4j Database: Index-free adjacency



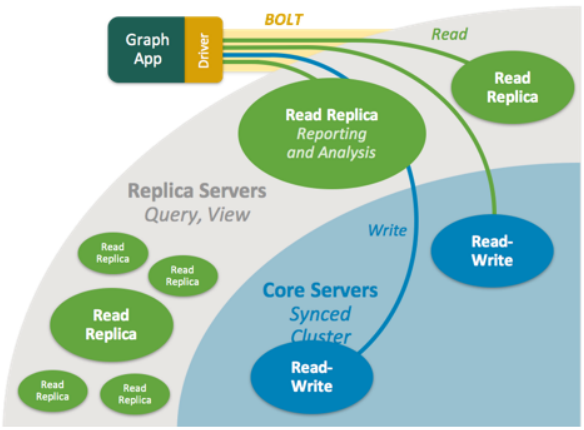
With index free adjacency, when a node or relationship is written to the database, it is stored in the database as connected and any subsequent access to the data is done using pointer navigation which is very fast. Since Neo4j is a native graph database (i.e. it has a graph as its core data model), it supports very large graphs where connected data can be traversed in constant time without the need for an index.

Neo4j Database: ACID (Atomic, Consistent, Isolated, Durable)



Transactionality is very important for robust applications that require an ACID (atomicity, consistency, isolation, and durability) guarantees for their data. If a relationship between nodes is created, not only is the relationship created, but the nodes are updated as connected. All of these updates to the database must all succeed or fail.

Clusters



Neo4j supports clusters that provide high availability, scalability for read access to the data, and failover which is important to many enterprises.

Graph engine

The Neo4j graph engine is used to interpret Cypher statements and also executes kernel-level code to store and retrieve data, whether it is on disk, or cached in memory. The graph engine has been improved with every release of Neo4j to provide the most efficient access to an application's graph data. There are many ways that you can tune the performance of the engine to suit your particular application needs.

Language and driver support

Because Neo4j is open source, developers can delve into the details of how the Neo4j Database is accessed, but most developers simply use Neo4j without needing a deeper understanding of the underlying code. Neo4j provides a full stack that implements all levels of access to the database and clustering layer using Neo4j's published APIs. The language used for querying the Neo4j database is Cypher, an open source language.

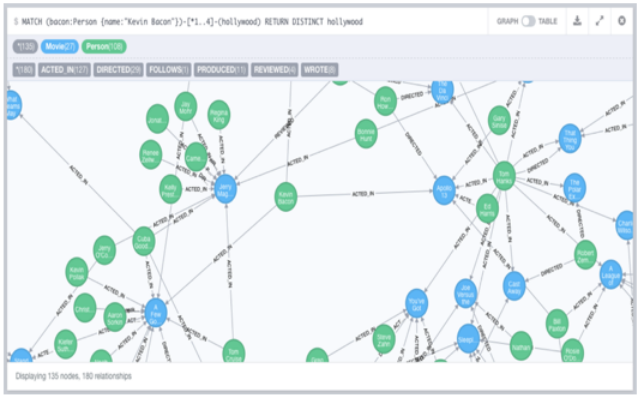
In addition, Neo4j supports Java, JavaScript, Python, C#, and Go drivers out of the box that use Neo4j's bolt protocol for binary access to the database layer. Bolt is an efficient binary protocol that compresses data sent over the wire as well as encrypting the data. For example, developers can write a Java application that uses the Bolt driver to access the Neo4j database, and the application may use other packages that allow data integration between Neo4j and other data stores or uses as common framework such as spring.

It is also possible to develop custom server-side extensions in Java that access the data in the database directly without using Cypher. The Neo4j community has developed drivers for a number of languages including Ruby, PHP, and R. Developers can also extend the functionality of Neo4j by creating user defined functions and procedures that are callable from Cypher.

Libraries

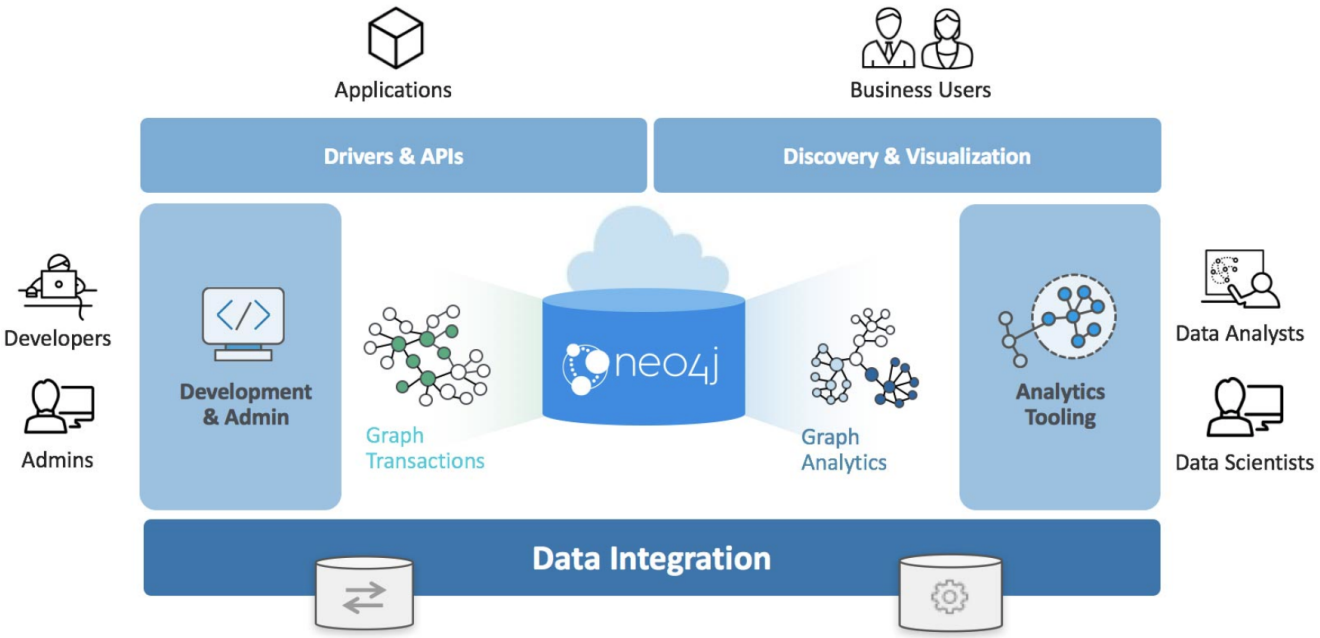
Neo4j has a published, open source Cypher library, Awesome Procedures on Cypher (APOC) that contain many useful procedures you can call from Cypher. Another Cypher library is the Graph Algorithms library, shown here, that can help to analyze data in your graphs. Graph analytics are important because with Neo4j, the technology can expose questions about the data that you never thought to ask. And finally, developers can use the GraphQL library (tree-based subset of a graph) to access a Neo4j Database. These libraries are available as plug-ins to a Neo4j development environment, but there are many other libraries that have been written by users for accessing Neo4j.

Tools



Developers use the Neo4j Browser or a Web browser to access data and test Cypher statements, most of which will be used as part of the application code. Neo4j Browser is an application that uses the JavaScript Bolt driver to access the graph engine of the Neo4j instance. Neo4j also has a new tool called **Bloom** that enables users to visualize a graph without knowing much about Cypher. In addition, there are many tools for importing and exporting data between flat files and a Neo4j Database, as well as an ETL tool.

Neo4j Graph Platform architecture



Here is the big picture of the Neo4j Graph Platform. The Neo4j Database provides support for graph transactions and analytics. Developers use the Neo4j Desktop, along with Neo4j Browser to develop graphs and test them, as well as implement their applications in a number of languages using supported drivers, tools and APIs. Administrators use tools to manage and monitor Neo4j Databases and clusters. Business users use out-of-the box graph visualization tools or they use custom tools. Data analysts and scientists use the analytics capabilities in the Graph Algorithm libraries or use custom libraries to understand and report findings to the enterprise. Applications can also integrate with existing databases (SQL or NoSQL), layering Neo4j on top of them to provide rich, graph-enabled access to the data.

Check your understanding

Question 1

What are some of the benefits provided by the Neo4j Graph Platform?

Select the correct answers.

- ▢ Database clustering
- ▢ ACID
- ▢ Index free adjacency
- ▢ Optimized graph engine

Question 2

What libraries are available for the Neo4j Graph Platform?

Select the correct answers.

- ▢ APOC
- ▢ JGraph
- ▢ Graph Algorithms
- ▢ GraphQL

Question 3

What are some of the language drivers that come with Neo4j out of the box?

Select the correct answers.

- ▢ Java
- ▢ Ruby
- ▢ Python
- ▢ JavaScript

Summary

You should now be able to:

- Describe the components and benefits of the Neo4j Graph Platform.

Overview of Neo4j Administration

Table of Contents

About this module

As a Neo4j database administrator, you will be responsible for ensuring that the deployed Neo4j application runs to meet the needs of its users. This includes ensuring that the Neo4j software is up-to-date and configured properly, monitoring the use of Neo4j, performing life cycle activities such as backups, and managing multiple Neo4j instances.

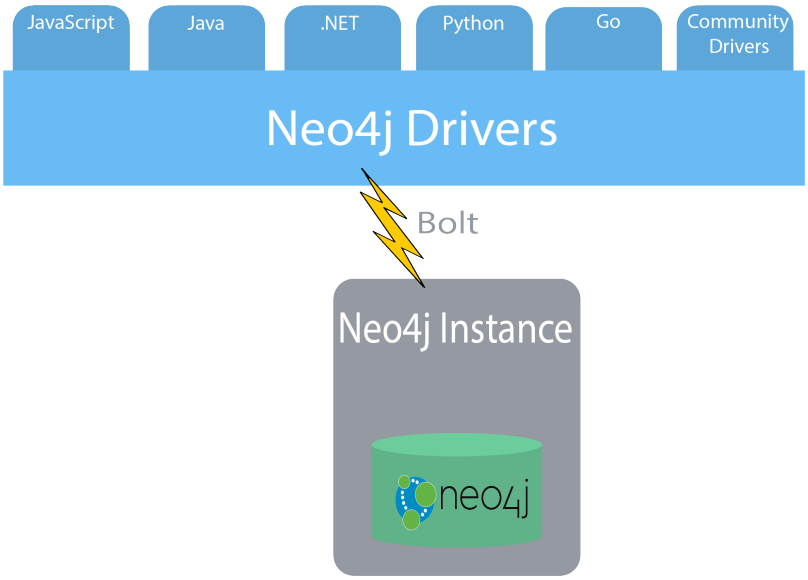
At the end of this module, you should be able to:

- Describe common application architectures that use Neo4j.
- Determine which edition of Neo4j to use.
- Download a specific Neo4j version.
- Determine which deployment option to use.
- Describe the administrative tasks for Neo4j.
- Install Neo4j Enterprise Edition.

Common application architectures with Neo4j

Neo4j can be used in a range of different application architectures. It is a transactional, real-time store for highly connected data.

Applications connect to Neo4j using the binary **Bolt** protocol, which is supported by the official language drivers. Currently the drivers for .NET, Java, JavaScript, Python, and Go are officially supported by Neo4j. There are also a set of other language drivers written by the Neo4j community.



Extending Neo4j

Some applications use Neo4j out-of-the-box, and others extend it with additional functionality from libraries that provide specialized procedures. Examples of these libraries include:

- Neo4j-supported procedures:
  - Graph Algorithms
  - GraphQL
- Community-supported procedures, for example the Awesome Procedures of Cypher (APOC)
- Custom-developed procedures for your application

Regardless of which library your application requires, you must ensure that Neo4j is configured to know how to access these libraries.

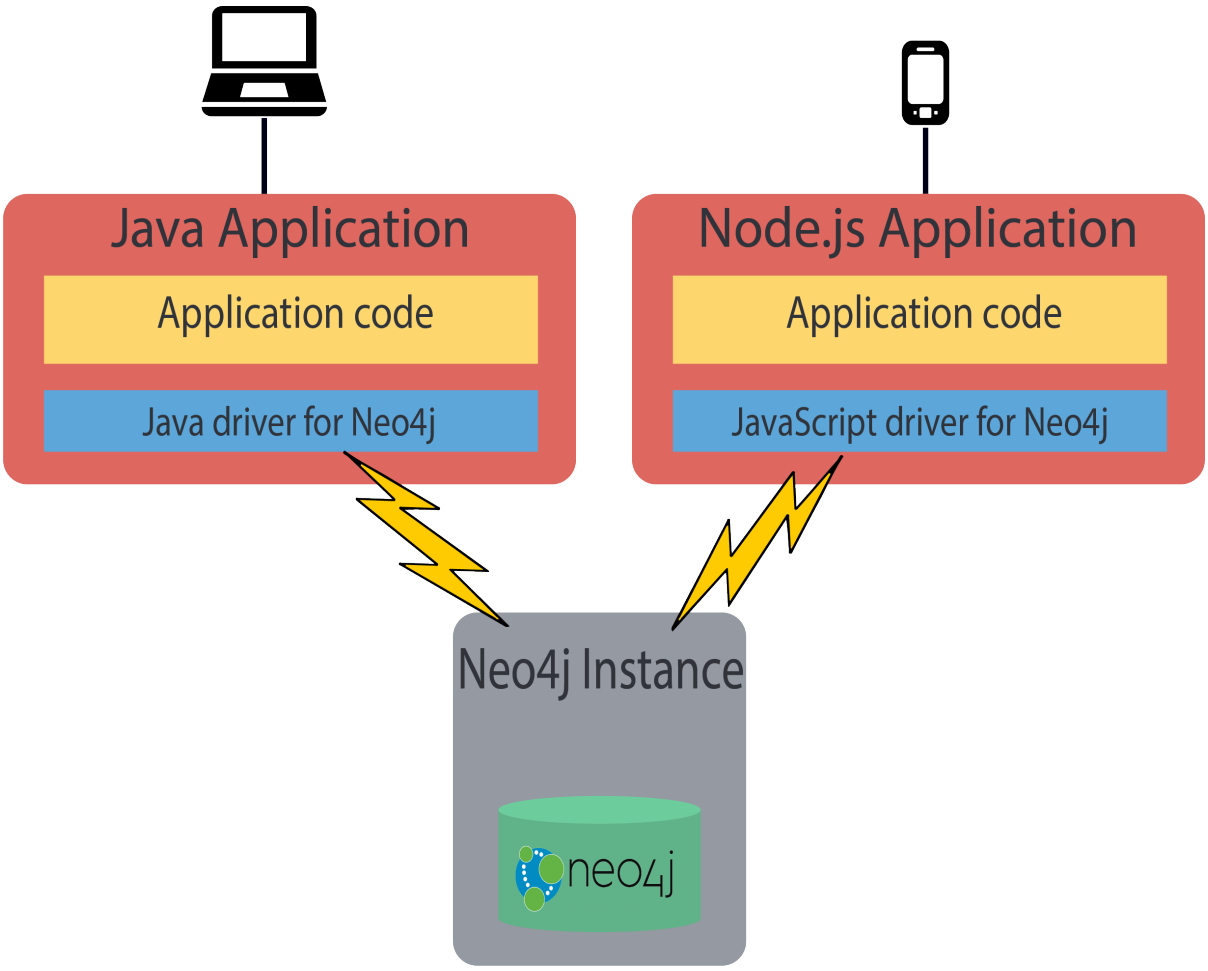
Your application will include at least one Neo4j database, but it could include multiple databases, each with their own Neo4j instance. A Neo4j instance can only support one active database.

Many deployed applications use Causal Clustering which you will learn about later in this training. Causal Clustering is an architecture which provides data safety and performance for critical systems that could be distributed all over the world.

Example architecture #1: Using Neo4j as the primary database

Some applications are developed using data that is stored in a single graph. They can be developed using any language for which there is a driver that can connect to Neo4j.

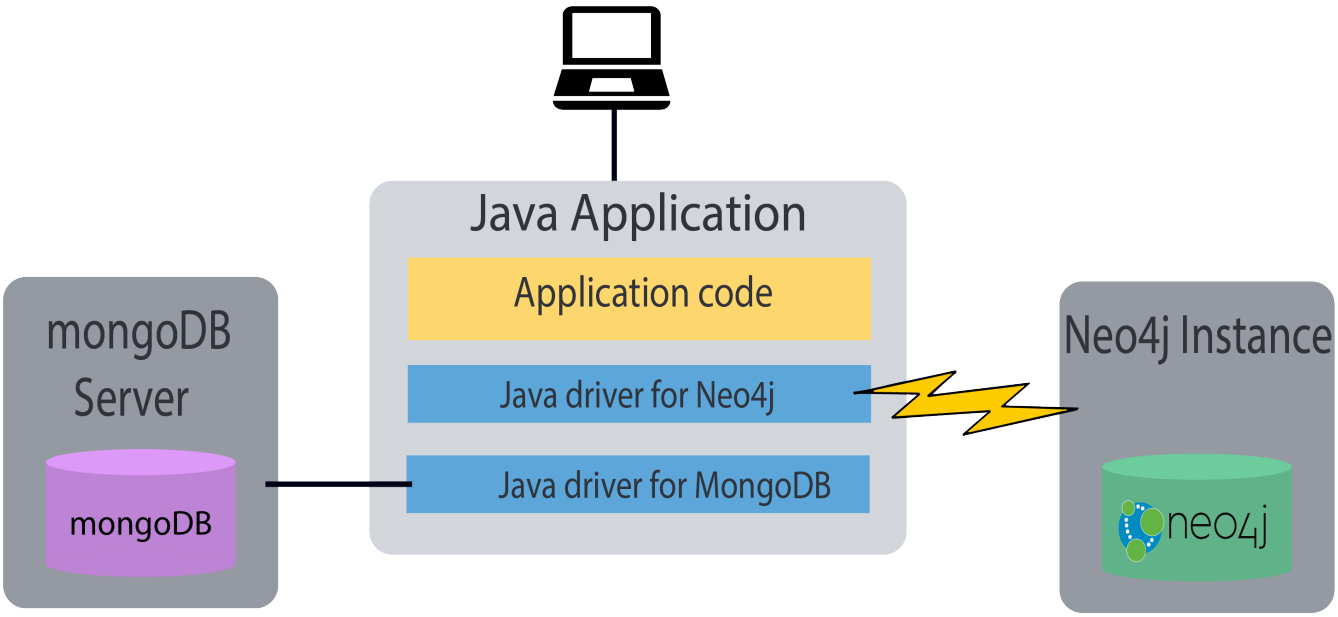
Each application (Neo4j client) communicates with Neo4j to access the graph using the Bolt or HTTP protocol. Neo4j-supported and some community drivers use the Bolt protocol. The clients can be JVMs, .NET resources, a Web server, all of which can be accessed by an end-user.



As a Neo4j database administrator, you are responsible for the availability and consistency of the Neo4j database.

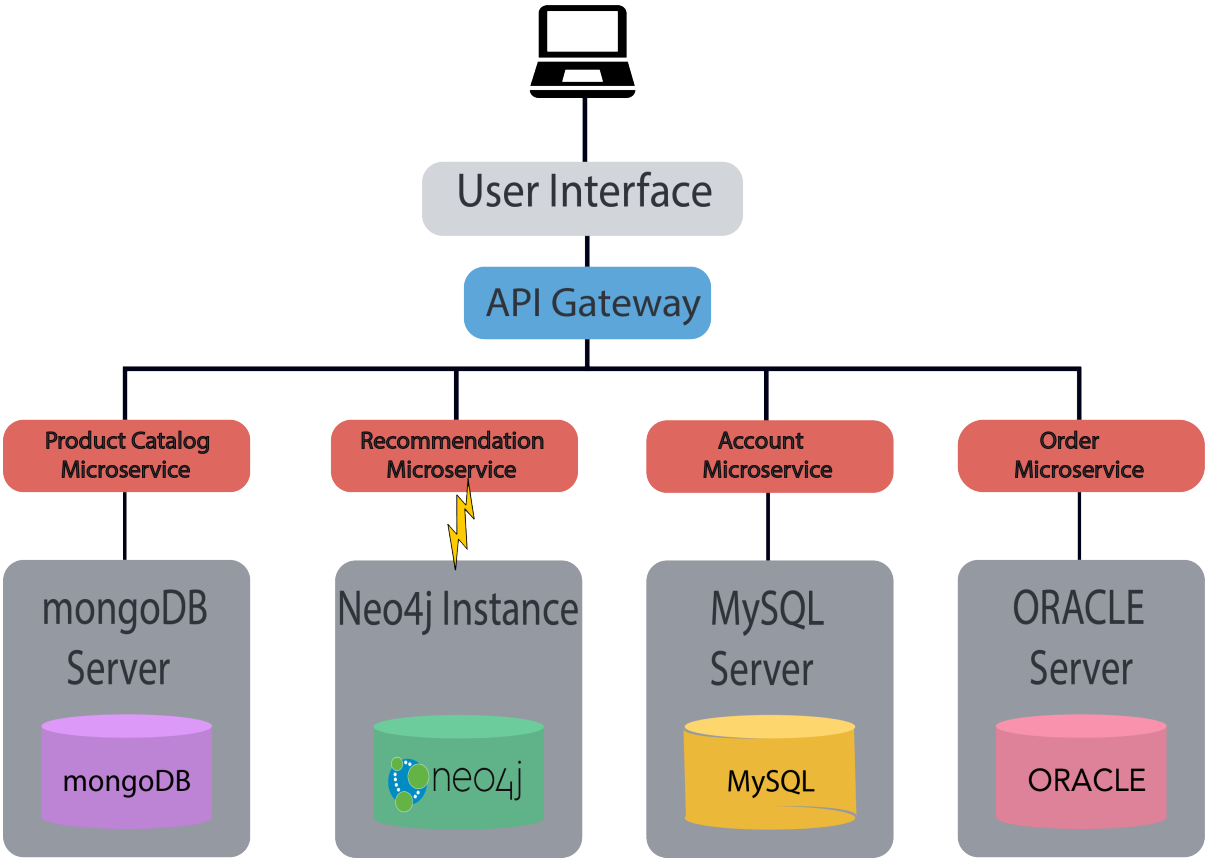
Example architecture #2: Integrating Neo4j with other databases

Neo4j is often used in combination with other data sources (polyglot persistence), such as relational or NoSQL databases. Data can be fetched from the different systems in order to be manipulated by an application and subsequently written back to Neo4j. Other solutions can involve loading of data from a number of databases into Neo4j in order to efficiently do connected data analysis on the combined data set.



Microservices

With cloud-hosted microservices, the implementation chosen for the microservice is one that performs best for the type of transaction. In a cloud-based microservice architecture, there may be a number of services that Neo4j can provide to the end-users.

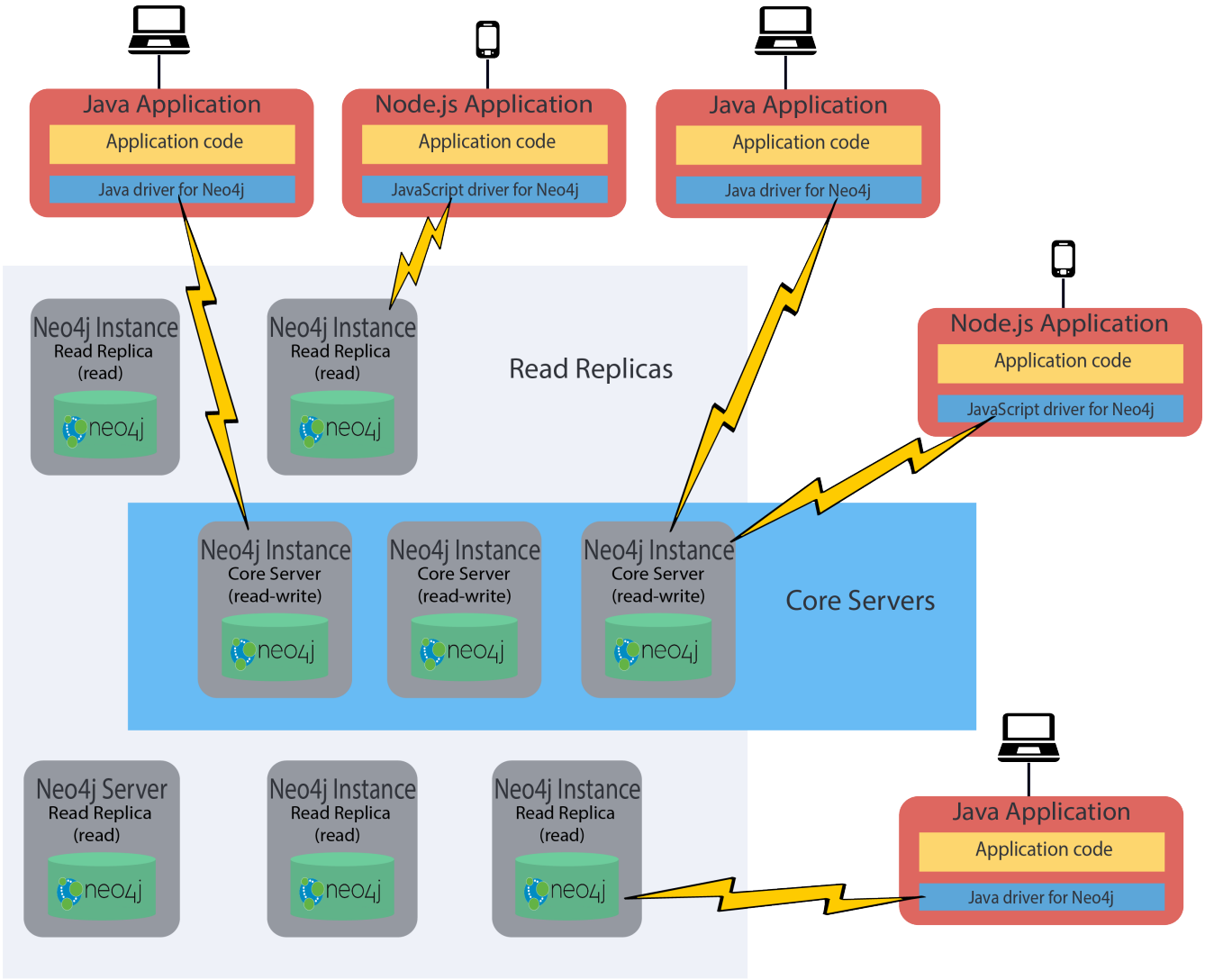


As a Neo4j database administrator, you are responsible for the availability and consistency of the Neo4j database, and for making the data accessible by applications. Application developers are responsible for the logic in transferring and manipulating data between the databases.

Example architecture #3: Using Neo4j Causal Clustering

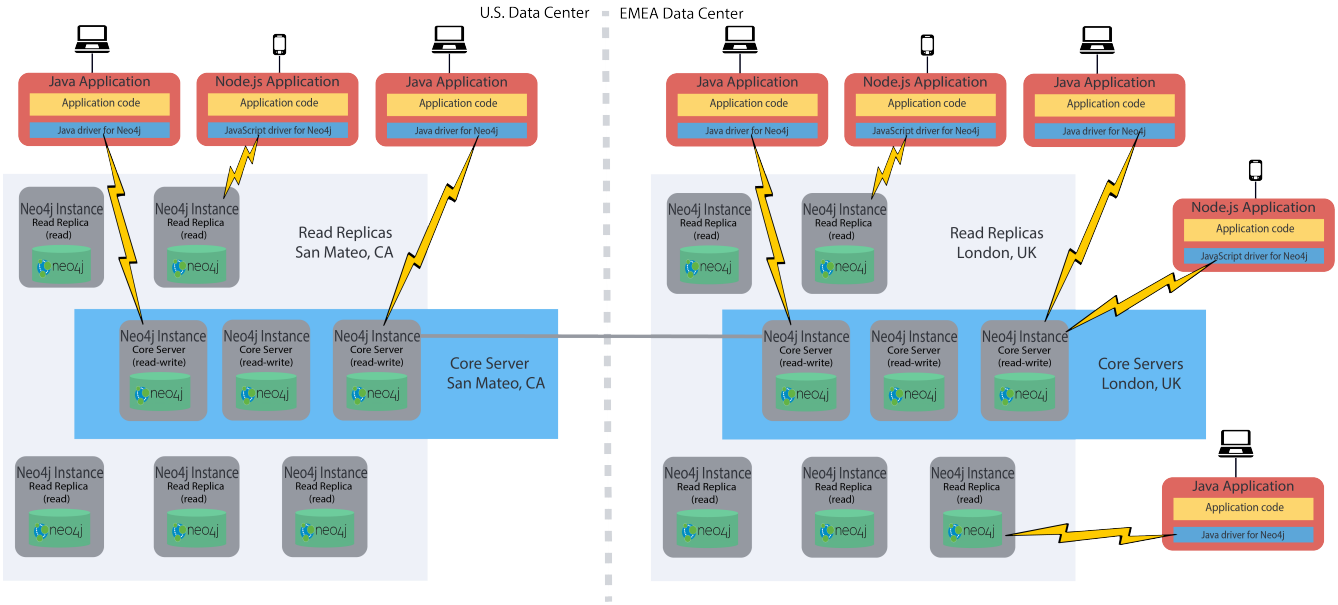
Causal Clustering is used in production environments where high throughput, continuous availability, and reliability are important factors. This feature is available with Neo4j Enterprise Edition.

With Causal Clusters, you configure multiple Neo4j instances that communicate with each other about updates to the database. Causal Clusters are used when data needs to reside in multiple physical locations, or if you want to implement a high availability architecture where access to data will not be affected if a Neo4j instance goes down.



Distributed servers in a cluster

A common use for Neo4j is when different data is required in a set of geographic locations. For example, an online retailer may have different data in the US and Europe. Some data could be shared between different geographically located data centers, but the most heavily updated data needs to physically reside closer to the end-user applications. Neo4j Causal Clusters can be configured to span geographic locations.



As a Neo4j database administrator, you will be responsible for configuring and monitoring Causal Clusters. You will work with architects to determine the appropriate configuration of the Causal Cluster, taking into account aspects such as: uptime requirements, performance requirements, and redundancy required in order to handle events of Neo4j instance failure or data center failure.

Architecting distributed servers

Some decisions you may need to make are:

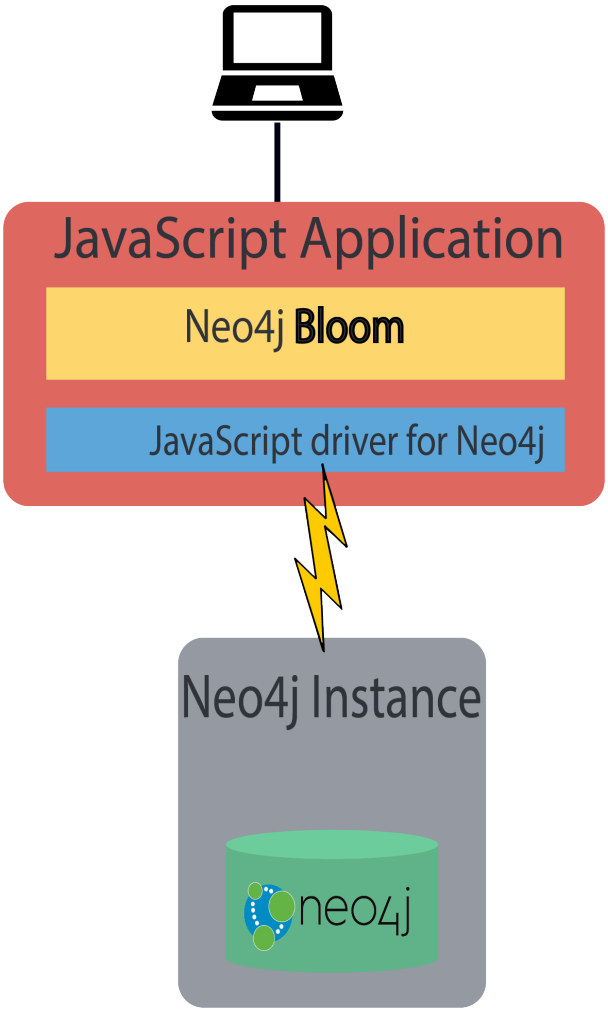
- Number and locations of data centers
- Number and location of Core Servers and Read Replicas
- Sizing of servers (hardware and CPU)
- How to route requests in order to obtain optimal performance
- Which servers to use as backup servers

We will cover the configuration, management and monitoring of Causal Clusters in depth later in this training.

Example architecture #4: Using Neo4j standalone for graph analytics

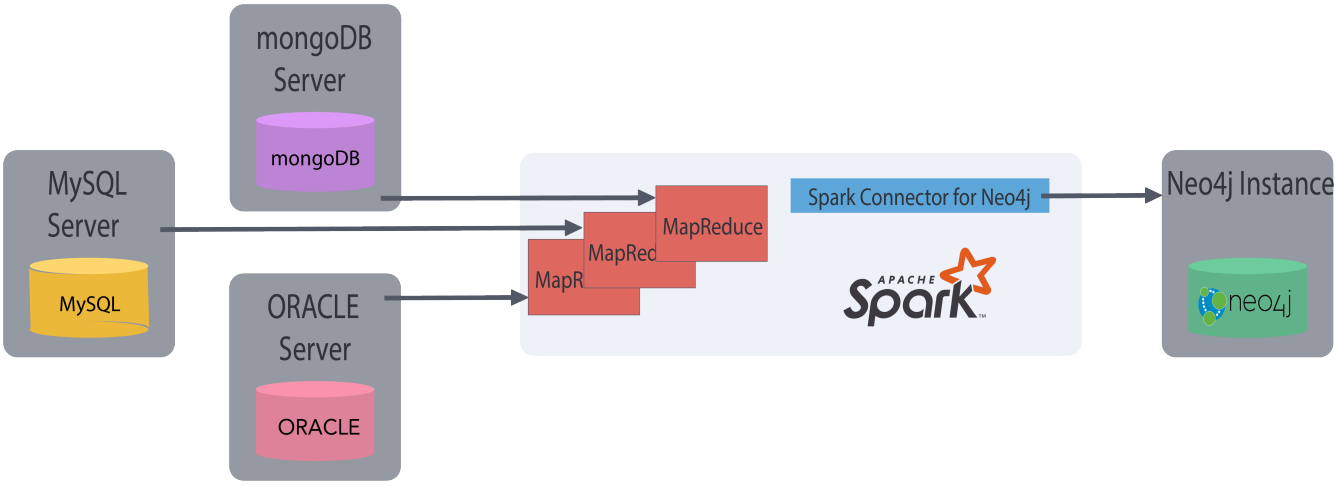
A common use case for Neo4j is for data scientists analyze complex patterns in connected data. Neo4j Bloom, or some other data visualization tool, may be used as the front-end. The following are two common patterns this use case:

1. Neo4j is used to find and analyze connections in data from other data sources. To cater for this, the standalone Neo4j database is loaded with data from several sources. As a Neo4j database administrator, you will ensure that the Neo4j database used for this purpose is kept up-to-date and that the data is secure and appropriately backed up.
2. Data scientists analyze the data in a Neo4j production database. In order to safeguard the production database from potentially heavy queries, a dedicated Read Replica is configured (Causal Cluster) . In this case, the analytics database is always up-to-date with the production database, and its administration is a part of the regular Causal Cluster maintenance work (see Example architecture #3).



Example architecture #5: Neo4j integrated with compute-heavy solutions

Some enterprises need to consolidate large amounts of data for analysis. The data can come from data lakes, NoSQL databases, document stores, relational, all of which is analyzed and placed into a Neo4j database for analytics. A common architecture for streaming and analyzing data for consolidations is Apache Spark.



As a Neo4j database administrator, you will work with architects to ensure that the Neo4j database in this type of environment is properly configured and available to the computational engines that will write to the graph.

Neo4j versions

All supported versions of Neo4j are available on the Neo4j download page. On the same page, you can also find pre-releases of the next release. These offer an opportunity to explore coming features. However, it is important to note that functionality in preview releases can be changed without notice. Additionally, the preview releases are not certified for production use.

General Availability releases are our certified releases. They include all features and functionalities intended for that version and supported for production deployments. Production deployments should only use *General Availability* releases.

WARNING	Data migration between non *General Availability* versions is unsupported.

Neo4j Editions

There are two versions of Neo4j to choose from: Community Edition and Enterprise Edition. The version you use will depend on the features you require, the nature of your application that uses Neo4j, and the level of professional support you would like to receive from Neo4j.

A full comparison between the Community and Enterprise Editions for the current release of Neo4j can be found at Compare Community and Enterprise Editions.

Community Edition

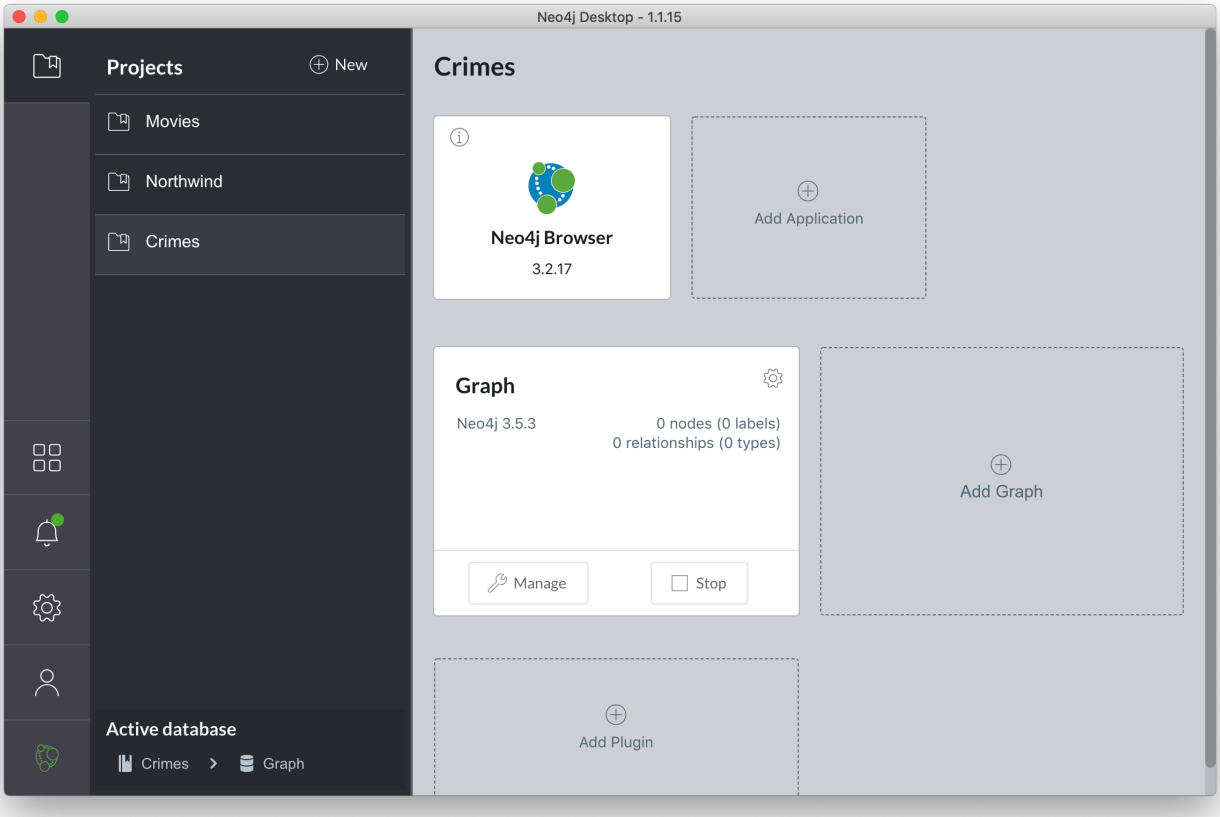
The Community Edition is a full functioning version of Neo4j suitable for single instance deployments. It has full support for key Neo4j features, such as ACID compliance, Cypher, and access via the binary protocol and HTTP APIs. It is ideal for smaller internal or do-it-yourself projects that do not require high levels of scaling or professional services and support. The Community Edition is free to download and use and is available from the Neo4j download page or from the Neo4j GitHub repository.

Enterprise Edition

The Enterprise Edition extends the functionality of the Community Edition to include key features for performance and scalability such as a clustering architecture for high availability and online backup functionality. It is the right choice for production systems with availability requirements or needs for scaling up or out. Enterprise Edition requires a license from Neo4j. You can download Neo4j Enterprise Edition from the Neo4j download page. When you install Neo4j Enterprise Enterprise, you have a 30 day evaluation license.

Neo4j Desktop

Neo4j Desktop is used by developers who will be deploying their Neo4j applications. It's purpose is to make development easier by providing access to supported plugins and graph visualization using the Neo4j Browser. The Neo4j instance that runs in Neo4j Desktop is Enterprise Edition so that developers have access to all of the functionality required for developing and testing a production Neo4j application. Developers can install Neo4j Desktop on Linux, OS X, and Windows. Developers can download Neo4j Desktop from the Neo4j download page. Neo4j Desktop is free for developers to use. Developers need to ensure that the version of Neo4j they use for development is the version that will be used in a production environment.



Installing Neo4j

After you have determined which Edition and version of Neo4j you must install for your application, you should follow the steps outlined in the Neo4j Operations Manual. The instructions include actions that must be taken before installing Neo4j. Later in this training, you will learn how to get started managing and monitoring Neo4j Enterprise Edition.

Upgrading Neo4j

The procedure for upgrading your Neo4j installation will depend upon what release you are upgrading from and to. If you are upgrading to a major release, the upgrade may include a data migration step. The Neo4j Operations Manual provides instructions for upgrading a Neo4j installation.

Supported software and hardware

To install and use Neo4j, the system(s) that host Neo4j Enterprise Edition have specific requirements for:

- JVM
- Operating system
- Hardware architecture
- Memory
- Disk
- Filesystem

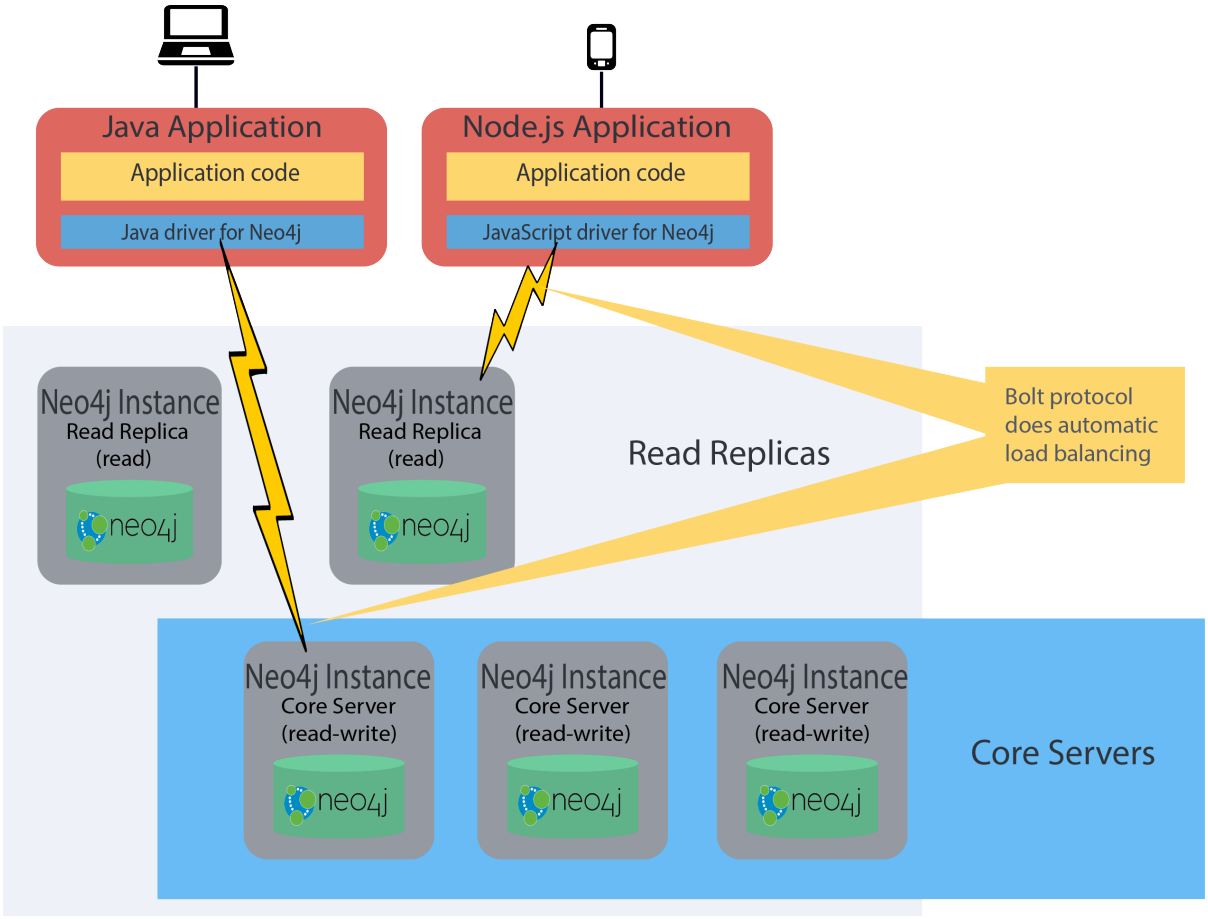
You should consult the System Requirements to learn more.

Neo4j deployment options

Early in the development of an application, a decision will be made whether to deploy Neo4j instance(s), embedded within the application, or to deploy Neo4j in the Cloud.

Server mode deployments

This is the most common deployment, and it is recommended for all onsite deployments that do not require embedded mode. In this architecture, Neo4j runs as a database server and can be accessed through binary and http APIs for data querying and updating.



Key advantages of server mode

- **Binary Bolt Protocol** or HTTP APIs allows clients to send requests and receive responses over the wire.
- When using one of the supported drivers together with Causal Clustering, **load balancing** is provided by Neo4j. It can also be configured to meet specific criteria.
- Supports **Platform independence** for the client/application accessing the server APIs due to dedicated language drivers.
- Can utilize Neo4j and query language extensions via user defined procedures.
- The database is managed independently from the application.
- Neo4j instances are easy to configure and provision in production.

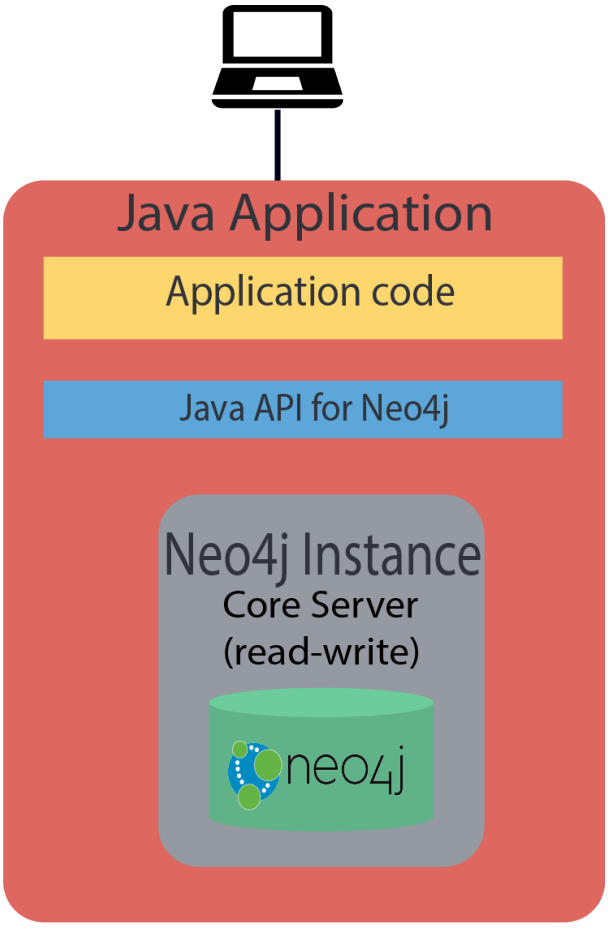
Server mode for the Neo4j administrator

When deployed as a server, you should expect to perform many common administration, configuration, and troubleshooting tasks as you would for any database deployment. Neo4j provides utilities and tools for enabling these capabilities, which you will learn about later in this training. This training teaches you about the configuration that will aid you in ensuring a robust and stable deployment.

This training will primarily emphasize the production decisions and administrative tasks associated with a server mode deployment.

Embedded deployments

This architecture is more common in OEM setups, where Neo4j runs integrated as part of a third party product. When Neo4j is embedded directly into an application, you get all the functionality of the database directly accessible through the Java APIs used by the application code. This makes it very easy to directly work with the database and get lightning fast performance.



Key advantages of embedded mode

- Low latency: Since Neo4j is in the application, there will be no network overhead.
- Choice of APIs: Access to a diversity of APIs for creating and querying data via the Neo4j Core API, traversal framework, and the Cypher query language.

For the Neo4j administrator

Your hands-on administration work with Neo4j is very minimal with embedded deployments. Most of the core administration tasks, such as initiating backups, monitoring, and configuration should be built into the application that is embedding Neo4j.

Neo4j in the Cloud

Neo4j has been successfully hosted in these environments:

- Amazon EC2
- Google Cloud (GCP)
- Microsoft Azure
- Kubernetes

To learn more about Neo4j Cloud deployments see the Developer Guide to Cloud Deployment that is updated regularly.

Administrative tasks for Neo4j

In this training, you will learn how to perform these common administrative tasks for a server mode deployment:

- Downloading and installing Neo4j
- Managing a Neo4j database
- Managing plugins used with the Neo4j database
- Managing logging
- Monitoring queries
- Backing up and restoring databases
- Managing Causal Clusters
- Managing database security

Exercise 1: Setting up Neo4j Enterprise Edition on your system

We are using Neo4j Enterprise Edition for the hands-on exercises in this training, as some Neo4j features are not included in Community Edition. Neo4j Enterprise Edition is available for download with a 30 day evaluation license. All screen captures for this training are from an *Amazon EC2 Debian instance* and will vary from what you experience if you are running on a different platform.

Before you begin

If you have installed Neo4j Desktop on your system, you should stop any database that is active and shut down Neo4j Desktop. You will be using Enterprise Edition that enables you to perform all common administrative tasks.

You should refer to the Neo4j Operations Manual for instructions that are specific to installing Neo4j Enterprise Edition on your platform.

Exercise steps:

1. Except for Debian-based distributions, download Neo4j from the Enterprise Edition download page as follows:

a. Select the latest release of Neo4j Enterprise Edition for your platform.

b. You must provide some identifying information to start your 30 day evaluation.

c. You will receive an email with a link for downloading the software, as well as instructions for installing it on your system.

2. Install Neo4j Enterprise Edition following the instructions for your platform. Ensure that the system requirements are met (specifically, the version of Java required).

3. Confirm that files have been installed as described here.

References

You should consult the Neo4J Operations Manual for more information about the installation requirements and procedures for your target platform, as well as details for Neo4j administration tasks.

The Java Developer Manual covers using embedded mode for Neo4j and is intended for developers.

Check your understanding

Question 1

Suppose your organization has two applications that require Neo4j. Each application uses a different Neo4j database and the clients access the database in server mode. For these two databases, how many Neo4j installations are required?

Select the correct answer.

- ▢ one, that will service two databases.
- ▢ two, one Neo4j installation for each database.
- ▢ two, one Core Server, and one Read Replica Server.
- ▢ three, two Core Servers, and one Read Replica Server.

Question 2

Which features below are available only in the Enterprise Edition of Neo4j?

Select the correct answers.

- ▢ ACID transactions
- ▢ Causal Clusters
- ▢ Bolt protocol
- ▢ Online backups

Question 3

What type of process must Neo4j run in?

Select the correct answer.

- ▢ Daemon
- ▢ JVM
- ▢ Docker Container
- ▢ Kubernetes

Summary

You should now be able to:

- Describe common application architectures that use Neo4j.
- Determine which edition of Neo4j to use.
- Download a specific Neo4j version.
- Determine which deployment option to use.
- Describe the administrative tasks for Neo4j.
- Install Neo4j Enterprise Edition.

Managing a Neo4j Database

Table of Contents

About this module

Now that you have installed the Neo4j Enterprise Edition, you will learn how to perform some administrative tasks with the Neo4j instance.

At the end of this module, you should be able to:

- Start a Neo4j instance.
- Stop the Neo4j instance.
- Set the password for the *neo4j* user.
- Copy a Neo4j database.
- Modify the location for a Neo4j database.
- Check the consistency of a Neo4j database.
- Create scripts for modifying a Neo4j database.
- Manage plugins for a Neo4j database.
- Configure ports used by the Neo4j instance.
- Perform an online backup of a Neo4j database.
- Create a database with the import tool.

Neo4j instance files

Depending on your platform, a Neo4j instance's files are, by default, placed as described here. Here is a brief overview of the default folders you will frequently use for managing the Neo4j instance.

Purpose of folder	Description
Tools	The **/usr/bin** folder contains the tooling scripts you will typically run to manage the Neo4j instance.
Configuration	**Neo4j.conf** is the primary configuration file for the Neo4j instance and resides in the **/etc/neo4j** folder.
Logging	The **/var/log/neo4j** folder contains log files that you can monitor.
Database(s)	The **/var/lib/neo4j/data** folder contains the database(s).

Post-installation preparation

In this training, all screenshots and examples are shown using Neo4j Enterprise Edition installed as a Debian package. If your system is different, you will need to adjust file locations as described later in this module.

When you are setting up a production environment, you want to control who can manage the Neo4j instance. You will also want to control when the Neo4j instance starts as you will performing some configuration changes and database operations that may require that the instance to be stopped.

When Neo4j is installed as a Debian package, the *neo4j* service is enabled and the Neo4j instance is automatically started. Other platforms do not start the Neo4j instance automatically.

Post-install: Changing the *neo4j* password (non-Debian)

After you install Neo4j and before you start the Neo4j instance, a best practice is to change the default password for the user *neo4j*. You do this on all platforms, except when you have installed a Debian package. You will learn about changing the *neo4j* password on Debian later in this module.

You change the password for the *neo4j* user by executing the following command:

```
[sudo] bin/neo4j-admin set-initial-password newPassword
```

where *newPassword* is a password you will remember.

Post-install: Debian

Initially and on Debian, you should disable *neo4j* as a service that is started automatically when the system starts. You do this with this command:

```
[sudo] systemctl disable neo4j
```

In addition, you should create the folder **/var/run/neo4j** that is owned by *neo4j:neo4j*. This is where the PID for the currently running Neo4j instance is placed.

Managing the Neo4j instance

When the instance is started, it creates a database named **graph.db** in the default location which is a folder under **/var/lib/neo4j/data/databases**. You can start and stop the instance regardless of whether the *neo4j* service is enabled.

You start, stop, restart, and check the status of the Neo4j instance on Debian as follows:

- `[sudo] systemctl start neo4j`
- `[sudo] systemctl stop neo4j`
- `[sudo] systemctl restart neo4j`
- `[sudo] systemctl status neo4j`

You start, stop, restart and check the status of the Neo4j instance on non-Debian systems as follows:

- `[sudo] bin/neo4j start`
- `[sudo] bin/neo4j stop`
- `[sudo] bin/neo4j restart`
- `[sudo] bin/neo4j status`

When the Neo4j instance starts, it opens the database, and writes to the folders for the database and to the log file.

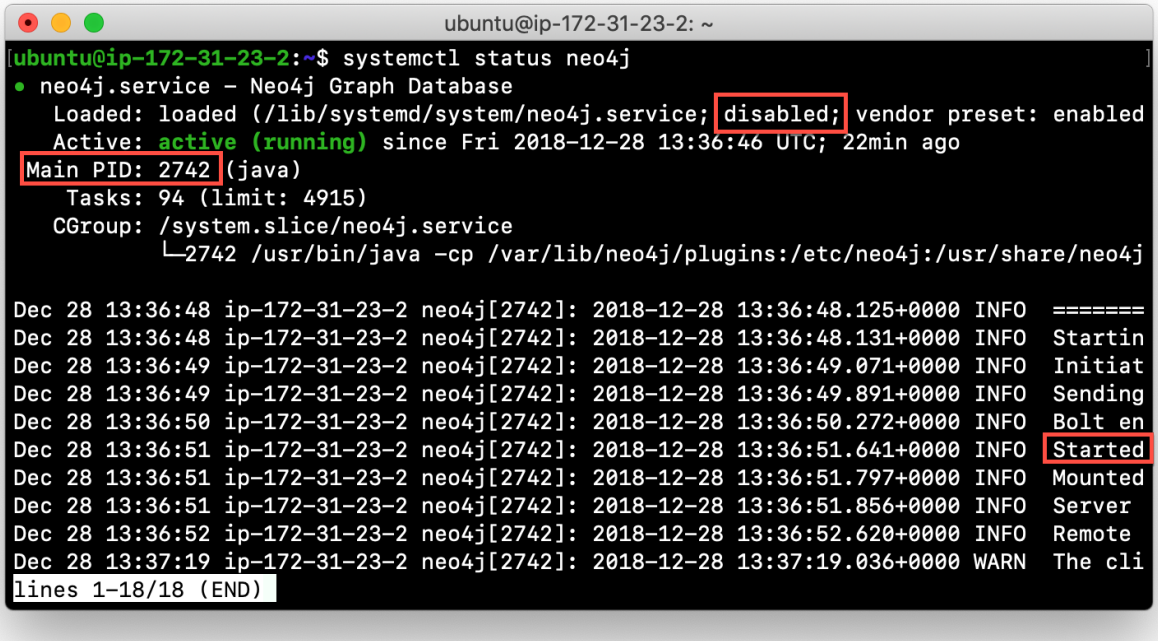
Checking the status of the instance

At any time, you can check the status of the Neo4j instance.

You check the status of the instance as follows:

```
systemctl status neo4j
```

Here is an example where we check the status of the Neo4j instance:



Here we see that the instance is started. Notice that the service is disabled as well. After the instance is started you can identify the process ID (Main PID) from the status command on Debian. It is sometimes helpful to know the process ID of the Neo4j instance (JVM) in the event that it is unresponsive and you must kill it.

However, knowing whether the instance is started (active) is generally not sufficient, especially if you have made some configuration changes. You can view details of the Neo4j instance by examining the log file.

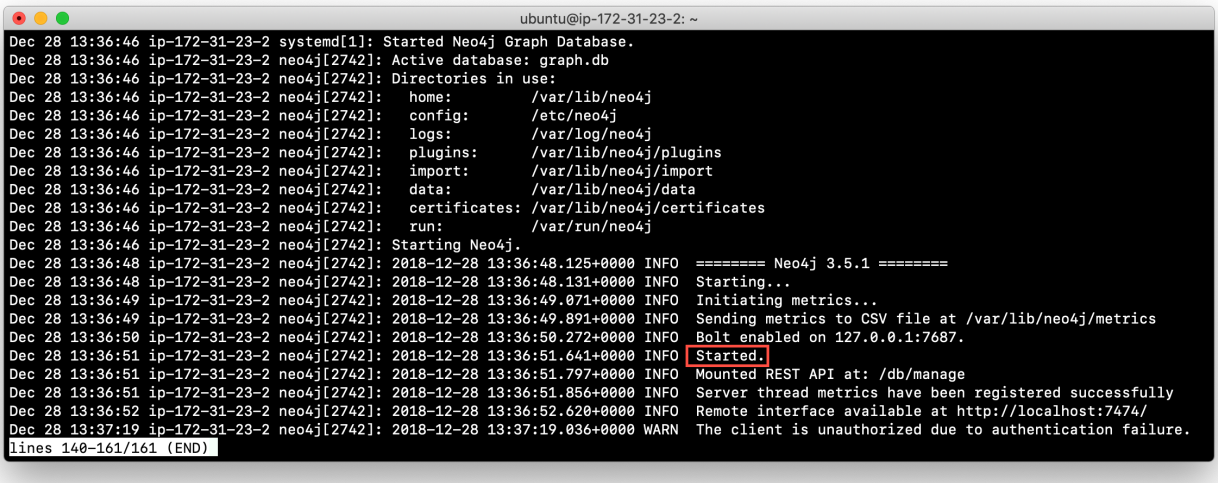
Viewing the neo4j log

The status command gives you a short glimpse of the status of the Neo4j instance. In some cases, although the instance is *active*, it may not have started successfully. You may want to examine more information about the instance, such as the folders it is using at runtime and information about activity against the instance, and especially if any errors occurred during startup. As an administrator, you should become familiar with the types of records that are written to the log files for the Neo4j instance.

You can view the log file for the instance on Debian as follows:

- `journalctl -u neo4j` to view the entire neo4j log file.
- `journalctl -e -u neo4j` to view the end of the neo4j log file.
- `journalctl -u neo4j -b > neo4j.log` where you can view **neo4j.log** in an editor.

Here is the result from `journalctl`:



When the Neo4j instance starts, you can also confirm that it is started by seeing the *Started* record in the log file.

NOTE	You can also view the log file in the **logs** folder on all platforms.

Exercise #1: Managing the Neo4j instance

In this Exercise, you will stop and start the Neo4j instance and view its status and log file.

Before you begin

You should disable the *neo4j* service `[sudo] systemctrl disable neo4j`, if you are using a system that utilizes the *neo4j* service (for example, Debian).

Exercise steps:

1. Open a terminal on your system.
2. View the status of the Neo4j instance.
3. Stop the Neo4j instance.
4. View the status of the Neo4j instance.
5. Examine the Neo4j log file.
6. Examine the files and folders created for this Neo4j instance.

Using `cypher-shell`

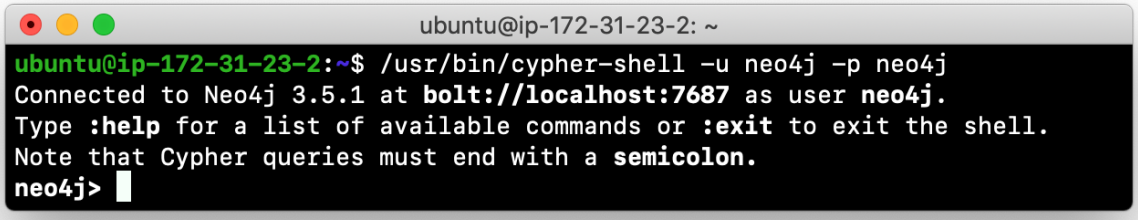
`cypher-shell` enables you to access the Neo4j database from a terminal window. You simply log into the database using `cypher-shell` with your credentials:

```
/usr/bin/cypher-shell -u <username> -p <password>
```

Once authenticated, you enter Cypher statements to execute just as you would in a Neo4j Browser session. One caveat with `cypher-shell`, however is that all Cypher commands must end with `;`. You exit `cypher-shell` with the command `:exit`.

Example: Using `cypher-shell`

Here is an example showing that we can successfully log in to the database for the Neo4j instance, providing the default credentials *neo4j/neo4j*:



NOTE	If you set the environment variables NEO4J_USER and NEO4J_PASSWORD with their respective values, then you need not enter your credentials when logging into `cypher-shell`.

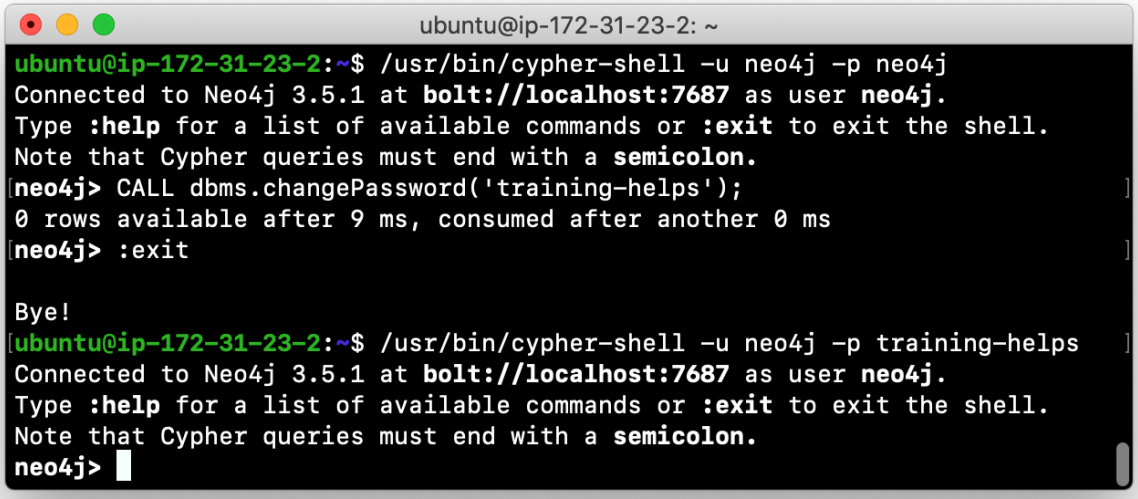
Changing the default password (Debian only)

If we were to attempt to access the database for the first time, we would receive an error. This is because the default credentials *neo4j/neo4j* must be changed. As an administrator, you want to control who can manage this Neo4j instance and its database. To do so, you change the default password for the *neo4j* user. Later in this training, you will learn more about securing Neo4j by managing users and their access.

While logged into the database in `cypher-shell`, you execute the procedure to change the password:

```
CALL dbms.changePassword('newPassword');
```

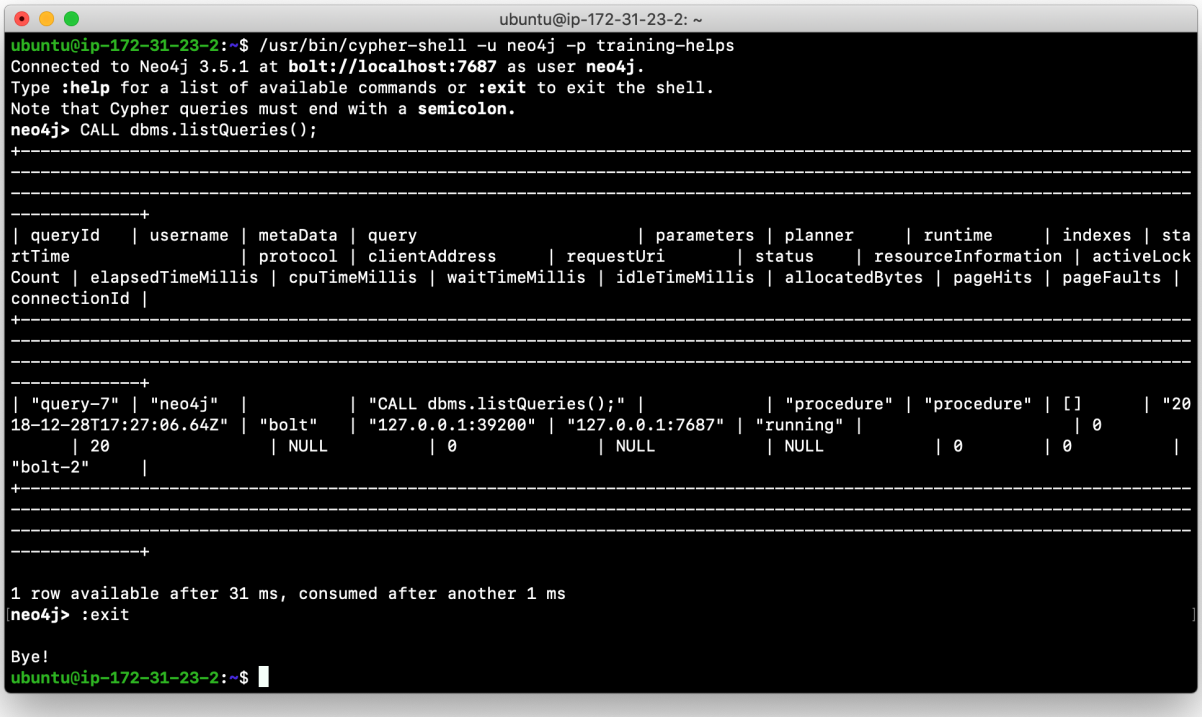
In this example, we log into `cypher-shell` with our credentials. Then we execute the Cypher command to change the password. Finally, we specify `:exit` to log out of `cypher-shell`.



After changing the default password for the Neo4j instance (database), we are now able to access the database after logging in with the new credentials.

Accessing the database

Here is an example where we execute a Cypher statement against the empty database where we list all active queries:



When you are done with `cypher-shell`, you enter `:exit` to exit.

Exercise #2: Using `cypher-shell` to change the password

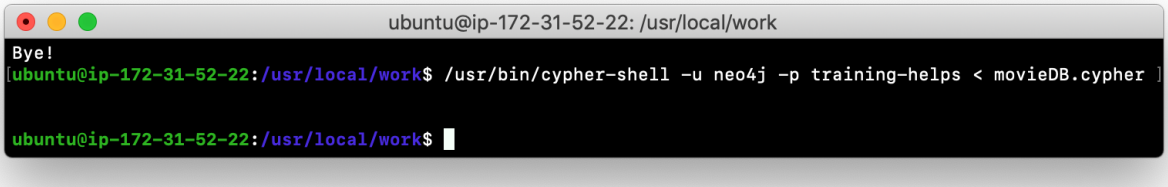
In this Exercise, you will log in to the database with `cypher-shell`, change the password for the database, and execute a Cypher statement to load the database. You can perform this Exercise regardless of the type of system you are using.

Before you begin

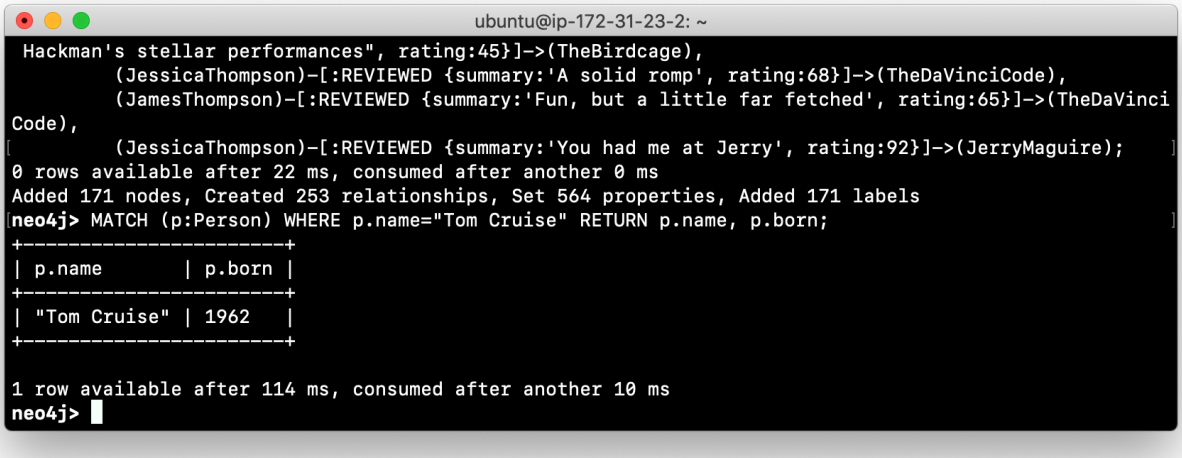
You should ensure that the Neo4j instance is started.

Exercise steps:

1. Open a terminal on your system.
2. Log into the database with `cypher-shell` using the default credentials of *neo4j/neo4j*. (or different credentials if you changed the password previously with)`neo4j-admin set-initial-password`)
3. Execute the Cypher statement, `CALL dbms.listQueries();`. Do you get an error? Note you will not get an error if you previously changed the password.
4. Execute the Cypher statement to change the password to something you will remember.
5. Exit out of `cypher-shell`.
6. Log into the database with `cypher-shell` using the new credentials.
7. Execute the Cypher statement, `CALL dbms.listQueries();`.
8. Exit out of `cypher-shell`.
9. Download this file. This file contains the Cypher statements to load the database with movie data.
10. Invoke `cypher-shell` sending movieDB.cypher as input. You should see something like the following:



11. The database is now populated with the *Movie* data. Log in to `cypher-shell` and execute a Cypher statement to retrieve data from the database, for example: `MATCH (p:Person) WHERE p.name='Tom Cruise' RETURN p.name, p.born;` You should see the following:

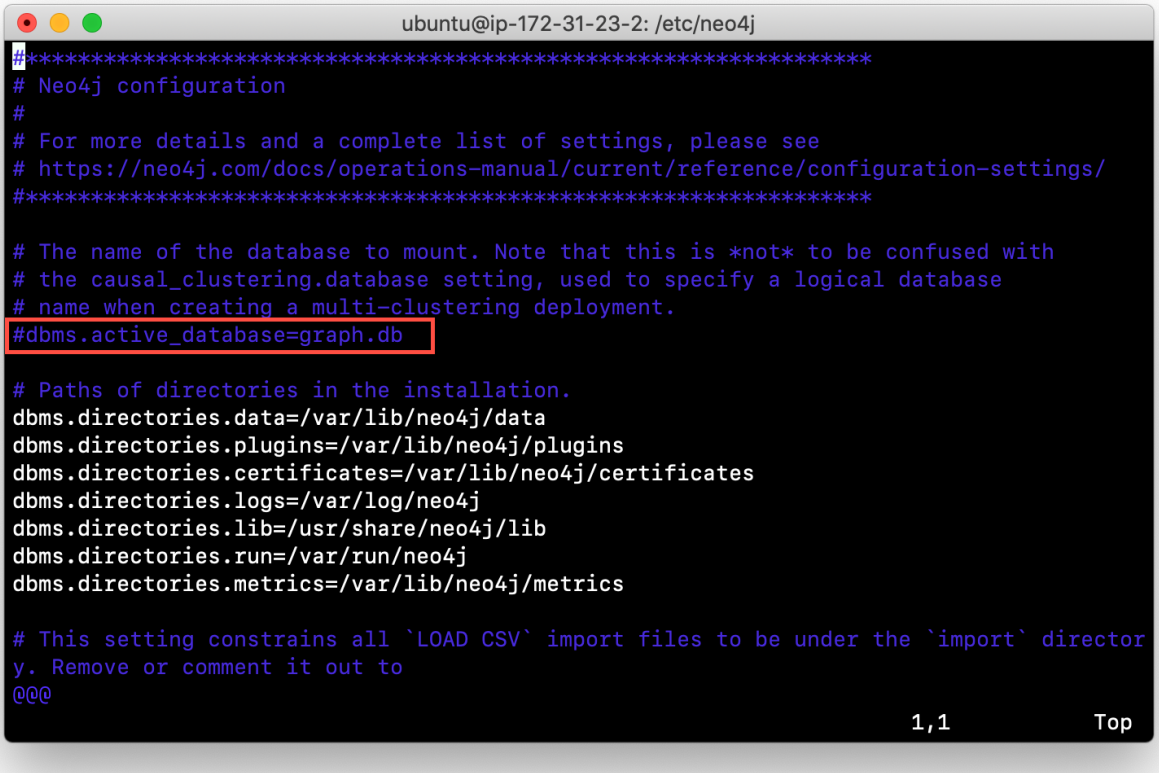


12. Exit `cypher-shell`.

Renaming a Neo4j database

By default, the Neo4j database (on Debian) is located in the **/var/lib/neo4j/data/databases** folder. The database is represented by a subfolder with the default name, **graph.db**. You should never modify, copy, or move any files or folders at or under **graph.db**.

A key file for a Neo4j instance is **/etc/neo4j/neo4j.conf**. This file contains all settings used by the Neo4j instance at runtime. Here is a portion of the default **neo4j.conf** file that is installed with Neo4j. The setting for the name of the database is the property *dbms.active_database*, which, by default, is **graph.db**. Since this is the default configuration as installed, this setting is commented out in the configuration file because Neo4j uses the default at runtime.



If you wanted to change the name of the Neo4j database, you could change the folder name **graph.db** to another name, but if you do so, you must uncomment the line in **neo4j.conf** for *dbms.active_database* to match what you have renamed the database folder to. You should make this type of change in the configuration when the Neo4j instance is stopped.

Deleting a Neo4j database

You would want to delete a Neo4j database for a couple of reasons:

- The database is no longer needed or usable and you want to recreate a fresh database.
- The database is no longer needed and you want to remove it so that a new database can be used. To do this you would load a new database which you will learn about next in this module.

To delete a Neo4j database used by a Neo4j instance you must:

1. Stop the Neo4j instance.
2. Remove the folder for the active database.

For example, delete the **graph.db** database:

```
[sudo] rm -rf /var/lib/neo4j/data/databases/graph.db
```

After deleting the Neo4j database, if you were to start the Neo4j instance, it would recreate an empty database. If you want to copy an existing database for use with this Neo4j instance, you dump and load an existing database to be used as the active database. Then you can start the Neo4j instance. You will learn about dumping and loading a database next.

Copying a Neo4j database

The structure of a Neo4j database is proprietary and could change from one release to another. You should never copy the database from one location in the filesystem/network to another location. You copy a Neo4j database by creating an offline backup.

To create an offline backup of a database that, perhaps you want to have as an additional copy or you want to give to another user for use on their system, you must:

1. Stop the Neo4j instance.
2. Ensure that the folder where you will dump the database exists.
3. Use the `dump` command of the `neo4j-admin` tool to create the dump file.
4. You can now copy the dump file between systems.

Then, if you want to create a database from any offline backup file to use for a Neo4j instance, you must:

1. Stop the Neo4j instance.
2. Determine what you will call the new database and adjust **neo4j.conf** to use this database as the active database.
3. Use the `load` command of the `neo4j-admin` tool to create the database from the dump file using the same name you specify in the **neo4j.conf** file.
4. Start the Neo4j instance.

NOTE	Dumping and loading a database is done when the Neo4j instance is stopped. Later in this module, you will learn about online backup and restore. Offline backup is typically done for initial setup and development purposes. Online backup and restore is done in a production environment.

Creating and offline backup

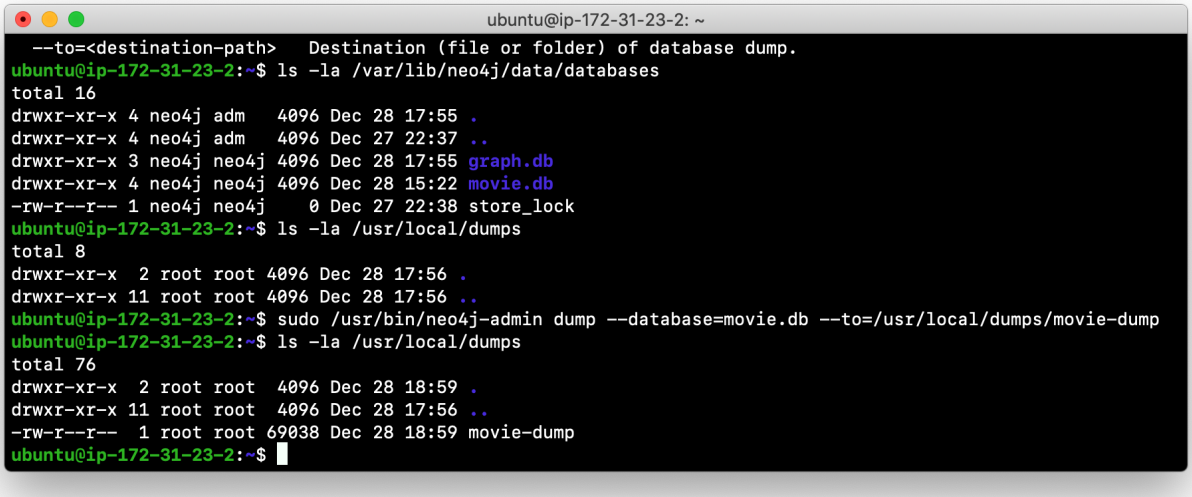
To create an offline backup, the Neo4j instance must be stopped. Here is how to use the `dump` command of the `neo4j-admin` tool to dump a database to a file:

```
[sudo] neo4j-admin dump --database=db-folder --to=db-target-folder/db-dump-file
```

where:

db-folder	is the name of the folder representing source database to be dumped.
db-target-folder	is the folder in the filesystem where you want to place the dumped database. This folder must exist.
db-dump-file	is the name of the dump file that will be created.

Here is an example where we have previously renamed the database to be *movie.db* and we have created a folder named *dumps*. We dump the *movie.db* using `neo4j-admin`:



After the dump file, *movie-dump* is created, you can move it anywhere on the filesystem or network.

Creating a database from an offline backup

Assuming that you have a dump file to use, you must first determine what the name of the target database will be. If you use an existing database name, the `load` command, can overwrite the database. If you want to create a new database, then you specify a database name that does not already exist. To perform the `load` command, the Neo4j instance must be stopped. In addition, the user:group permissions of the files created must be *neo4j:neo4j*.

NOTE	You must either perform the `load` operation as the *neo4j* user, or after the load, you must change the owner of all files and folders created to *neo4j:neo4j*.

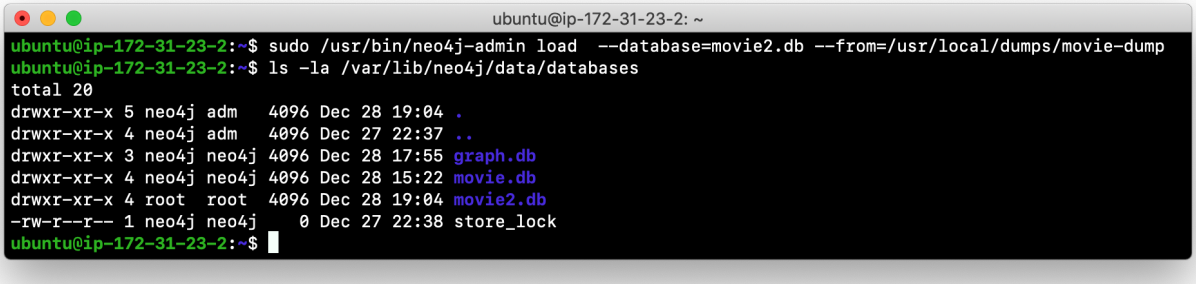
Here is how to use the `load` command of the `neo4j-admin` tool to load a database from a file:

```
[sudo] neo4j-admin load --from=path/db-dump-file --database=db-folder [--force=true]
```

where:

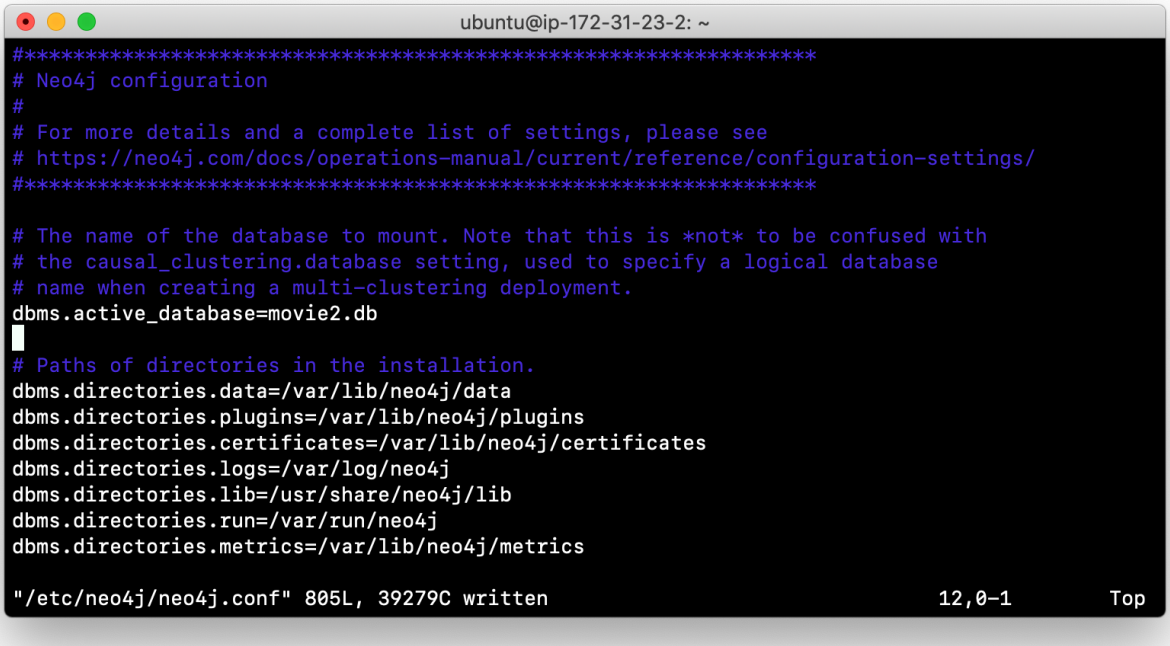
path	is a folder in the filesystem where the dump file resides.
db-dump-file	is the file previously created with the `dump` command of `neo4j-admin`.
db-folder	is the name of the database that will be created. The database is overwritten if --force is specified as `true`.

Here is an example where we load the contents of **movie-dump** into a database named **movie2.db**.



Modifying config for new database

In order to access this newly created and loaded database, we must modify **neo4j.conf** to use **movie2.db** as the active database before starting the Neo4j instance:



In addition, we must change the owner:group for the database folder and its sub-folders to *neo4j:neo4j* before we start the Neo4j instance.

A best practice is to examine the log file for the Neo4j instance after you have made any configuration changes to ensure that the instance starts with no errors.

Exercise #3: Copying a database

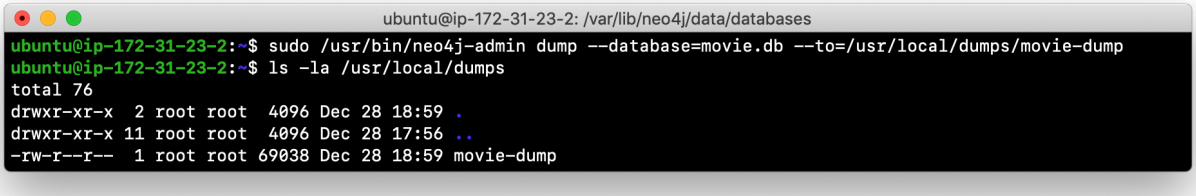
In this Exercise, you will make a copy of your active database that has the movie data in it and use the dump file to create a database.

Before you begin

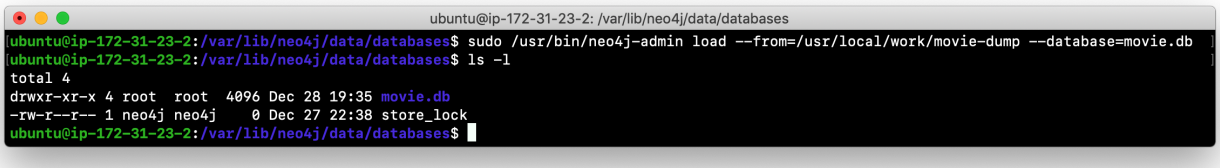
You should have loaded the **graph.db** database with the movie data (Exercise #2) and stopped the Neo4j instance.

Exercise steps:

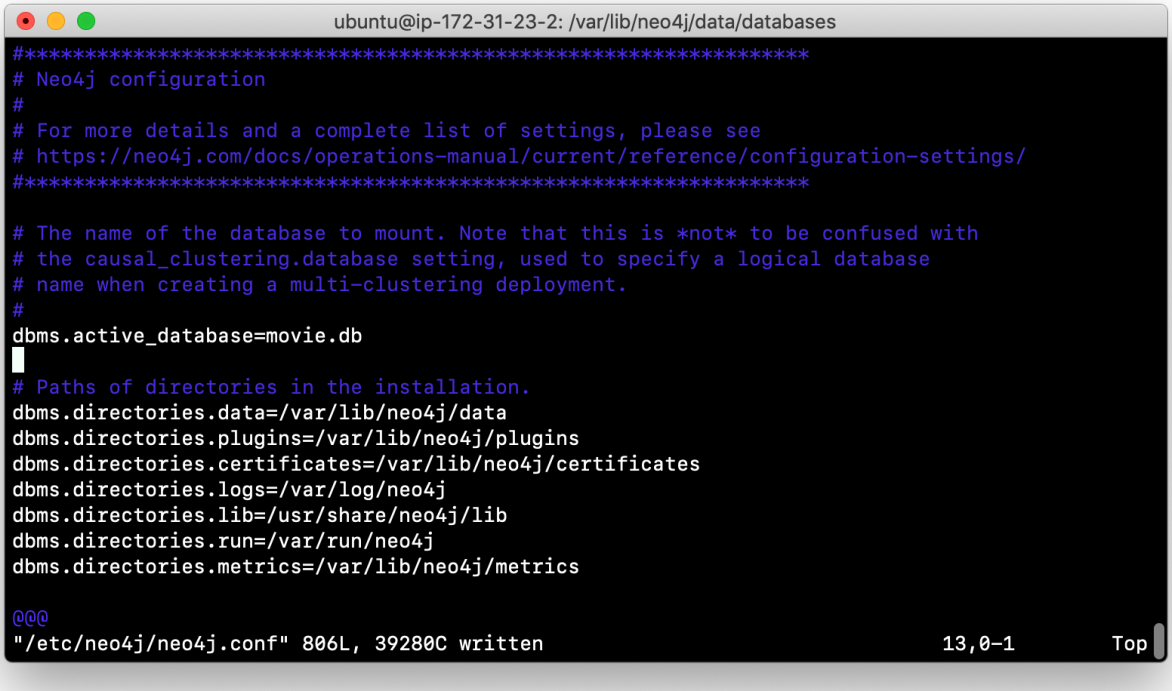
1. Open a terminal on your system.
2. Create a folder named /**usr/local/work**.
3. Use the `neo4j-admin` script to dump the **graph.db** database to the **work** folder. You should do something like this:



4. Notice that this dump file is simply a file that can be copied to any location.
5. Delete the **graph.db** database by removing the **graph.db** folder and its subfolders.
6. Use the `neo4j-admin` script to load the database from the dump file you just created. Name the database **movie.db**. You should do something like this:

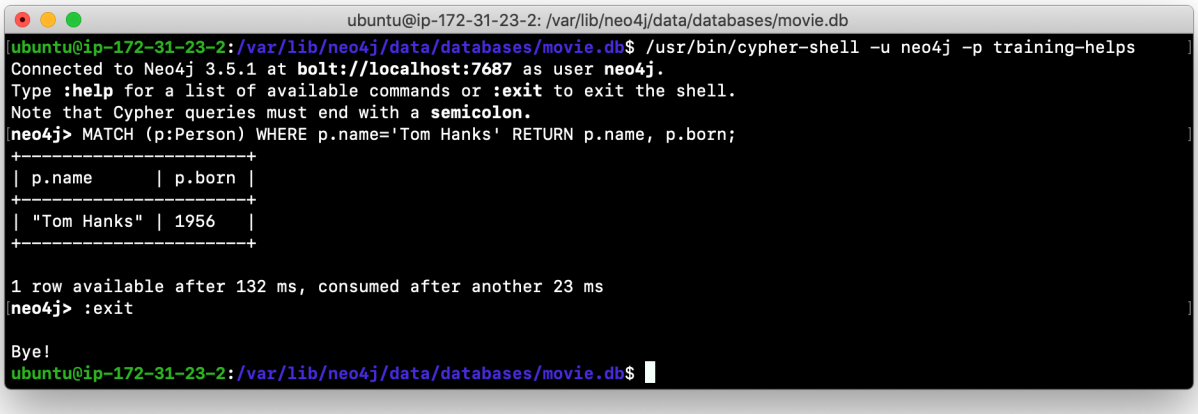


7. Modify **neo4j.conf** to use **movie.db** as the active database.



8. If you did not perform the load as the user *neo4j*, you must change the owner:group of all files and folders under **movie.db** to be *neo4j:neo4j*. For example, change directory to the **movie.db** folder and then enter the command: `[sudo] chown -R neo4j:neo4j movie.db` This will recursively change the owner and group to all files and folders including and under the **movie.db** folder.
9. Start the Neo4j instance.
10. Examine the log file to ensure that the instance started with no errors.

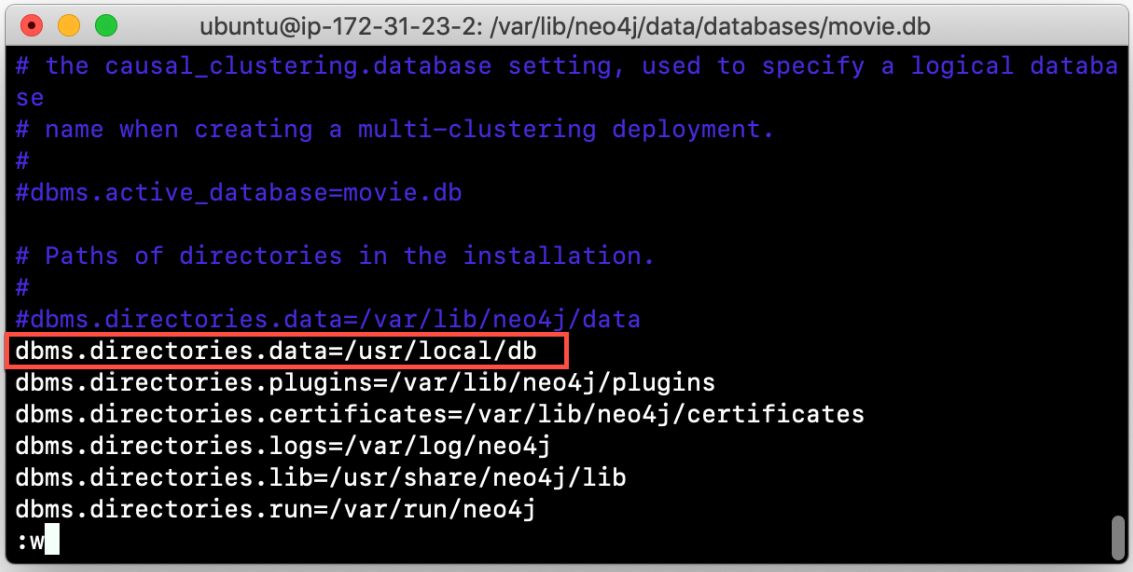
11. Access the database using `cypher-shell`. Can you see the movie data in the database?



Modifying the location of the database

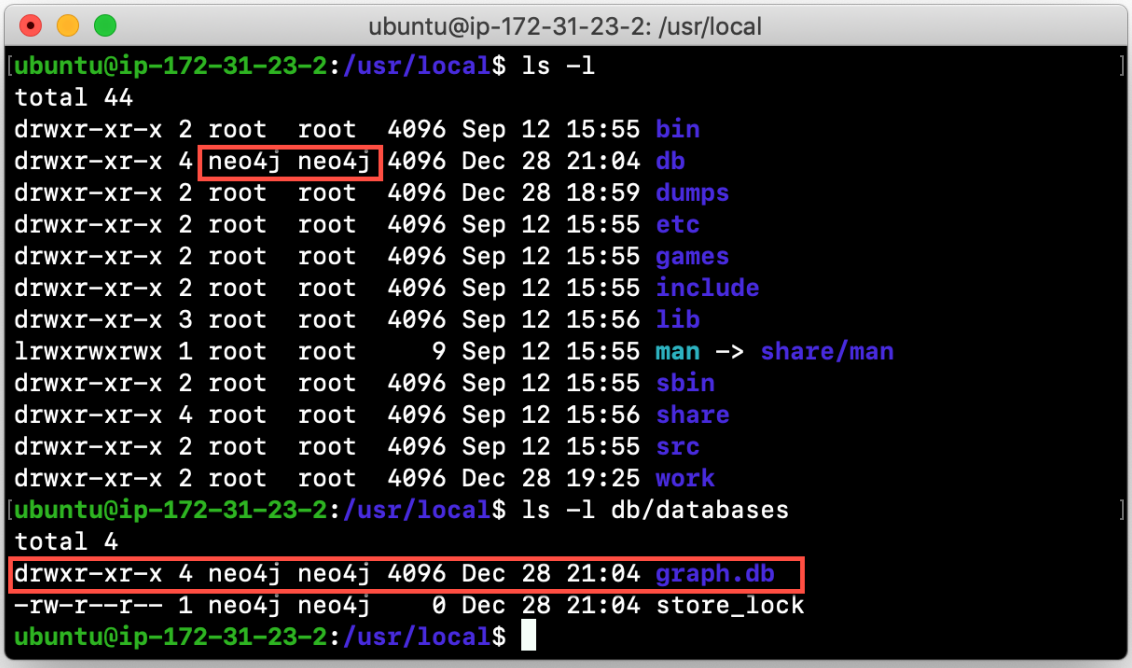
If you do not want the database used by the Neo4j instance to reside in the same location as the Neo4j installation, you can modify its location in the **neo4j.conf** file. If you specify a new location for the data, it must exist in the filesystem and the folder must be owned by *neo4j:neo4j*.

Here we have specified a new location for the data in the configuration file:



Starting Neo4j instance with a new location

We ensure that the location for the data exists and then we can start the Neo4j instance. If this is the first time Neo4j has been started for this location, a new database named **graph.db** will be created. This is because we are using the default database name in the configuration file.



If you have an existing database that you want to reside in a different location for the Neo4j instance, remember that you must dump and load the database to safely copy it to the new location.

Using a different location for the database

If you are starting the Neo4j instance with a new location and do not want to use the default **graph.db** database, you must follow these steps to ensure that the folders for the database are set up properly:

1. Specify the new location in the configuration file, but do not specify the active database name.
2. Start or restart the Neo4j instance. A new **graph.db** folder will be created as well as the **dbms** folder required by the instance (contains important authentication information).
3. Examine the log file to ensure that it started without errors.
4. Stop the Neo4j instance.
5. Specify the name of the active database in the configuration file.
6. Load the data into the database name that will be the active database.
7. Ensure that the database folder and its subfolders are owned by *neo4j:neo4j*.
8. Start the Neo4j instance.
9. Examine the log file to ensure it started without errors.
10. Optionally, you can remove the **graph.db** folder as you will be working with a different database you loaded.

Exercise #4: Modifying the location of the database

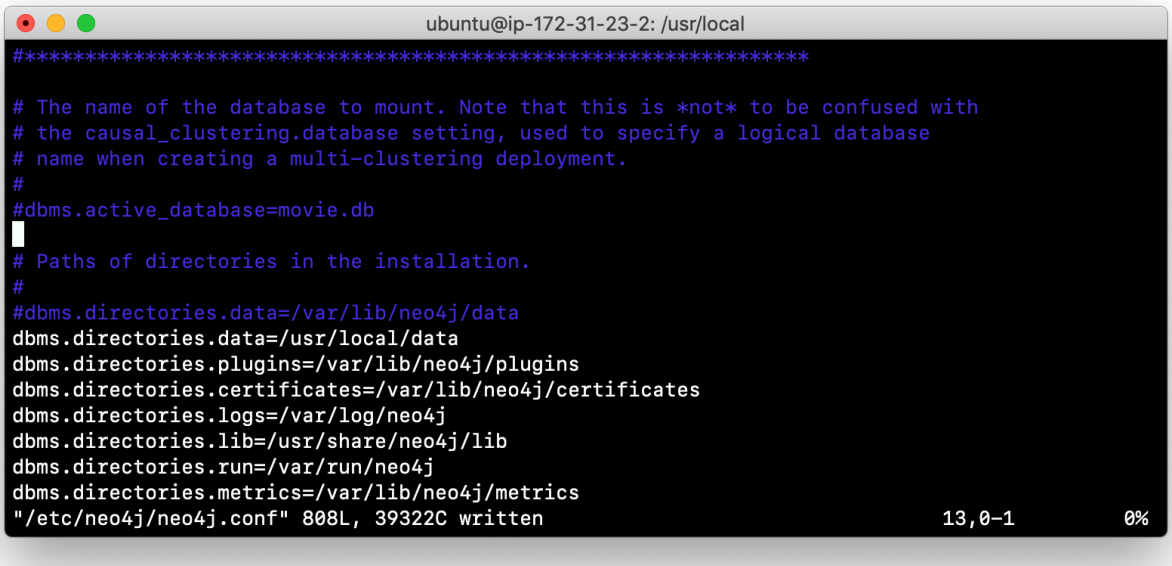
In this Exercise, you will set up a different location for the database in your local filesystem and start the Neo4j instance using the database at this new location.

Before you begin

1. You should have created the dump file for the movie database (Exercise #3).
2. Stop the Neo4j instance.

Exercise steps:

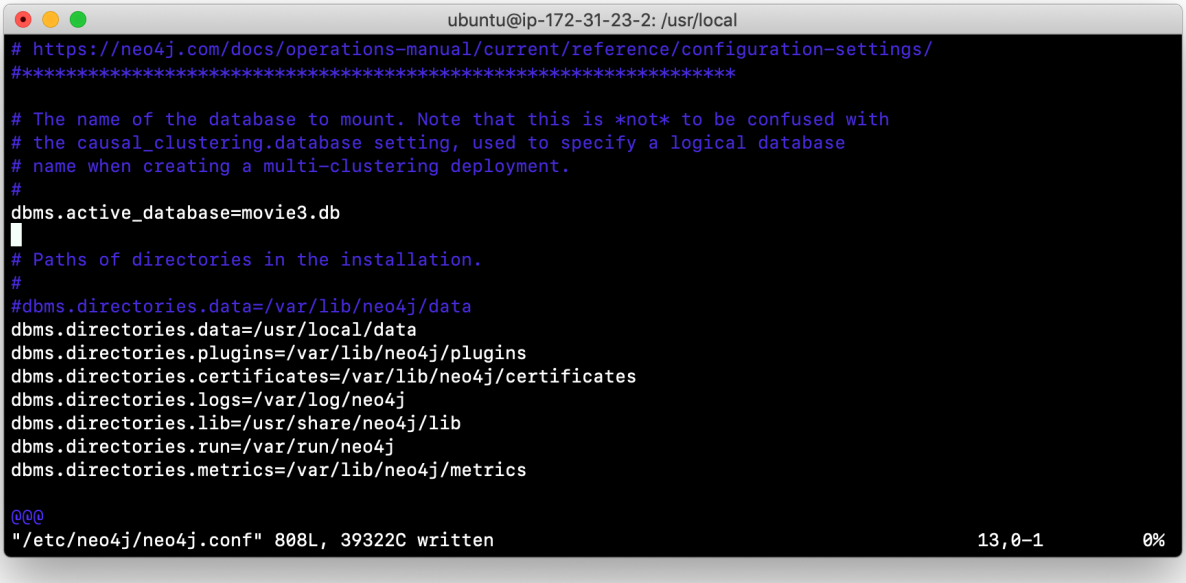
1. Open a terminal on your system.
2. Create a folder named **/usr/local/data**. This is the folder where the database will reside which is different from the default location used by Neo4j.
3. Make sure that this **data** folder is owned by *neo4j:neo4j*. For example, navigate to the **/usr/local** folder and enter `[sudo ]chown neo4j:neo4j data`.
4. Modify the **neo4j.conf** file to use **/usr/local/data** as the data directory. Also ensure that there is no active database specified. Your **neo4j.conf** file should look something like this:



5. Start the Neo4j instance.
6. Examine the log file to ensure that the instance started without errors.

7. Examine the files in the **/usr/local/data** location. The instance should have created the **databases** and **dbms** folders. They should look as follows:

8. Stop the Neo4j instance.
9. Modify the **neo4j.conf** file to use **movie3.db** as the active database. Your **neo4j.conf** file should look something like this:



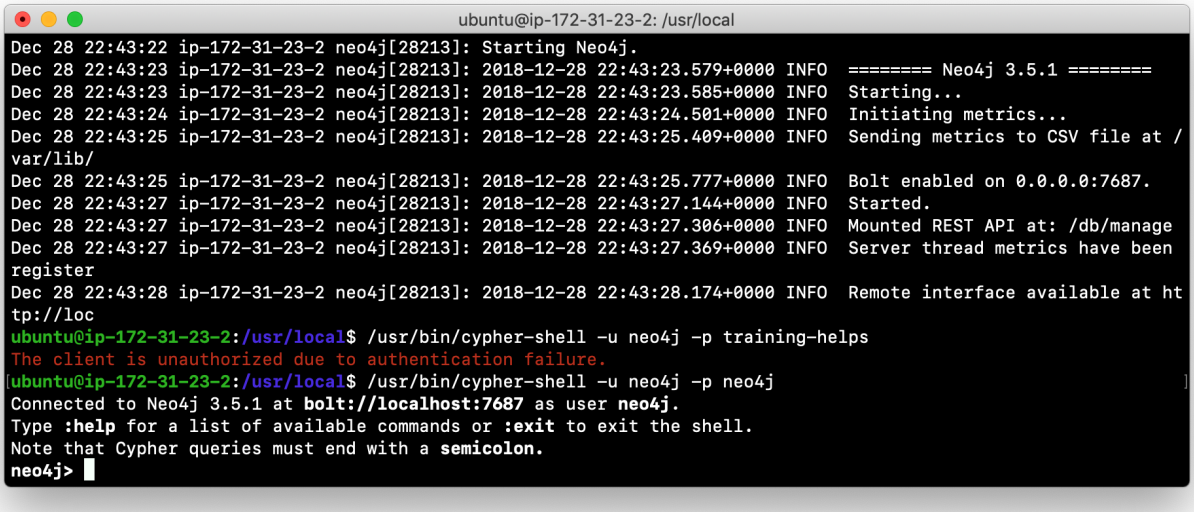
10. Use the `neo4j-admin` script to load the database from the dump file you created in Exercise 3. Name the database **movie3.db** You should do something like this:

11. Ensure that all files and folders including and under **movie3.db** are owned by *neo4j:neo4j*. For example, change directory to the **databases** folder and then enter the command: `[sudo] chown -R neo4j:neo4j movie3.db` This will recursively change the owner and group to all files and folders under **movie3.db**.

12. Start the Neo4j instance.

13. Examine the log file to ensure that no errors occurred.

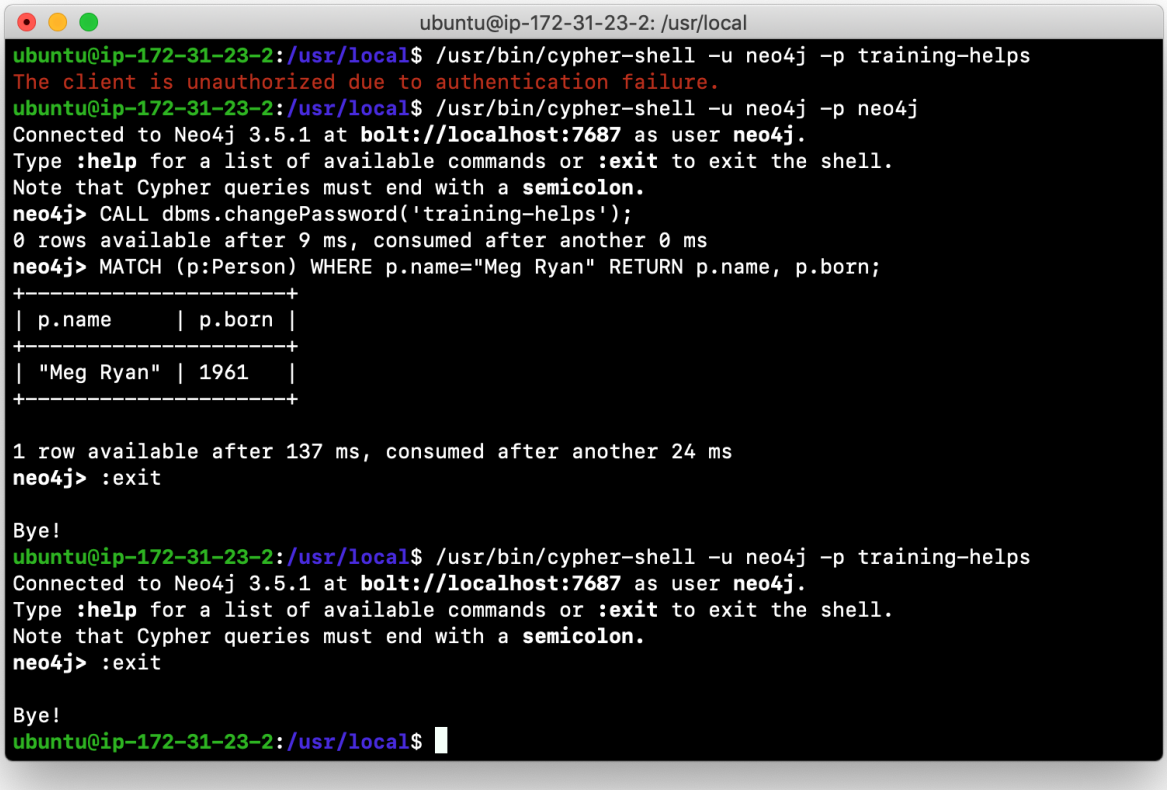
14. Access the database using `cypher-shell`. Do you get an authentication error? This is because the database is now located in a different location and the default credentials of *neo4j/neo4j* are used.



15. Enter the Cypher statement to change the password: `CALL dbms.changePassword('newPassword');`

16. Enter a Cypher statement to retrieve some data: `MATCH (p:Person) WHERE p.name='Meg Ryan' RETURN p.name, p.born;`

17. Exit `cypher-shell`.



Checking the consistency of a database

A database's consistency could be compromised if a software or hardware failure has occurred that affects the Neo4j instance. You will learn later in this module about live backups, but if you have reason to believe that a specific database has been corrupted, you can perform a consistency check on it.

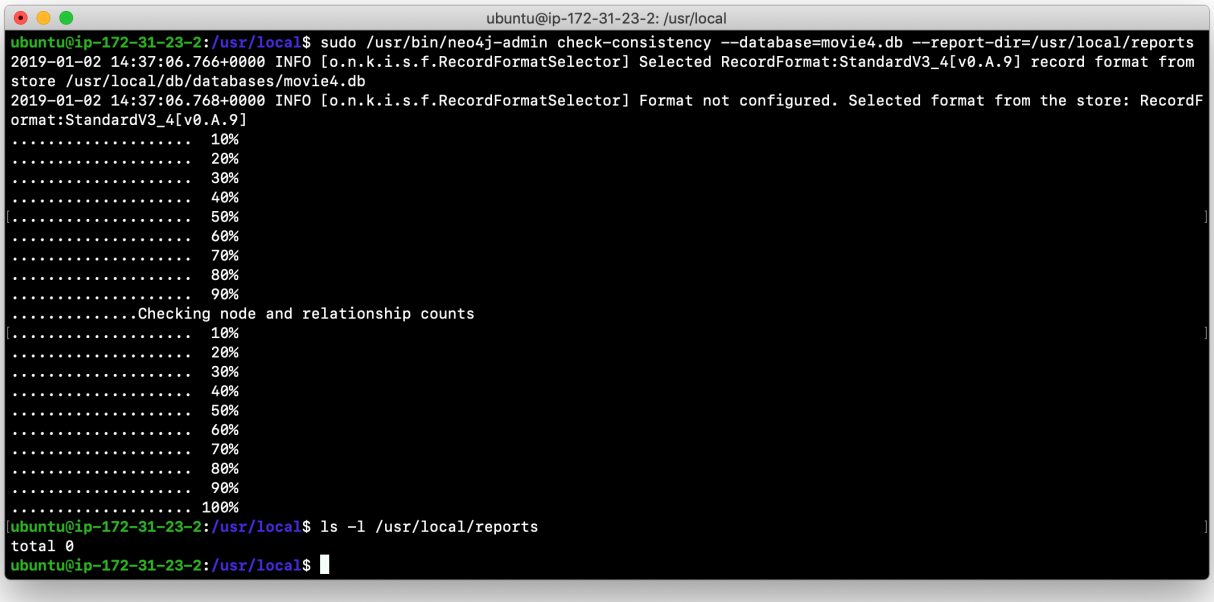
The Neo4j instance must be stopped to perform the consistency check.

Here is how you use the `neo4j-admin` tool to check the consistency of the database:

```
[sudo] neo4j-admin check-consistency --database=db-name --report-dir=report-location [--verbose=true]
```

The database named *db-name* is found in the data location specified in **neo4j.conf** file. If the tool comes back with no error, then the database is consistent. Otherwise, an error is returned and a report is written to *report-location*. You can specify verbose reporting. See the Neo4j Operations Manual for more options. For example, you can check the consistency of a backup which is a best practice.

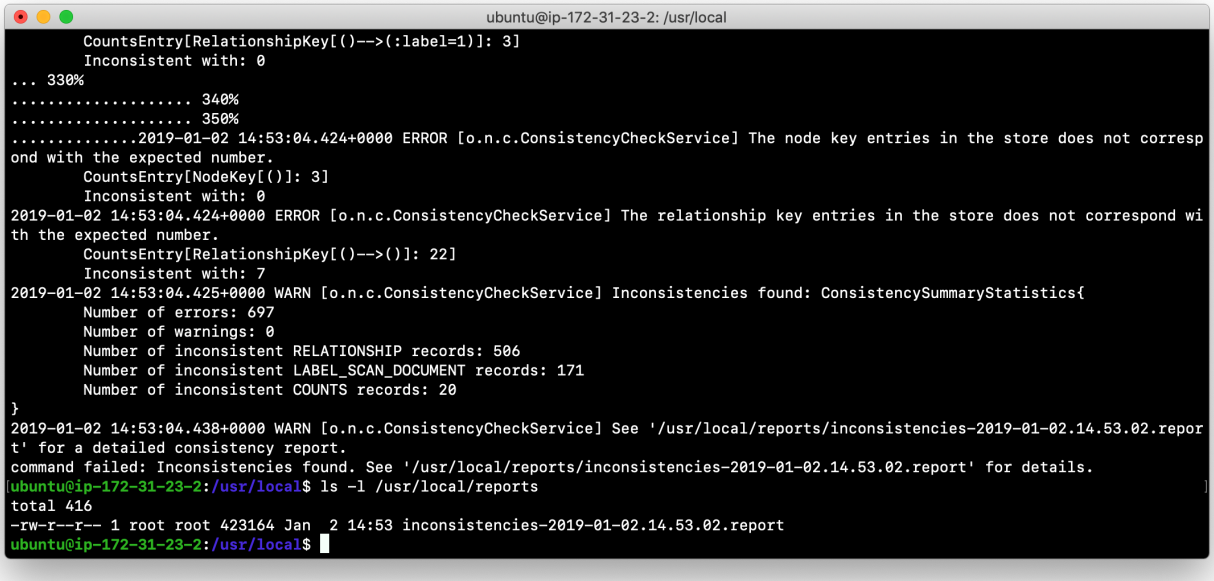
Suppose we had loaded the **movie4.db** database with `neo4j-admin`. Here is what a successful run of the consistency checker should produce:



No report is written to the reports folder because the consistency check passed.

Inconsistencies found

Here is an example of what an unsuccessful run of the consistency checker should produce:



If inconsistencies are found, a report is generated and placed in the folder specified for the report location.

Inconsistencies in a database are a serious matter that should be looked into with the help of Neo4j Technical Support.

Exercise #5: Checking consistency of a database

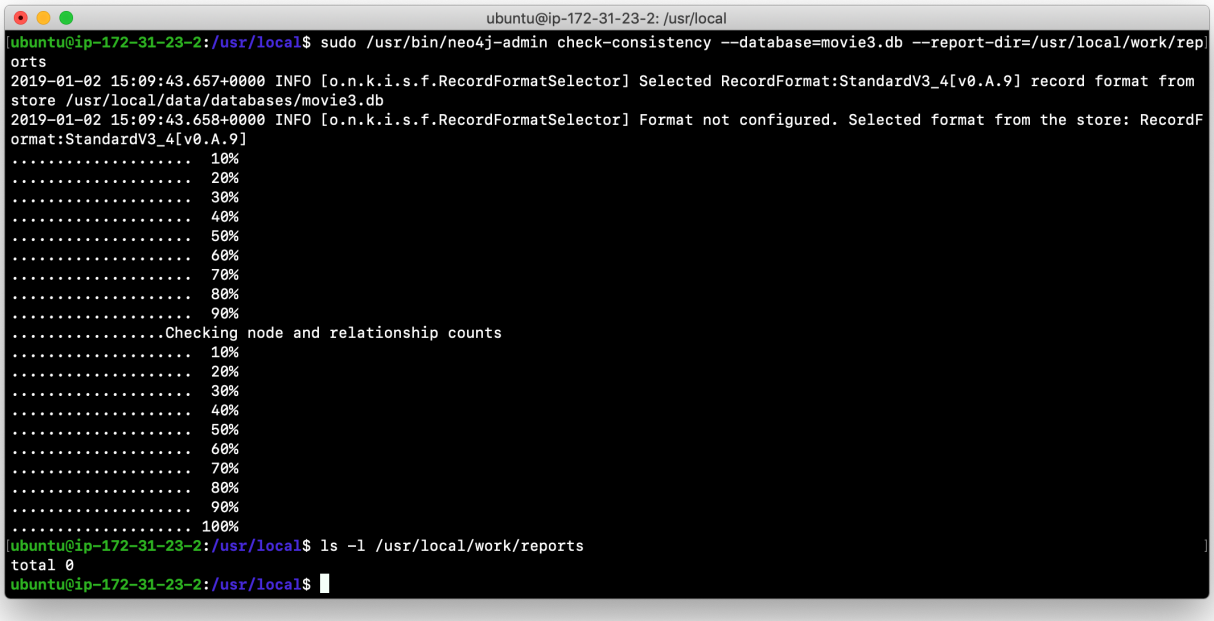
In this Exercise, you check the consistency of a database that is consistent. Then you modify a file that causes the database to become corrupt and then check its consistency.

Before you begin

1. You should have created and started the **movie3.db** database (Exercise #4).
2. Stop the Neo4j instance.
3. Create a folder named **/usr/local/work/reports**.

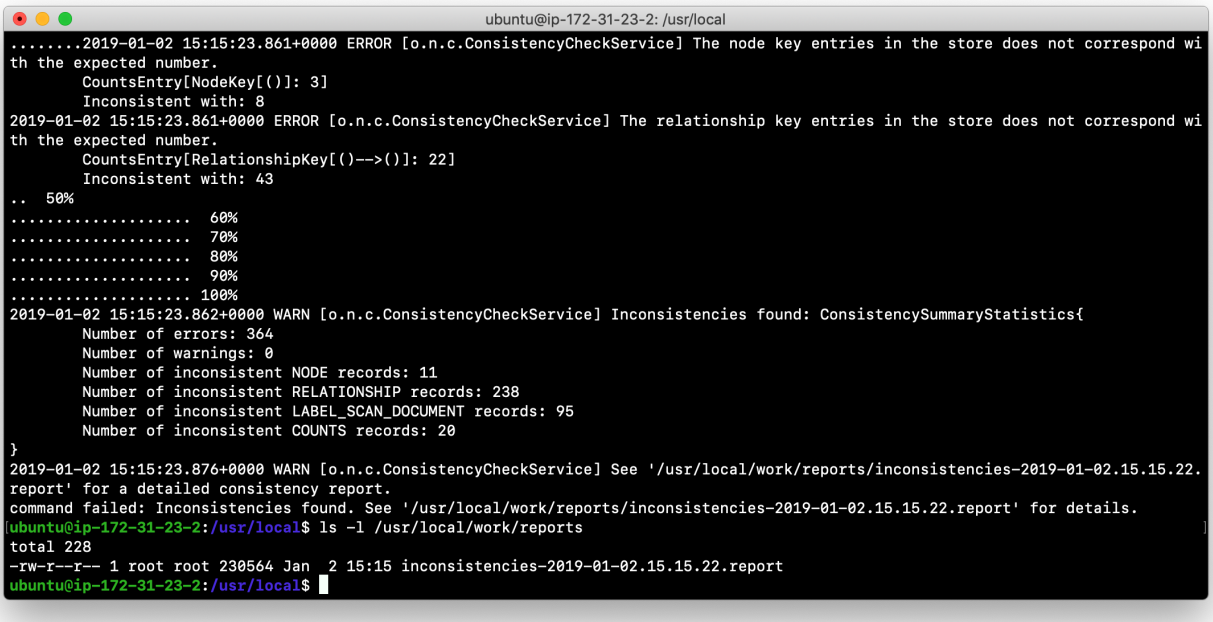
Exercise steps:

1. Open a terminal on your system.
2. Run the consistency check tool on **movie3.db** using `neo4j-admin` specifying **reports** as the folder where the report will be written. The consistency check tool should return the following:



3. Modify the Neo4j configuration to use a database named **movie3-copy.db**, rather than **movie3.db**.
4. Use `neo4j-admin` to create and load **movie3-copy.db** from the movie dump file you created earlier.
5. Ensure that the owner of the **movie3-copy.db** is *neo4j:neo4j*.
6. Next, you will corrupt the database. Modify the file **movie3-copy.db/neostore.nodestore.db** by adding some text to the file.

7. Run the consistency check tool on **movie3-copy.db** using `neo4j-admin` specifying **/usr/local/work/reports** as the folder where the report will be written. The consistency check tool should return something like the following:



Scripting with cypher-shell

As a database administrator, you may need to automate changes to the database. The most common types of changes that administrators may want to perform are operations such as adding/dropping constraints or indexes. Note that you will need to work with the developers and architects of your application to determine what indexes must be created. You can create scripts that forward the Cypher statements to `cypher-shell`. The number of supporting script files you create will depend upon the tasks you want to perform against the database.

Examples: Adding constraints

Suppose that we use *bash*. We create 3 files:

1. AddConstraints.cypher that contains the Cypher statements to execute in `cypher-shell`:

```
CREATE CONSTRAINT ON (m:Movie) ASSERT m.title IS UNIQUE;
CREATE CONSTRAINT ON (p:Person) ASSERT p.name IS UNIQUE;
CALL db.constraints();
```

Each Cypher statement must end with a `;`.

2. AddConstraints.sh that invokes `cypher-shell` using a set of Cypher statements and specifies verbose output:

```
cat /usr/local/work/AddConstraints.cypher | /usr/bin/cypher-shell -u neo4j -p
training-helps --format verbose
```

3. PrepareDB.sh that initializes the log file, **PrepareDB.log**, and calls the script to add the constraints:

```
rm -rf /usr/local/work/PrepareDB.log
/usr/local/work/AddConstraints.sh 2>&1 >> /usr/local/work/
PrepareDB.log
# Other scripts here
```

When the **PrepareDB.sh** script runs its scripts, all output will be written to the log file, including error output. Then you can simply check the log file to make sure it ran as expected.

Exercise #6: Scripting changes to the database

In this Exercise, you will gain experience scripting with `cypher-shell`. You will create three files in the **/usr/local/work** folder:

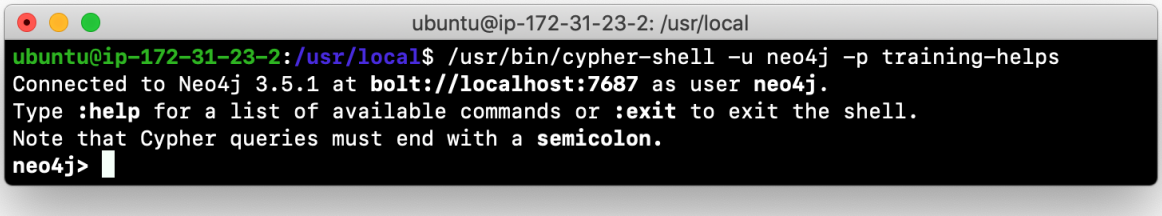
1. **AddConstraints.cypher**
2. **AddConstraints.sh**
3. **MaintainDB.sh**

Before you begin

1. Remove the **databases/movie3-copy.db** folder as this database is now corrupt.
2. Ensure that the Neo4j configuration uses **movie3.db** for the database.
3. Start or restart the Neo4j instance.

Exercise steps:

1. Open a terminal on your system.
2. Start `cypher-shell`, providing the credentials for the *neo4j* user.



3. Enter some simple Cypher statements to confirm that you can access the database. For example:
   a. `CALL db.schema();`
   b. `CALL db.constraints();`
4. Exit Cypher-shell by typing `:exit`.
5. Create a Cypher script in the **/usr/local/work** folder named **AddConstraints.cypher** with the following statements:

```
CREATE CONSTRAINT ON (m:Movie) ASSERT m.title IS UNIQUE;
CREATE CONSTRAINT ON (p:Person) ASSERT p.name IS UNIQUE;
```

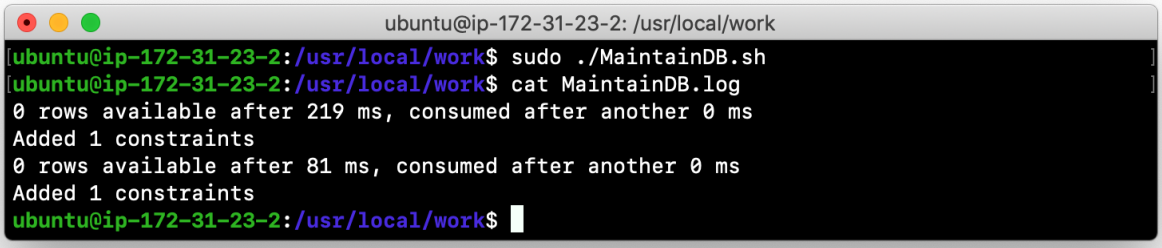
6. Create a shell script in the **/usr/local/work** folder named **AddConstraints.sh** that will forward **AddConstraints.cypher** to `cypher-shell`. This file should have the following contents:

```
cat /usr/local/work/AddConstraints.cypher | /usr/bin/cypher-shell -u neo4j -p
training-helps --format verbose
```

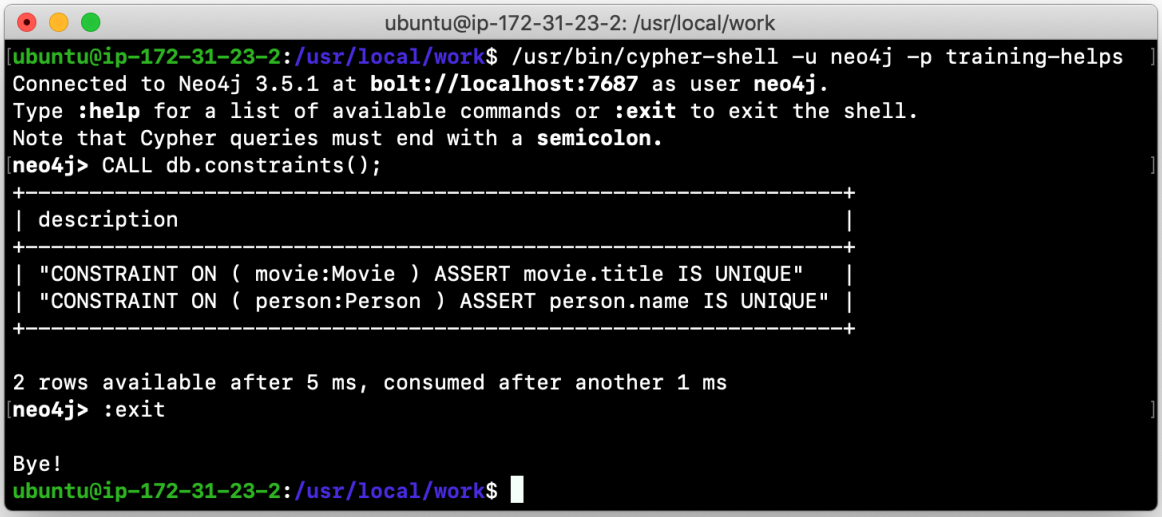
7. Create a shell script in the **/usr/local/work** folder named **MaintainDB.sh** that will initialize the log file and then call **AddConstraints.sh**. This file should have the following contents:

```
rm -rf /usr/local/work/MaintainDB.log
/usr/local/work/AddConstraints.sh 2>&1 >> /usr/local/work/MaintainDB.log
```

8. Ensure that the scripts you created have execute permissions.
9. Run the **MaintainDB.sh** script and view the log file.



10. Confirm that it created the constraints in the database. (Check using cypher-shell (`CALL db.constraints();`))



Managing plugins

Some applications can use Neo4j out-of-the-box, but many applications require additional functionality that could be:

- A library supported by Neo4j such as GraphQL or GRAPH ALGORITHMS.
- A community-supported library, such as APOC.
- Custom functionality that has been written by the developers of your application.

We refer to this additional functionality as a *plugin* that contains specific procedures. A *plugin* is typically specific to a particular release of Neo4j. In many cases, if you upgrade to a later version of Neo4j, you may also need to install a new *plugin*. First, you should understand how to view the procedures available for use with the Neo4j instance. You do so by executing the Cypher statement `CALL db.procedures();`.

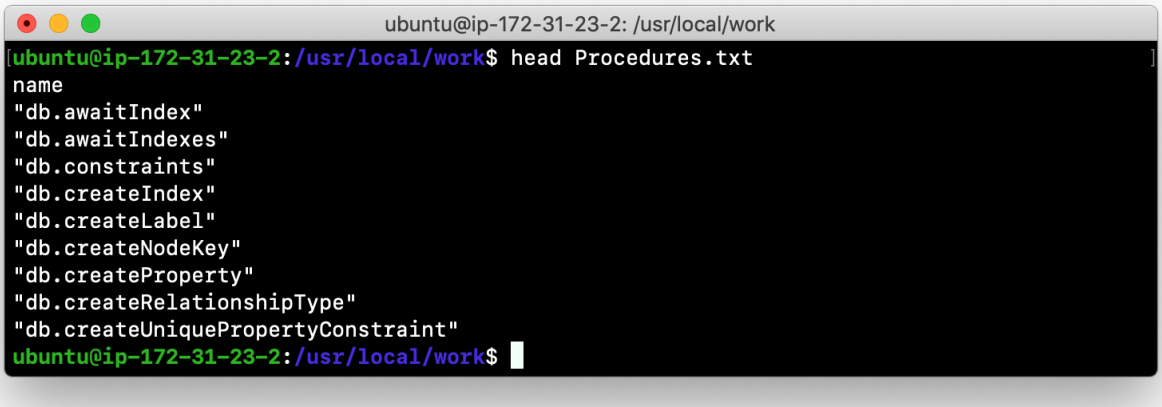
Retrieving available procedures

Here is an example of a script you can run to produce a file, **Procedures.txt** that contain the names of the procedures currently available for the Neo4j instance:

```
echo "CALL dbms.procedures() YIELD name;" | /usr/bin/cypher-shell -u neo4j -p
training-helps --format plain > /usr/local/work/Procedures.txt
```

This script calls `dbms.procedures()` to return the name of each procedure in the list returned.

Here is a view of **Procedures.txt**:



By default, the procedures available to the Neo4j instance are the built-in procedures that are named db.* and dbms.*.

Adding a plugin to the Neo4j instance

To add a plugin to your Neo4j instance, you must first obtain the **.jar** file. It is important to confirm that the **.jar** file you will use is compatible with the version of Neo4j that you are using. For example, a plugin released for release 3.4 of Neo4j can be used by a Neo4j 3.5 instance, but the converse may not be true. You must check with the developers of the plugin for compatibility.

Some plugins require a configuration change. You should understand the configuration changes required for any plugin you are installing.

NOTE	When you install Neo4j, the **plugins** folder contains a **README.txt** folder that contains instructions related to sandboxing and whitelisting. These instructions will change in future releases of Neo4j.

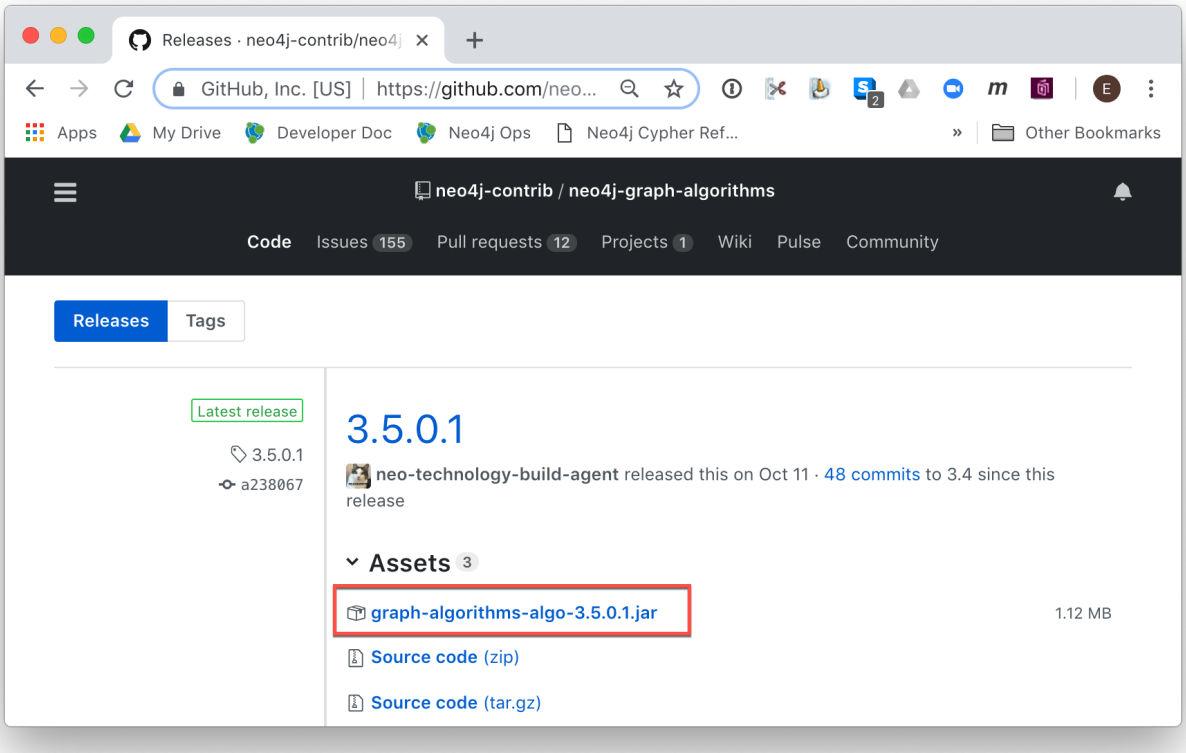
Sandboxing and whitelisting

Neo4j provides *sandboxing* to ensure that procedures do not inadvertently use insecure APIs. For example, when writing custom code it is possible to access Neo4j APIs that are not publicly supported, and these internal APIs are subject to change, without notice. Additionally, their use comes with the risk of performing insecure actions. The sandboxing functionality limits the use of extensions to publicly supported APIs, which exclusively contain safe operations, or contain security checks.

Neo4j *whitelisting* can be used to allow loading only a few extensions from a larger library. The configuration setting *dbms.security.procedures.whitelist* is used to name certain procedures that should be available from a library. It defines a comma-separated list of procedures that are to be loaded. The list may contain both fully-qualified procedure names, and partial names with the wildcard *.

Example: Installing the Graph Algorithms plugin

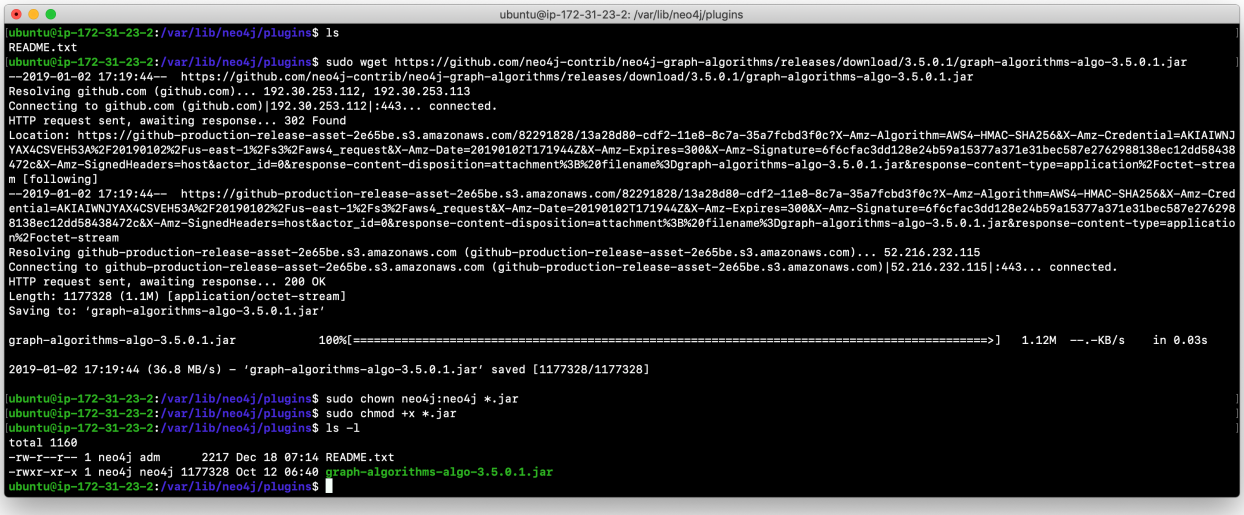
Suppose we wanted to install the Graph Algorithms library that is compatible with Neo4j 3.5. We find the library in GitHub and simply download the **.jar** file. Here is the release area in GitHub for the graph algorithms library:



The main page for Graph Algorithms in GitHub contains details about the plugin and instructions for installing it.

Example: Download and ownership of plugin

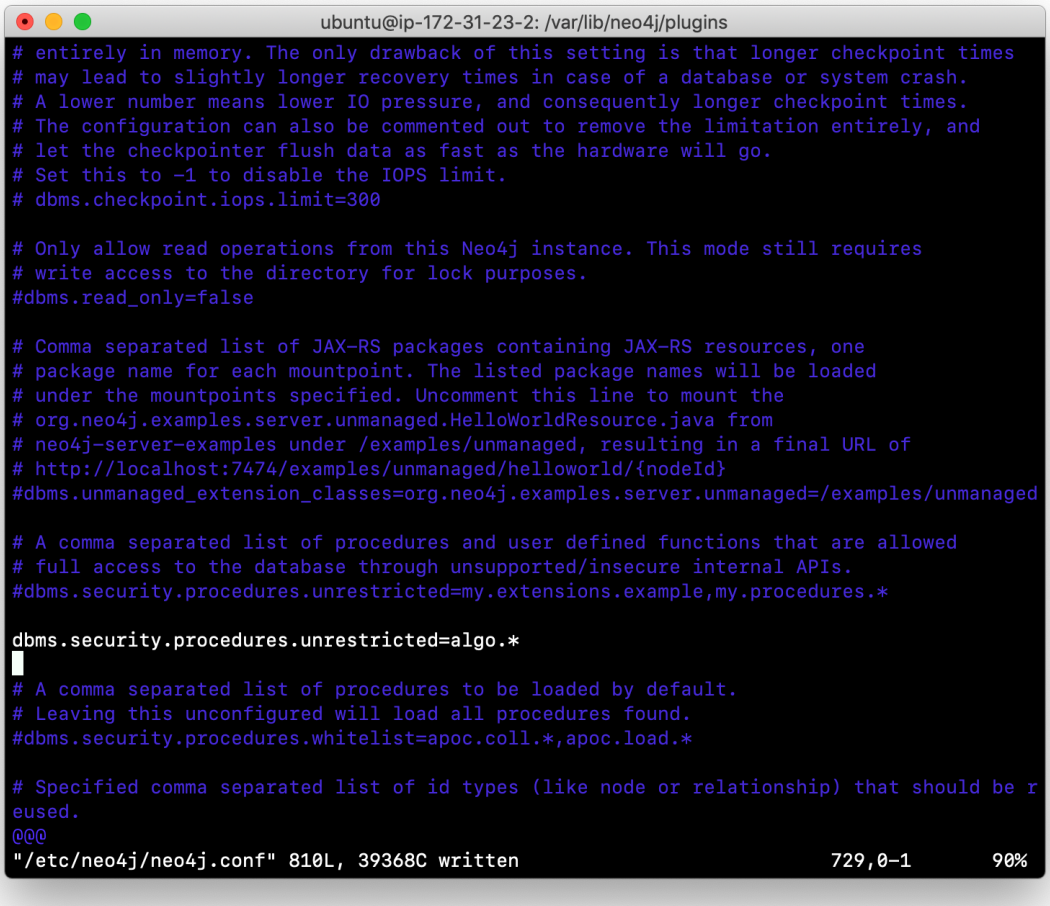
You download any plugins that your application will use to the /var/lib/neo4j/plugins folder:



Ensure that the **.jar** file is owned by *neo4j:neo4j* and that it has execute permissions.

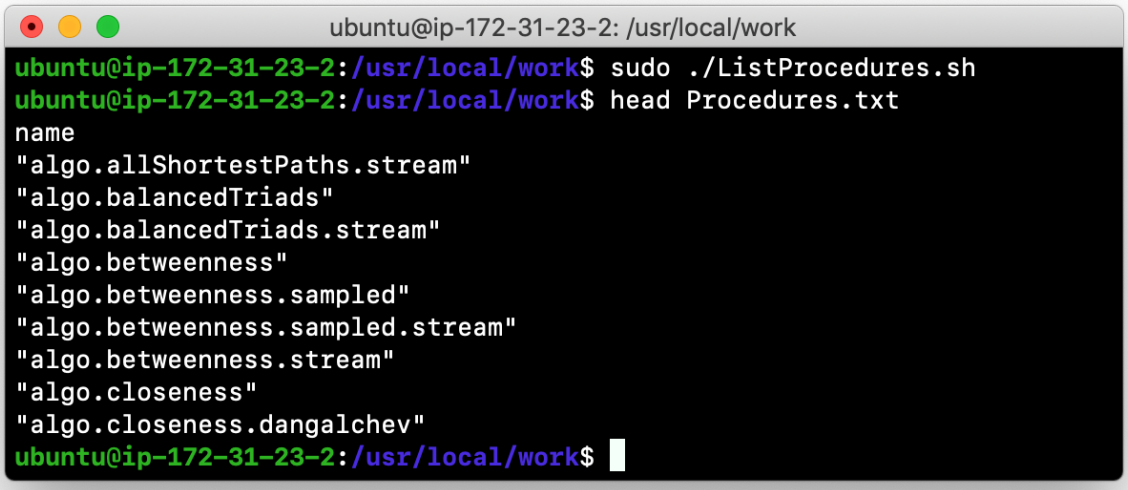
Example: Sandboxing

The graph algorithms plugin requires *sandboxing*. Here is how we enable the procedures in the graph algorithms plugin. We modify the *Miscellaneous Configuration* section of the **neo4j.conf** file as follows:



Example: Restart with plugin

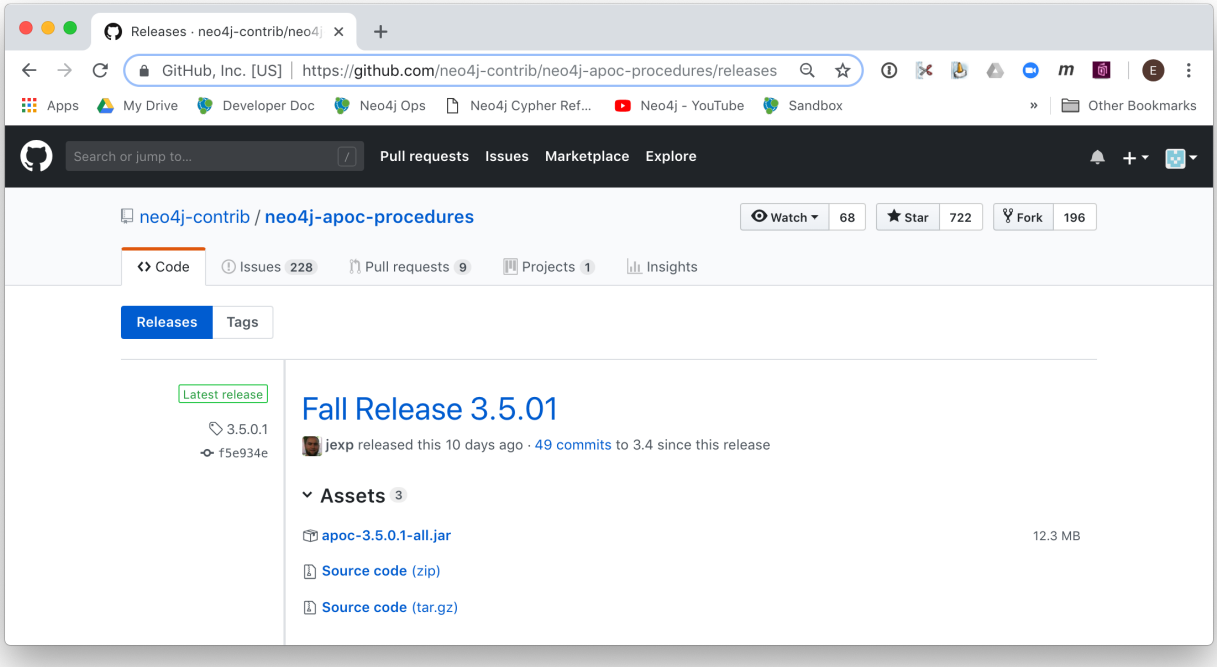
You must then start or restart the Neo4j instance. Once started, you can then run the script to return the names of the procedures that are available to the Neo4j instance. Here we see that we have the additional procedures for the graph algorithms plugin:



Example: Installing the APOC plugin

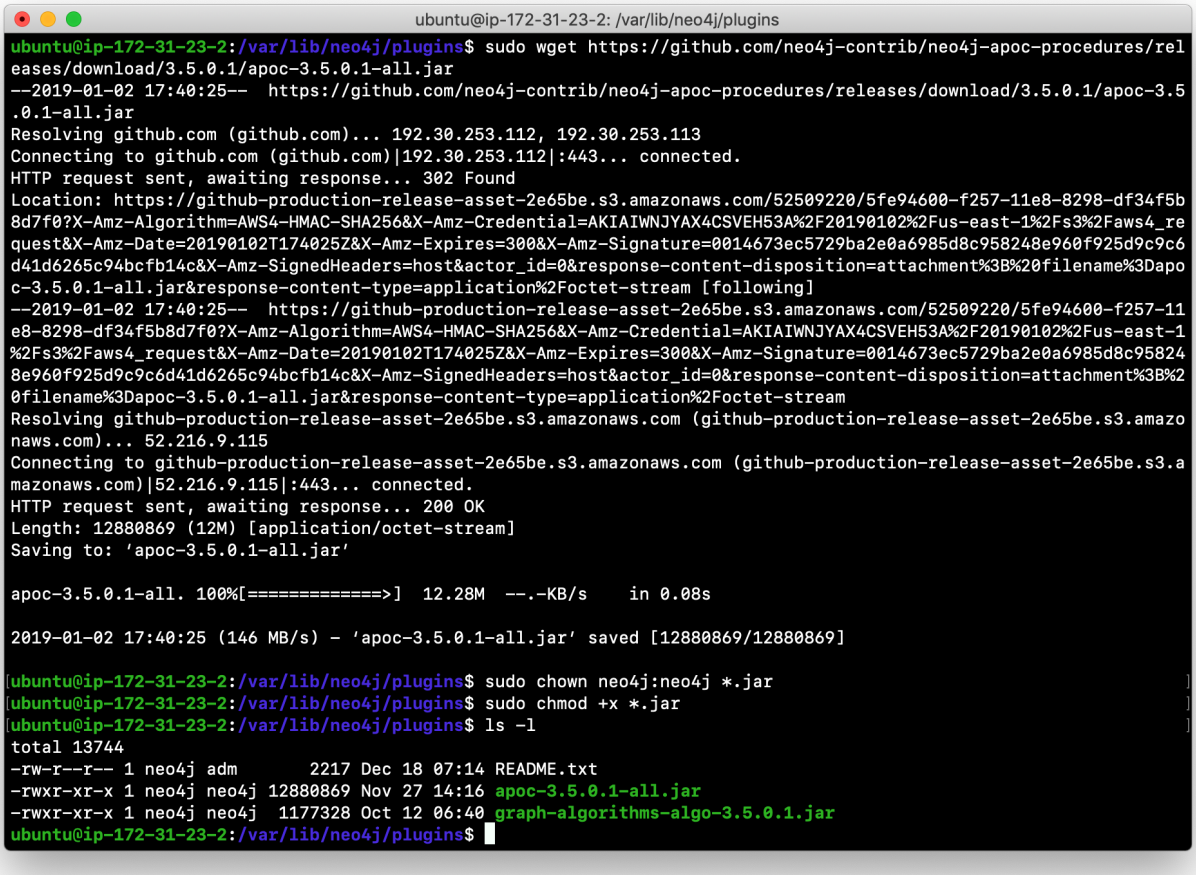
APOC (Awesome Procedures on Cypher) is a very popular plugin used by many applications. It contains over 450 user-defined procedures that make accessing a graph incredibly efficient and much easier than writing your own Cypher statements to do the same thing.

You obtain the plugin from the APOC releases page:



Example: Download and ownership of plugin

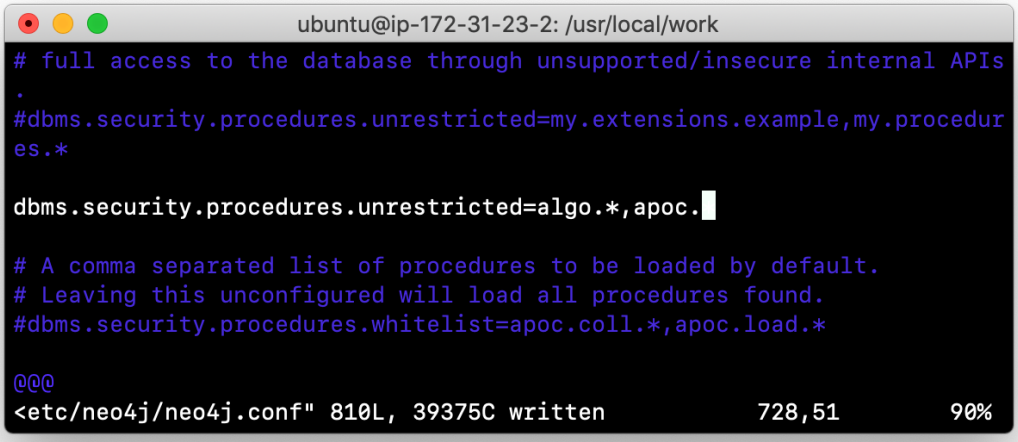
Here we download the **.jar** file, change its permissions to execute, and change the owner to be *neo4j:neo4j*.



Example: Sandboxing

After you have placed the **.jar** file into the **plugins** folder, you must modify the configuration for the instance as described in the main page for APOC. As described on this page, you have an option of either *sandboxing* or *whitelisting* the procedures of the plugin. How much of the APOC library is used by your application is determined by the developers so you should use them as a resource for this type of configuration change.

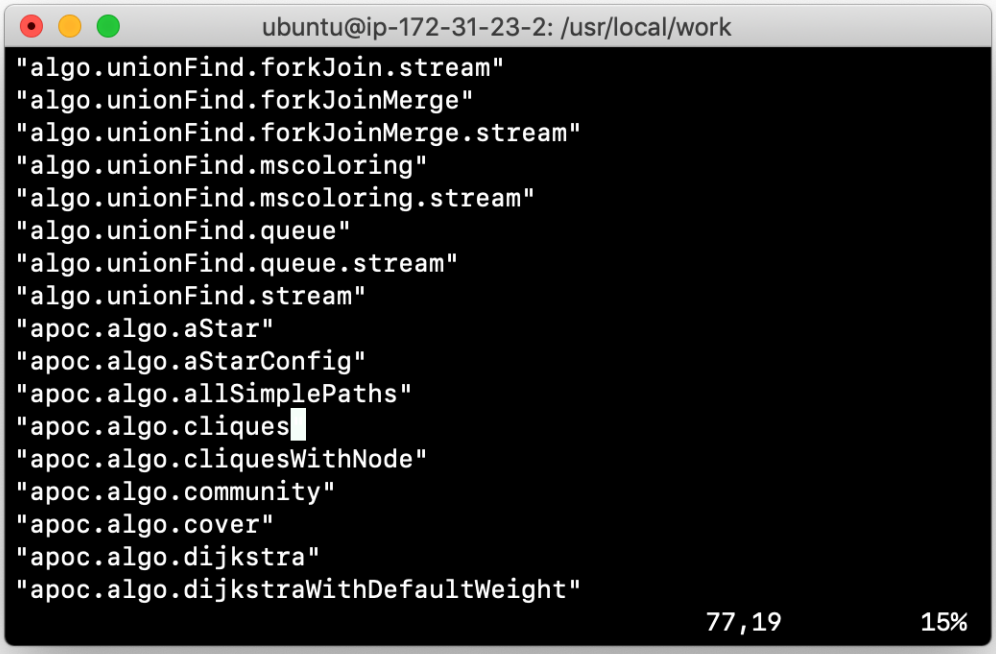
Suppose we want to allow all APOC procedures to be available to this Neo4j instance. We would sandbox the plugin in the **neo4j.conf** file as follows, similar to how we sandboxed the graph algorithms where we specify `dbms.security.procedures.unrestricted=algo.,apoc.`.



Since APOC is large, you will most likely want to whitebox specific procedures so that only the procedures needed by the application are loaded into the Neo4j instance at runtime.

Example: Restart with plugin

And here we see the results after restarting the Neo4j instance and running the script to list the procedures loaded in the instance:



Exercise #7: Install a plugin

In this Exercise, you will install the Spatial library for use by your Neo4j instance and you will create and execute a script to report all of the procedures available to the Neo4j instance.

Before you begin:

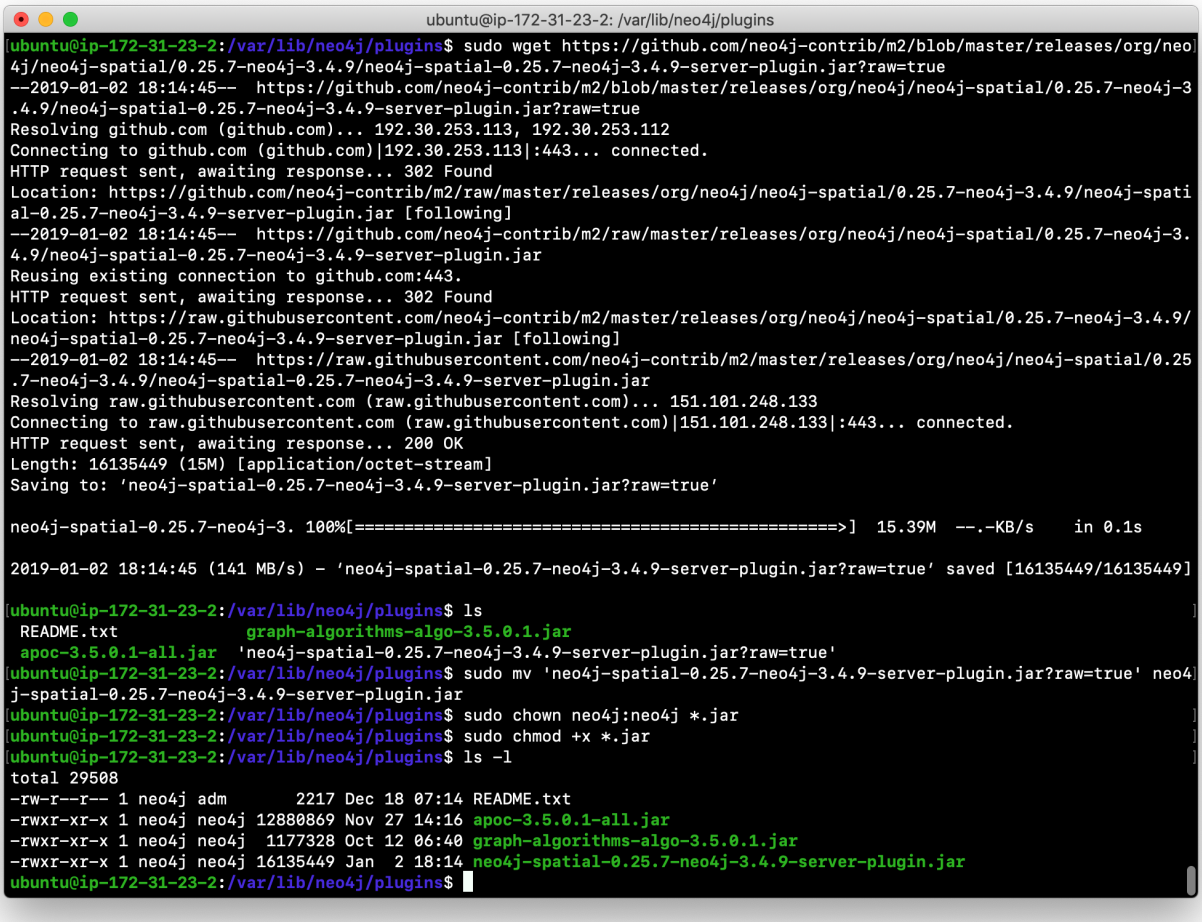
1. Stop the Neo4j instance.
2. Make sure you have a terminal window open for executing test commands.

Exercise steps:

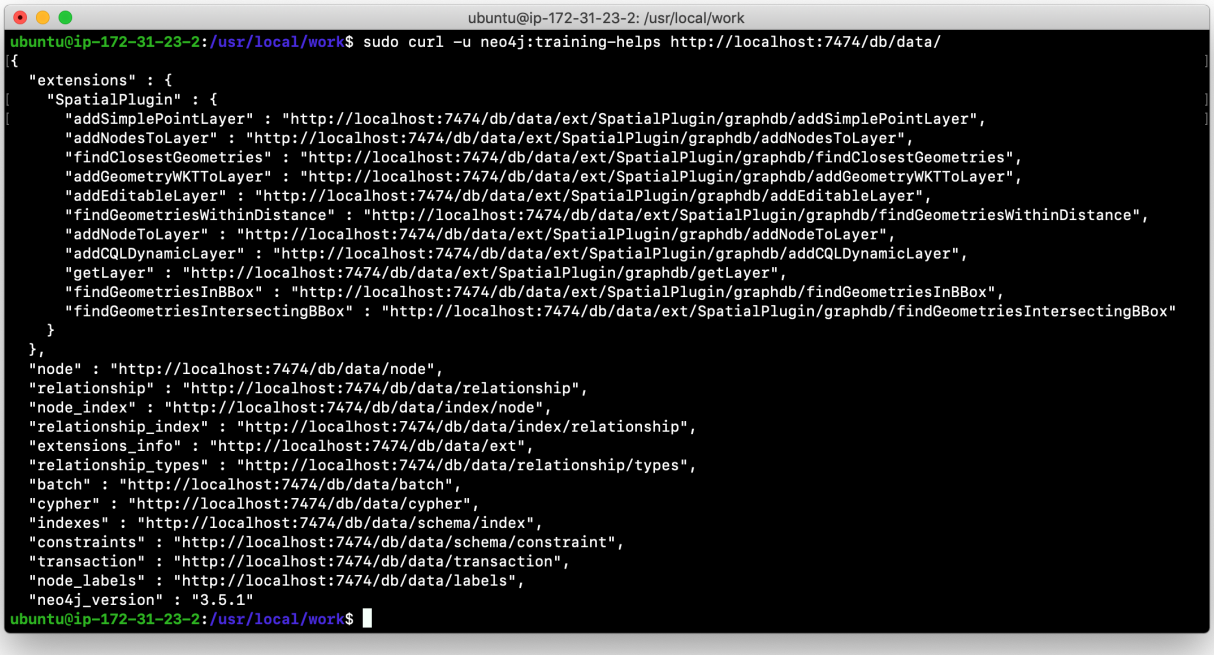
1. In a Web browser, go to the GitHub repository for the Neo4j Spacial Library.
2. On the main page for this repository, find the latest release of the library that is compatible with your version of Neo4j Enterprise Edition.
3. Download the already-built **.jar** file into the **/var/lib/neo4j/plugins** folder.
4. Ensure that the file size is correct and that the file name ends with **.jar**.
5. Change the owner of the **.jar** file to *neo4j:neo4j* and add execute permissions to the file.
6. Restart the Neo4j instance.
7. Follow the steps on the GitHub page for testing the library.

For example, you should see the following in the repository main page:

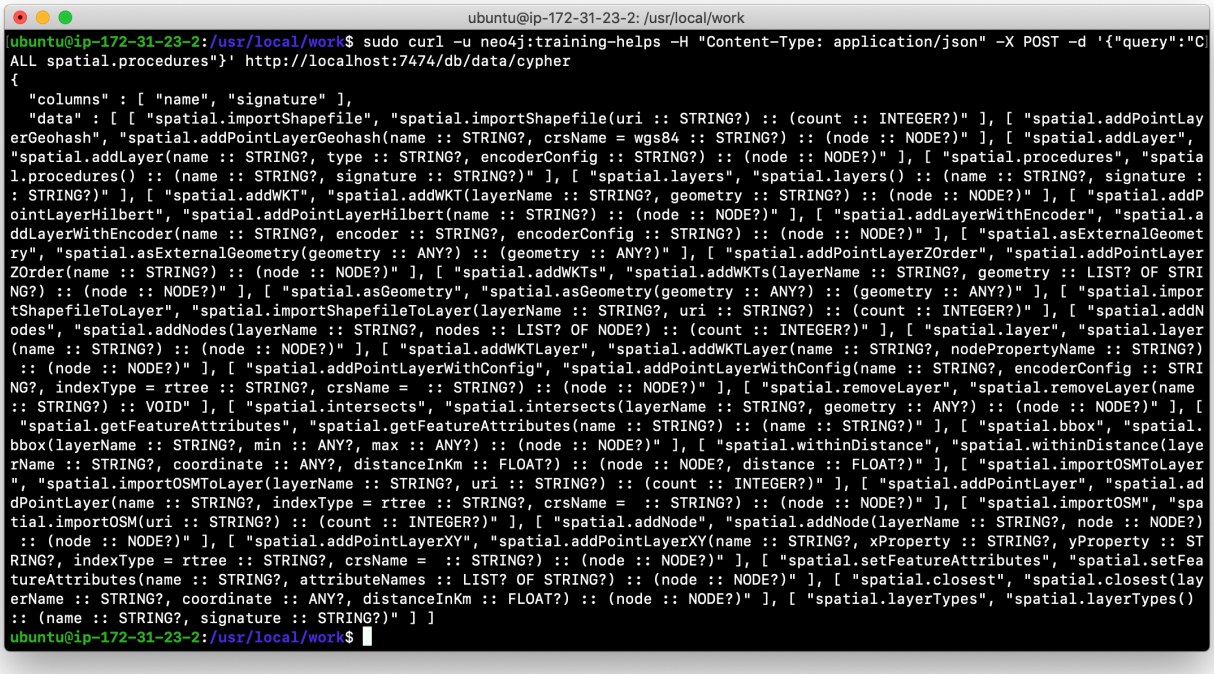
Here is how you download the **.jar** file into the **/var/lib/neo4j/plugins** folder. You should confirm that the file size is correct and that the owner is *neo4j:neo4j* with execute permissions.



Here is what you should see when you execute the first `curl` command:



Here is what you should see when you execute the second `curl` command:



8. In the **/usr/local/work** folder, create a script named **ListProcedures.sh** that will write the list of procedures available to the Neo4j instance to the **/usr/local/work/Procedures.txt** file.

9. Run the **ListProcedures.sh** script and examine the contents to also verify that the plugin has been installed. The **Procedures.txt** file should contain these items:

Configuring connector ports for the Neo4j instance

The Neo4j instance uses default port numbers that may conflict with other processes on your system. The ports frequently used are the connector ports:

Name	Port Number	Description
HTTP	7474	Used by Neo4j Browser and REST API. It is **not** encrypted so it should never be exposed externally.
HTTPS	7473	Used by REST API. Requires additional SSL configuration.
Bolt	7687	Bolt connection used by Neo4j Browser, cypher-shell, and client applications.

Modifying the default connector ports

If any of these ports conflict with ports already used on your system, you can change these connector ports by modifying these property values in the **neo4j.conf** file:

```
# Bolt connector
dbms.connector.bolt.enabled=true
#dbms.connector.bolt.tls_level=OPTIONAL
#dbms.connector.bolt.listen_address=:*7687*

# HTTP Connector. There can be zero or one HTTP connectors.
dbms.connector.http.enabled=true
#dbms.connector.http.listen_address=:*7474*

# HTTPS Connector. There can be zero or one HTTPS connectors.
dbms.connector.https.enabled=true
#dbms.connector.https.listen_address=:*7473*
```

As you learn more about some of the other administrative tasks for a Neo4j instance, you will work with other ports.

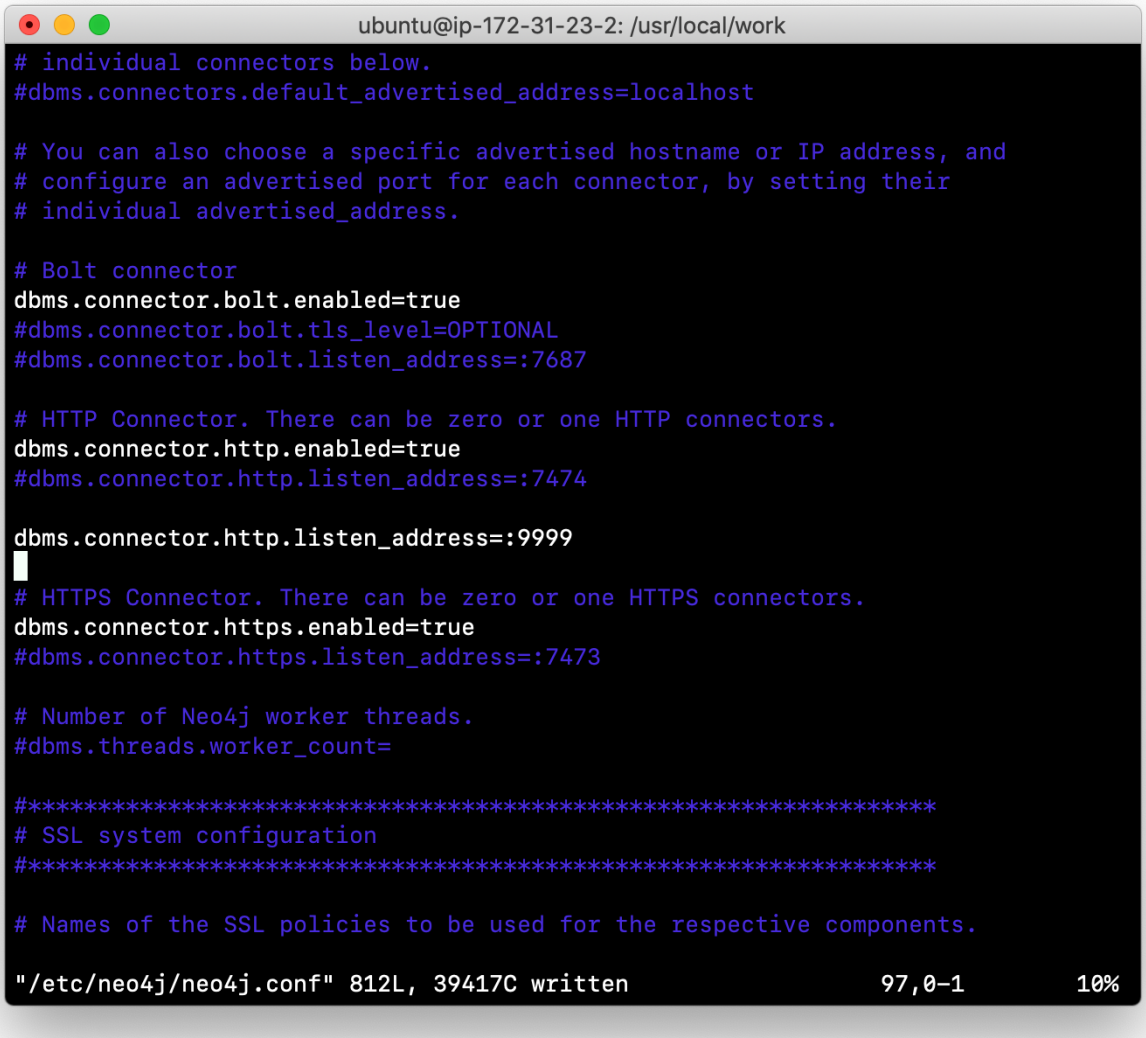
NOTE The REST API will be deprecated in Neo4j 4.0.

Exercise #8: Modify the HTTP port

In this Exercise, you will modify the default HTTP port used by the HTTP instance and use the new port.

Exercise steps:

1. Change the HTTP port to a value that is not in use on your system, for example **9999**. For example, your **neo4j.conf** file should look something like this:



2. Restart the Neo4j instance.

3. Confirm that the port works by entering the following `curl` command that uses the Neo4j HTTP API to create a node, where it will ask you for the password the *neo4j* user:

```
curl -v -H "Content-Type: application/json" -d '{ "statements" : [ { "statement"  :
"CREATE (n) RETURN id(n)" }]}' http://localhost:9999/db/data/transaction/commit -u
neo4j
```

4. Change the HTTP port back to its default (7474).
5. Restart the Neo4j instance.

Performing online backup and restore

Online backup is used in production where the application cannot tolerate the database being unavailable. In this part of the training, you will learn how to back up and restore a stand-alone Neo4j database. Later in this training, you will learn about backup and restore in a Neo4j Causal Cluster environment.

Enabling online backup

To enable a Neo4j instance to be backed up online, you must add these two properties to your **neo4j.conf** file:

```
dbms.backup.enabled=true
dbms.backup.address=<host-address>:<6362-6372>
```

Where *host-address* is the address of a server from which you will run the backup tool from. You must specify a port number that will not conflict with existing ports used on the backup server.

A best practice for online backup of a stand-alone production database is to perform the backup on a different server. This is because the backup process and consistency checking is expensive and you want to offload this to another server.

A common practice for many enterprises is to back up their databases to Amazon S3 sites. In addition, if any backups are to be stored in S3, they should be encrypted as well as the channel used to create send the backup to S3.

Performing the backup

After you restart the Neo4j instance, you can then perform the backup on the server you specified in *host_address* as follows with consistency checking:

```
neo4j-admin backup --backup-dir=<backup-folder>
                   --name=<backup-instance-folder-name>
                   --from=<Neo4j-instance-host-address:<port>
                   --check-consistency=true
                   --cc-report-dir=<report-directory>
```

This will perform a full backup to *backup-instance-folder-name* for the Neo4j instance running on *Neo4j-instance-host-address*.

Restoring from a backup

If you need to restore a database from a backup, you must first stop the Neo4j instance. Since the instance is down, you can restore the database on the same server that runs the instance, provided the server has access to the backup location in the network.

Here is how you restore the database from a backup:

```
neo4j-admin restore
          --from=<absolute-path-to-backup-instance-folder-name>
          --database=<database-name>
          --force=true
```

You specify *true* for force so that the existing database will be replaced.

NOTE	If you restore a database as *root*, make sure that you change the ownership (recursively) of the database directory to *neo4j:neo4j* before starting the Neo4j instance.

There are many ways for performing online backups, including incremental backups. See the Neo4j Operations Manual for details.

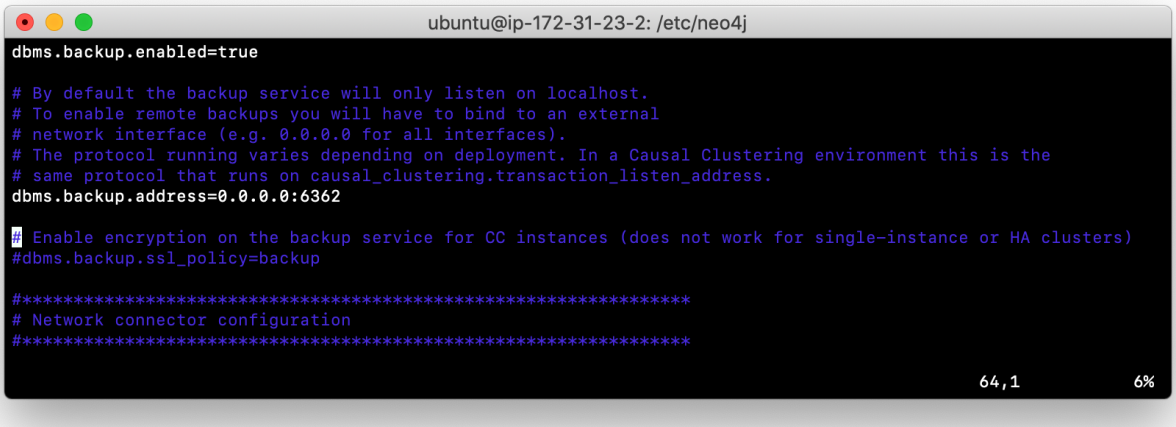
Exercise #9: Performing online backup and restore

In this Exercise, you will perform an online backup of your database where you use the same host for the backup process. Then you will modify the database. Finally, you will restore the database from the backup.

NOTE	In your real application, if you were to back up a production stand-alone Neo4j instance, you would use a different host from the host that is running the Neo4j instance.

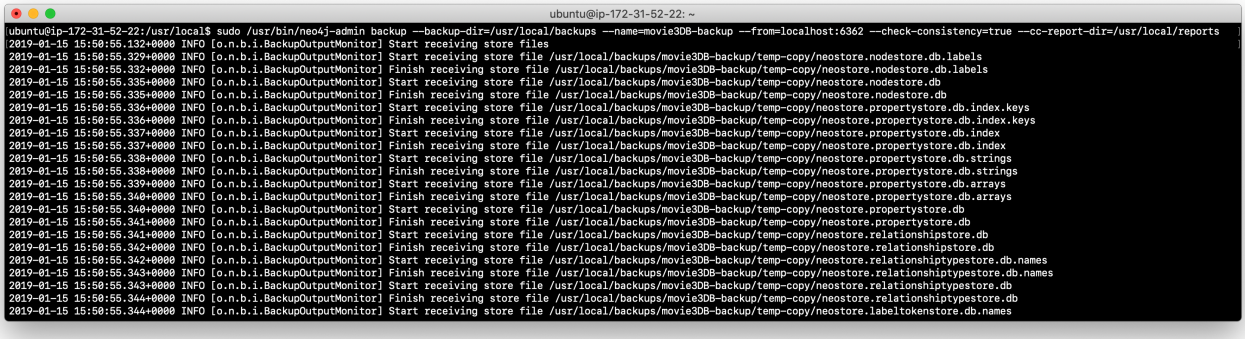
Exercise steps:

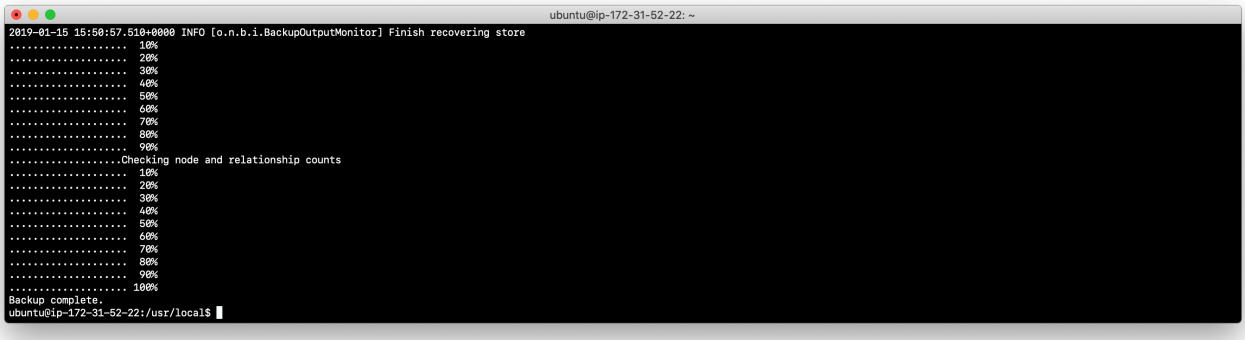
1. Stop the Neo4j instance.
2. Modify the Neo4j configuration so that online backup is enabled and will be done on this same host. For example, your **neo4j.conf** file should look something like this:



3. Restart the Neo4j instance.
4. Create a folder named **/usr/local/backup** and ensure that it is owned by *neo4j:neo4j*.

5. Perform an online backup of the active database (**movie3.db**). The result of the backup should look something like this:

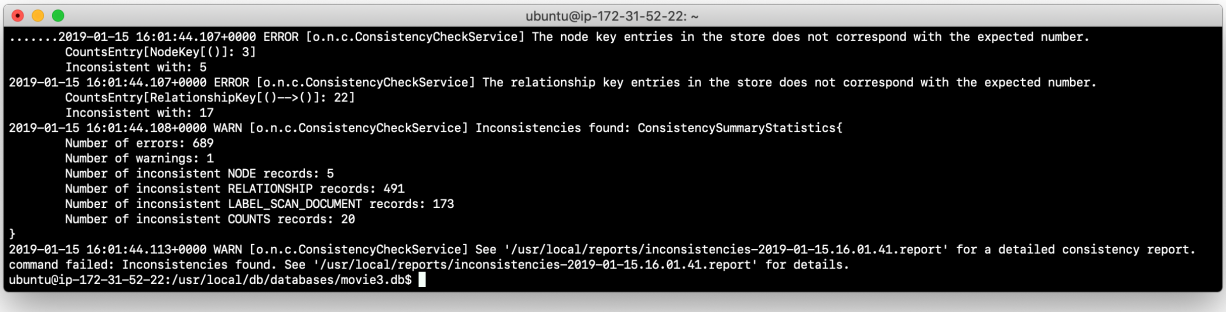




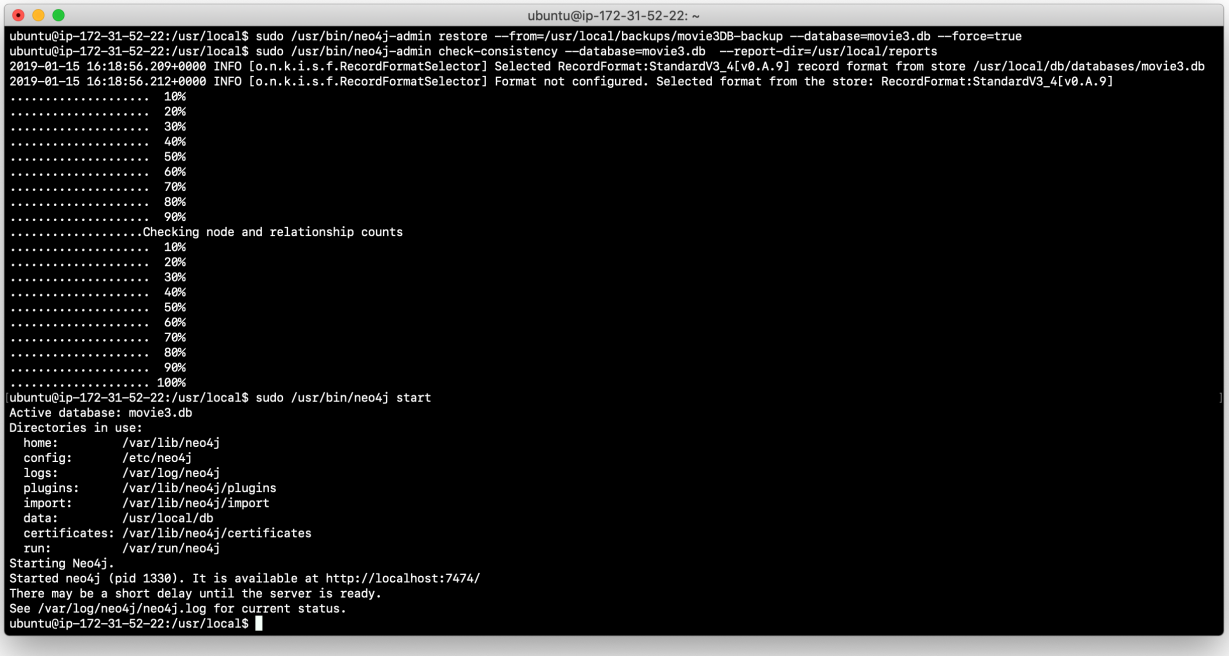
6. Stop the Neo4j instance.
7. Corrupt the database like you did earlier in this module. Modify the file **movie3.db/neostore.nodestore.db** by adding some text to the file.
8. Run the consistency check tool on **movie3.db** using `neo4j-admin` specifying **/usr/local/work/reports** as the folder where the report will be written.

```
neo4j-admin check-consistency --database=movie3.db --report-dir=/usr/local/reports
--verbose=true
```

9. The consistency check tool should return that inconsistencies were found.



10. Restore the **movie3.db** database from the backup.
11. Check its consistency.



11. Confirm that the Neo4j instance starts without error.

Using the import tool to create a database

The course, *Introduction to Neo4j*, teaches you how to import **.csv** data using `LOAD CSV` in Cypher. `LOAD CSV` works fine for datasets containing fewer than 10M nodes. For large datasets, it may also be possible to import the data with some of the APOC procedures.

Data import for a graph database is resource-intensive because it needs to pre-compute joins (relationships) between records (nodes). For large datasets, a best practice is to import the data using the `import` command of the `neo4j-admin` tool. This tool creates the database from a set of **.csv** files.

You can read details about using the import tool in the Neo4j Operations Manual.

Creating CSV files for the import

The format of the **.csv** files is important. For both nodes and relationships, header information must be associated with the data. Header information contains an ID to uniquely identify the record, optional node labels or relationship types, and names for the properties representing the imported data. A **.csv** can have a header row, or you can place the header information in a separate file.

In this training, you will use data that has been created for you that represents crimes.

CSV files for nodes

Here is portion of the **beats.csv** file with embedded header information for loading nodes of type *Beat*:

```
:ID(Beat),id,:LABEL
1132,1132,Beat
0813,0813,Beat
0513,0513,Beat
```

The **beats.csv** records represent data that will be loaded into a node with the label *Beat*. In this example the record ID is the same as the *id* property value that will be used to create the node in the graph.

Here is an example of the **crimes_header.csv** header file for loading nodes of type *Crime*:

```
:ID(Crime),id,:LABEL,date,description
```

The nodes loaded with **Crimes_header.csv** will have the label, *:LABEL*. In addition, the data in the associated **crimes.csv** file will have values for the ID of the record, and property values for *id*, *date*, and *description*.

And here is a portion of the associated **crimes.csv** file for loading nodes of type *Crime*:

```
8920441,8920441,Crime,12/07/2012 07:50:00 AM,AUTOMOBILE
4730813,4730813,Crime,05/09/2006 08:20:00 AM,POCKET-PICKING
7150780,7150780,Crime,09/28/2009 01:00:00 AM,CHILD ABANDONMENT
4556970,4556970,Crime,12/16/2005 08:39:24 PM,POSS: CANNABIS 30GMS OR LESS
9442492,9442492,Crime,12/28/2013 12:15:00 PM,OVER $500
```

In addition, this dataset includes information about the types of crimes. These nodes are created without a label for the node, but their ID, *PrimaryType* will be used to link them to *Crime* nodes. Here is a portion of the **primaryTypes.csv** file for loading these nodes:

```
:ID(PrimaryType),crimeType
ARSON,ARSON
OBSCENITY,OBSCENITY
ROBBERY,ROBBERY
THEFT,THEFT
CRIM SEXUAL ASSAULT,CRIM SEXUAL ASSAULT
BURGLARY,BURGLARY
```

CSV files for relationships

.csv files for loading relationships contain a row for every relationship where the ID for the starting and ending node is specified, as well as the relationship type. If you do not specify the relationship in the file, then you must specify it in the arguments to the import tool.

Here is a portion of the **crimesBeats.csv** file that will be used to create the *:ON_BEAT* relationships between *Crime* and *Beat* nodes:

```
:START_ID(Crime),:END_ID(Beat),:TYPE
6978096,0911,ON_BEAT
3170923,2511,ON_BEAT
3073515,1012,ON_BEAT
8157905,0113,ON_BEAT
```

Here is a portion of a portion of the **crimesPrimaryTypes.csv** file that will be used to create the relationships between the *Crime* nodes and the nodes that contain the *CrimeType* data:

```
:START_ID(Crime),:END_ID(PrimaryType)
5221115,NARCOTICS
4522835,DECEPTIVE PRACTICE
3432518,BATTERY
6439993,CRIMINAL TRESPASS
```

The relationship, *:TYPE* is not specified in this file so it will be specified in the arguments when you load the data from this file.

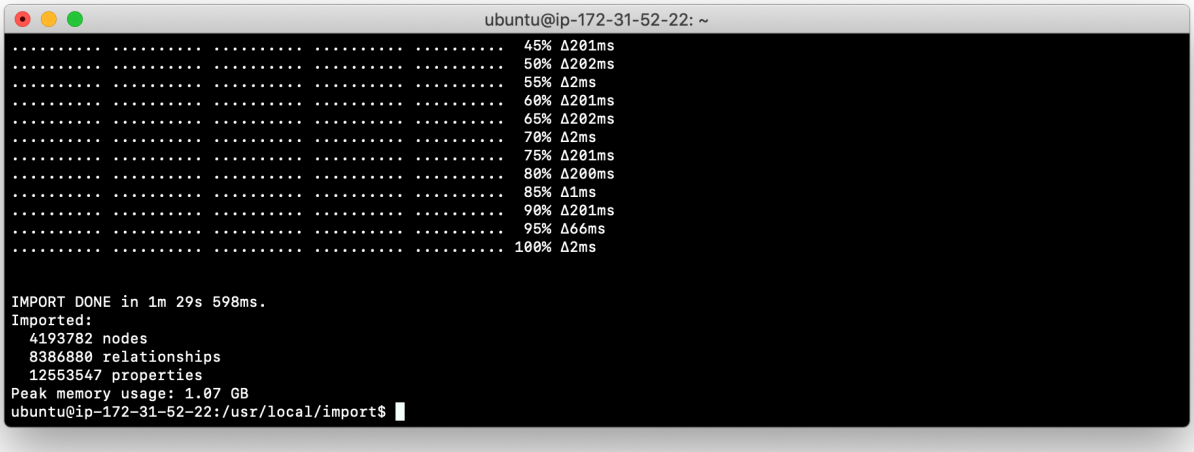
Importing the data

After you have created or obtained the **.csv** files for the data, you import the data. The data import creates a database and you must run the import tool with the Neo4j instance stopped.

Here is the simplified syntax for creating a database from **.csv** files:

```
neo4j-admin import  --database <database-name>
                    --nodes[:Label1:Label2] [<rheader-csv-file-1>,]<csv-file-1>
                    --nodes[:Label1:Label2] [<rheader-csv-file-2>,]<csv-file-2>
                    --nodes[:Label1:Label2] [<rheader-csv-file-n>,]<csv-file-n>
                    --relationships[:REL_TYPE] [<jheader-csv-file-1>,]<join-csv-file-
1>
                    --relationships[:REL_TYPE] [<jheader-csv-file-2>,]<join-csv-file-
2>
                    --relationships[:REL_TYPE] [<jheader-csv-file-n>,]<join-csv-file-
n>
                    --report-file <report-file-path>
```

Here is the result of using the `import` command of `neo4j-admin` to create a database and import **.csv** files.



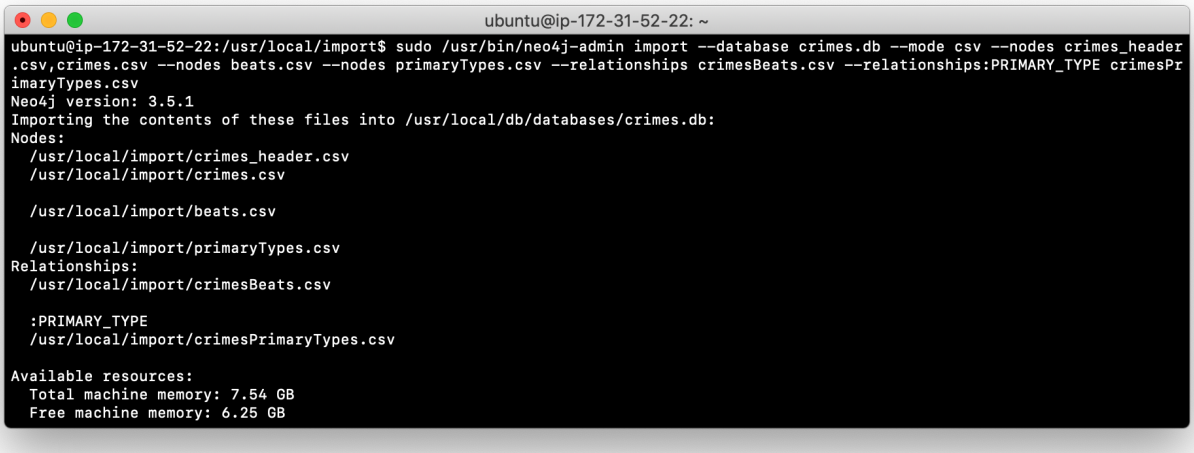
Exercise #10: Importing data with the import command

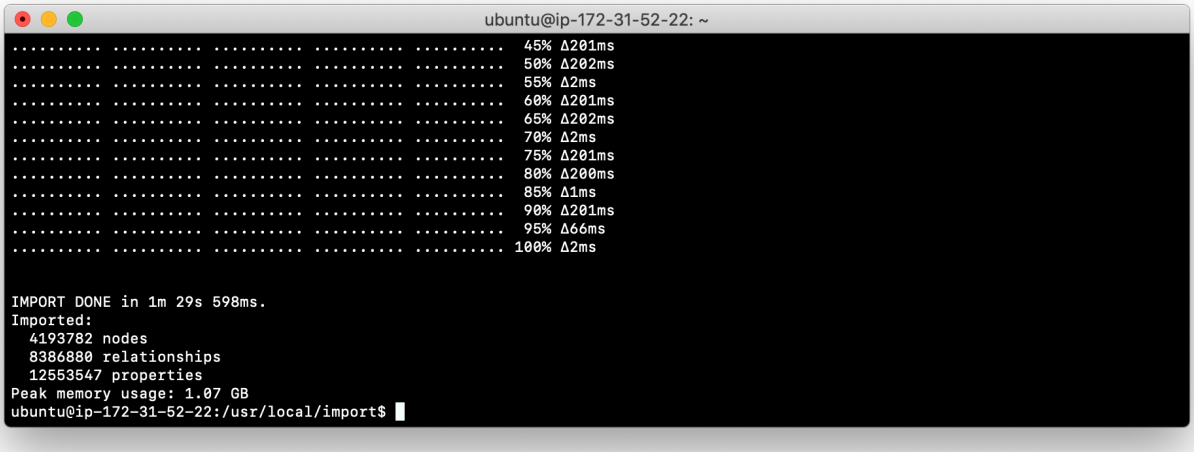
In this Exercise, you create a new database by importing data using **.csv** files. Data import is very common when preparing a database for production where the data originally comes from relational tables.

Exercise steps:

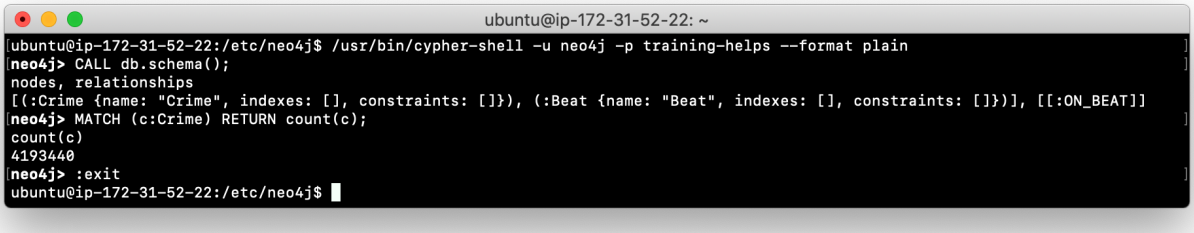
1. Stop the Neo4j instance.
2. In a terminal window, create the **/usr/local/import** folder.
3. Navigate to the **import** folder and download this file: https://s3-us-west-1.amazonaws.com/data.neo4j.com/admin-neo4j/crime-data.zip. **Hint**: use `curl -O` or `wget`.
4. Unzip the file. You should have six **.csv** files.
5. Examine the contents of the files to become familiar with their format and data.
6. Use the `import` command to import the data into a new database named **crimes.db**, using these guidelines:

```
--nodes crimes_header.csv,crimes.csv
--nodes beats.csv
--nodes primaryTypes.csv
--relationships crimesBeats.csv
--relationships:PRIMARY_TYPE crimesPrimaryTypes.csv
```





7. Modify the **neo4j.conf** file to use **crimes.db** as the active database.
8. Ensure that the ownership of the **crimes.db** directory and everything under it is owned by *neo4j:neo4j*.
9. Start the Neo4j instance.
10. Run `cypher-shell` to retrieve the schema of the database and also count the number of *Crime* nodes in the graph.



Check your understanding

Question 1

Suppose that you have installed Neo4j Enterprise Edition and have modified the name of the active database in the Neo4j configuration file. What tool and command do you run to create the new database?

Select the correct answer.

- ☐ `neo4j-admin create-database`
- ☐ `neo4j-admin initialize`
- ☐ `neo4j create-database`
- ☐ `neo4j start`

Question 2

Suppose that you want the existing Neo4j database to have the name **ABCRecommendations.db**. Assuming that you have stopped the Neo4j instance, what steps must you perform to modify the name of the database, which currently has a default name of **graph.db**?

Select the correct answers.

- ☐ Rename the **NEO4J_HOME/graph.db** folder to **NEO4J_HOME/ABCRecommendations.db**.
- ☐ Modify **neo4j.conf** to use *dbms.active_database=ABCRecommendations.db*.
- ☐ Run `neo4j-admin rename graph.db ABCRecommendations.db`.
- ☐ Run `neo4j-admin move graph.db ABCRecommendations.db`.

Question 3

How do you copy a database that you want to give to another user?

Select the correct answer.

- ☐ With the Neo4j instance started, run `neo4j-admin copy` providing the location where the copy will be created.
- ☐ With the Neo4j instance stopped, run `neo4j-admin copy` providing the location where the copy will be created.
- ☐ With the Neo4j instance started, run `neo4j-admin dump` providing the location where the dump file will be created.
- ☐ With the Neo4j instance stopped, run `neo4j-admin dump` providing the location where the dump file will be created.

Summary

You should now be able to:

- Start a Neo4j instance.
- Stop the Neo4j instance.
- Set the password for the *neo4j* user.
- Copy a Neo4j database.
- Modify the location for a Neo4j database.
- Check the consistency of a Neo4j database.
- Create scripts for modifying a Neo4j database.
- Manage plugins for a Neo4j database.
- Configure ports used by the Neo4j instance.
- Perform an online backup of a Neo4j database.
- Create a database with the import tool.

Causal Clustering in Neo4j

Table of Contents

About this module

Now that you have gained experience managing a Neo4j instance and database , you will learn how to get started with creating and managing Neo4j Causal Clusters.

At the end of this module, you should be able to:

- Describe why you would use clusters.
- Describe the components of a cluster.
- Configure and use a cluster.
- Seed a cluster with data.
- Monitor and manage core servers in the cluster.
- Monitor and manage read replica servers in the cluster.
- Back up a cluster.

This module covers the basics of clusters in Neo4j. Later in this training, you will learn more about security and encryption related to clusters.

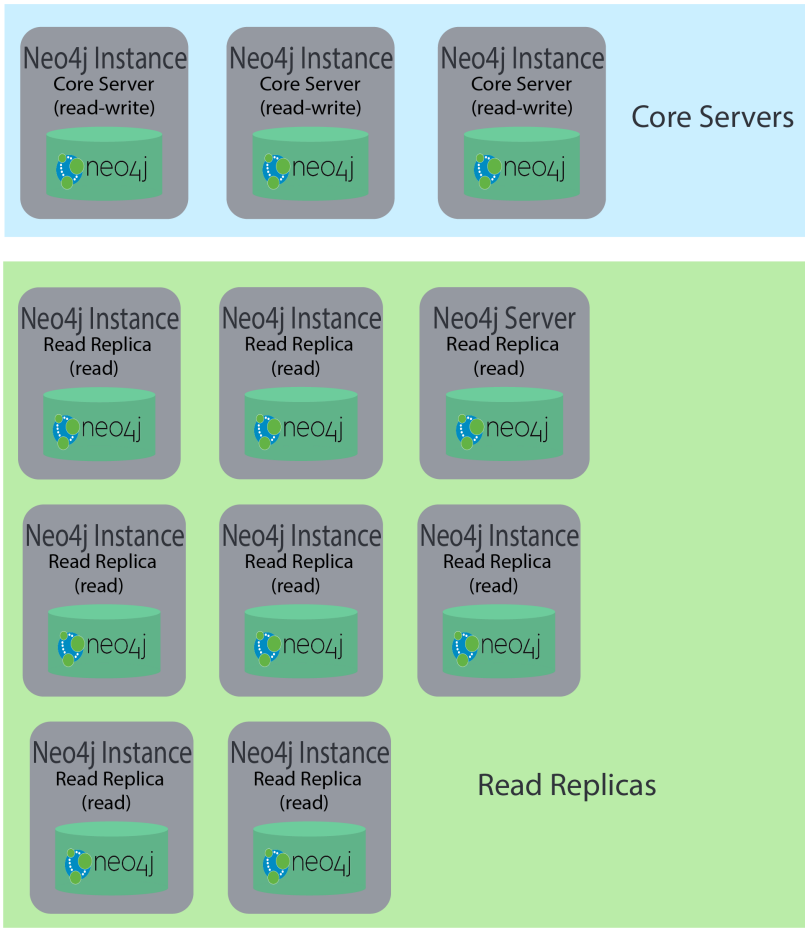
What is Clustering?

Neo4j's clustering architecture enables an enterprise to utilize a Neo4j database in production. First, it provides a high-available solution whereby if a Neo4j instance has a failure, another Neo4j instance can take over automatically. Secondly, it provides a highly-scalable database whereby some parts of the application update the data, but other parts of the application are widely distributed and do not need immediate access to new or updated data. That is, read latency is acceptable. Causal consistency in Neo4j means that an application is guaranteed to be able to consistently read all data that it has written.

Most Neo4j applications use clustering to ensure high-availability. The scalability of the Neo4j database is used by applications that utilize multiple data centers.

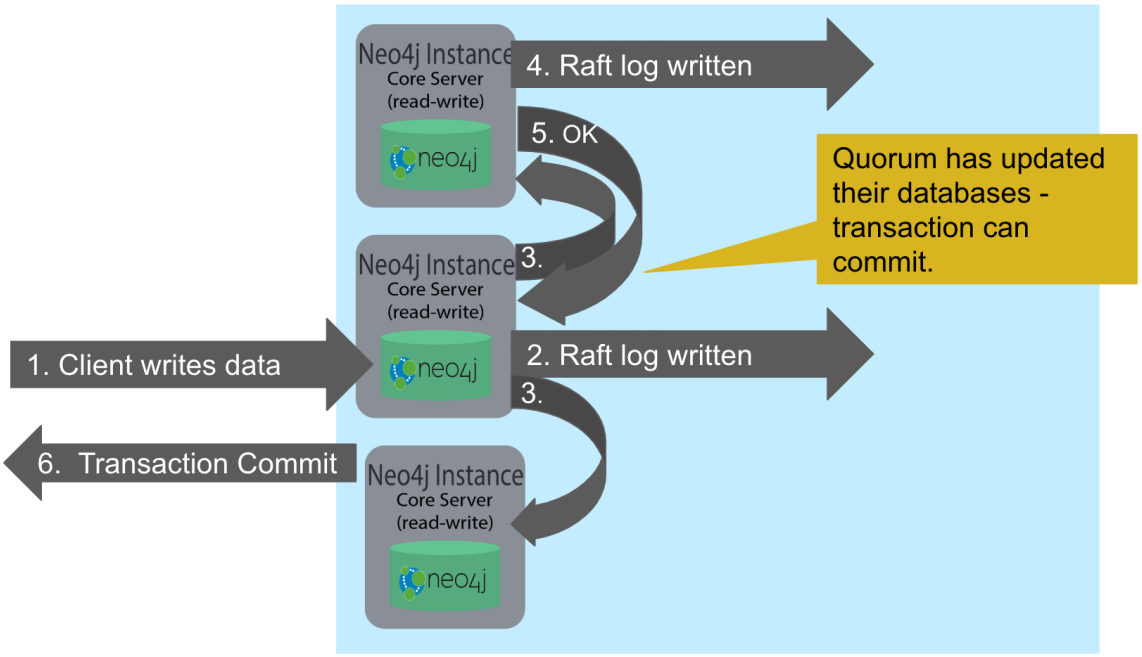
Cluster architecture

There are two types of Neo4j instances in a cluster architecture: core servers and read replica servers.



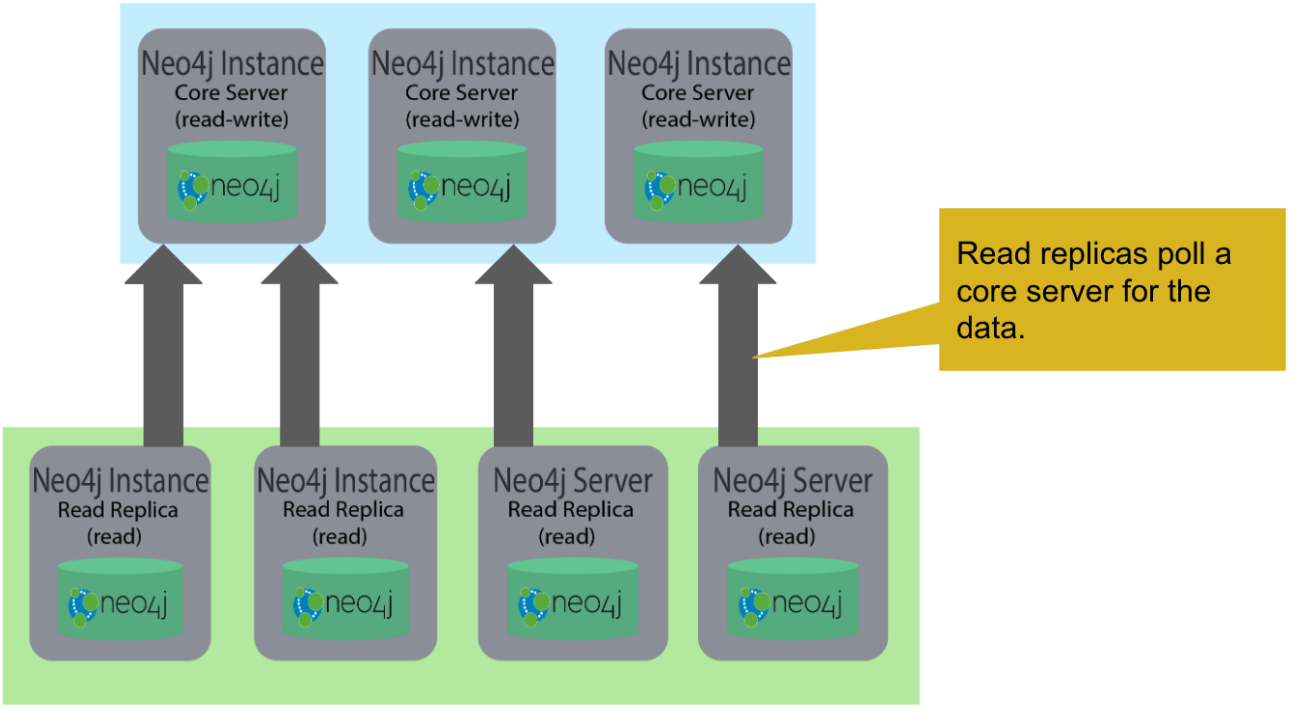
Core servers

Core servers are used for read and write access to the database. The core servers are used to synchronize updates to the database, regardless of the number and physical locations of the Neo4j instances. By default, in a cluster architecture, a transaction is committed if a majority (*quorum*) of the core servers defined as the minimum required for the cluster have written the data to the physical database. This coordination between core servers is implemented using the *Raft* protocol. You can have a large number of core servers, but the more core servers in the application architecture, the longer a "majority" commit will take. At a minimum, an application should use three core servers to be considered fault-tolerant. If one of the three servers fail, the cluster is still operable for updates to the database. If you want an architecture that can support two servers failing, then you must configure five core servers. You cannot configure a cluster with two core servers because if one server fails, the second server is automatically set to be read-only, leaving your database to be inoperable for updates.



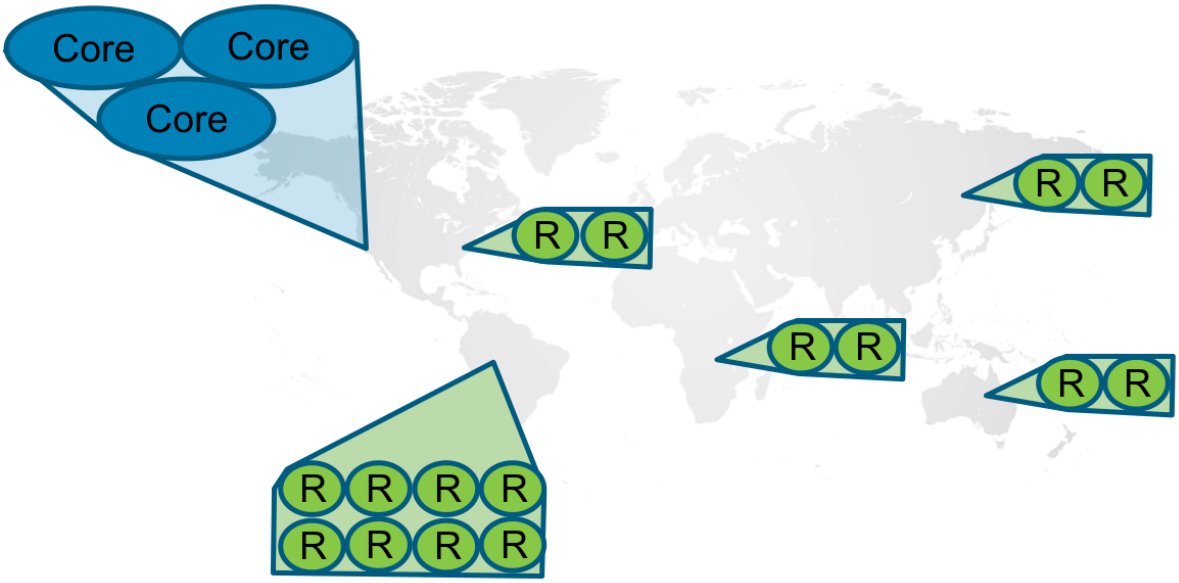
Read replica servers

Read replica servers are used to scale data across a distributed network. They only support read access to the data. The read replica servers regularly poll the core servers for updates to the database by obtaining the transaction log from a core server. You can think of a read replica as a highly scalable and distributed cache of the database. If a read replica fails, a new read replica can be started with no impact on the data and just a slight impact for the application that can be written to reconnect to a different read replica server.



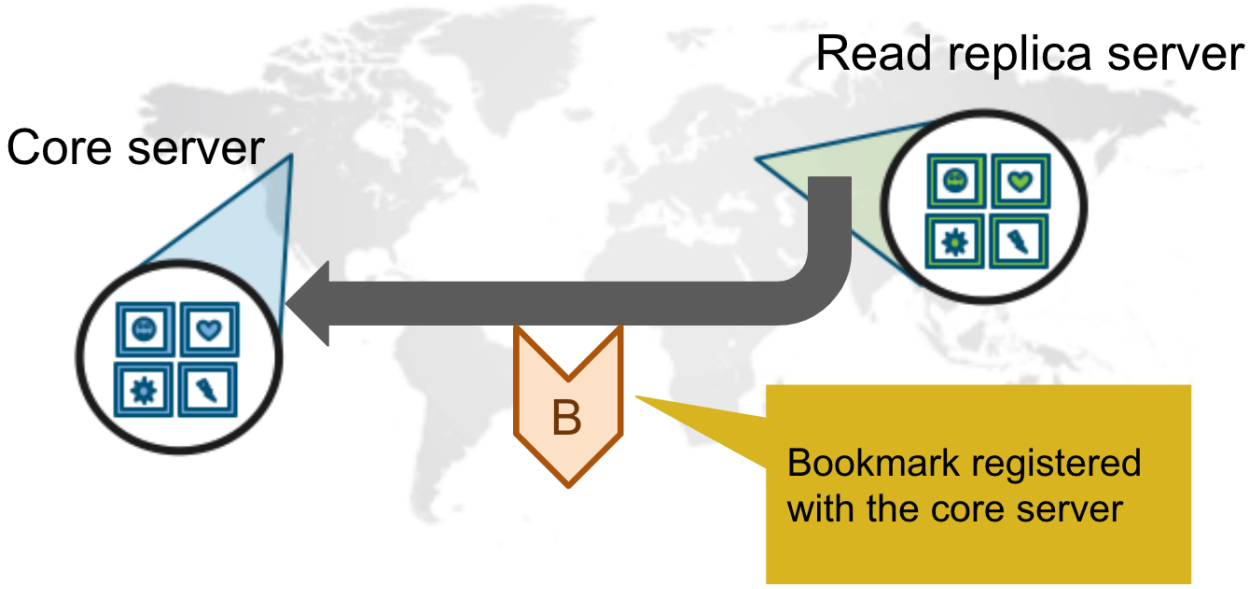
Distributed architecture

Here is an example where the core servers are located in one data center, but the read replicas are located in many distributed data centers.



Bookmarks with Causal Clustering

An application can create a bookmark that is used to mark the the last transaction committed to the database. In a subsequent read, the bookmark can be used to ensure that the appropriate core servers are used to ensure that only committed data will be read by the application.



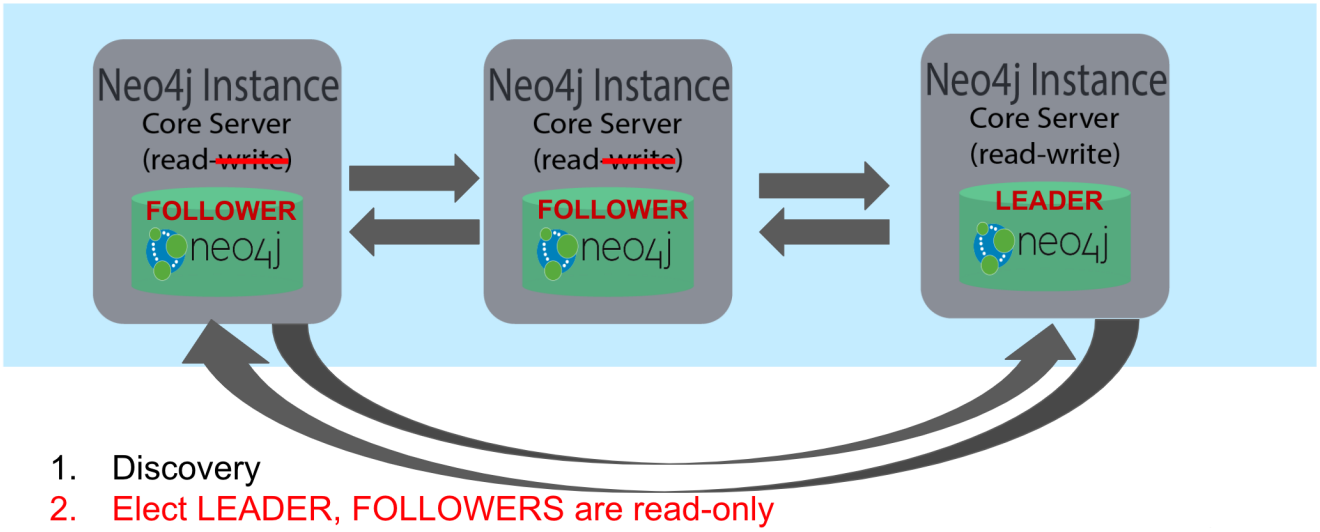
Configuring clustering

As an administrator, you must determine the physical locations of the servers that will be used as core servers and read replica servers. You configure the casual cluster by updating the **neo4j.conf** file on each server so that they can operate together as a cluster. The types of properties that you configure for a cluster include, but are not limited to:

- Whether the server will be a core server or a read replica server
- Public address for the server
- Names/addresses of the servers in the core server membership
- Ports used for communicating between the members
- Published ports for bolt, http, https (non-conflicting port numbers)
- Number of core servers in the cluster

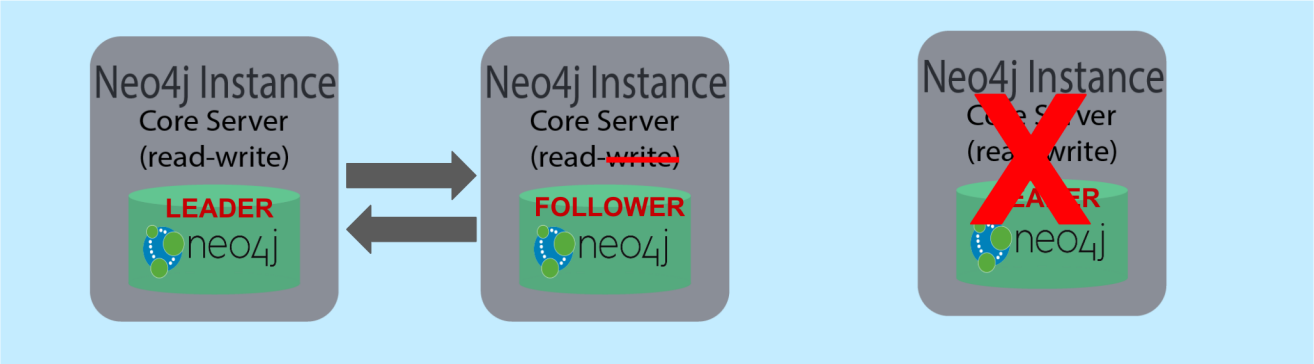
Core server startup

When a core server starts, it first uses a discovery protocol to join the network. At some point it will be running with the other members of the core membership. In a cluster, exactly one core server is elected to be the *LEADER*. The *LEADER* is the coordinator of all communication between the core servers. All of the other core servers are *FOLLOWERS* as the servers in the cluster use the *Raft* protocol to synchronize updates. If a core server joins the network after the other core servers have been running and updating data, the late-joining core server must use the *catchup* protocol to get to a point where it is synchronized as the other *FOLLOWERS* are.



Core server shutdown

When a core server shuts down, the shutdown may be initiated by an administrator, or it may be due to a hardware or network failure. If the core server that is a *FOLLOWER* shuts down, the *LEADER* detects the shutdown and incorporates this information into its operations with the other core servers. If the core server that is the *LEADER* shuts down, the remaining core servers communicate with each other and an existing *FOLLOWER* is promoted to the *LEADER*.



1. Server shutdown
2. Elect new LEADER

Cluster below quorum

If a core server shutdown leaves the cluster below a configured threshold for the number of core servers required for the cluster, then the *LEADER* becomes inoperable for writing to the database. This is a serious matter that needs to be addressed by you as the administrator.

1. Server shutdown
2. Below quorum - no LEADER, cluster inoperable for writes

Core server updates database

A core server updates its database based upon the requests from clients. The client's transaction is not complete until a quorum of core servers have updated their databases. Subsequent to the completion of the transaction, the remaining core servers will also be updated. Core servers use the *Raft* protocol to share updates. Application clients can use the *bolt* protocol to send updates to a particular core server's database, but the preferred protocol for an cluster is the *bolt+routing* protocol. With this protocol, applications can write to any core server in the cluster, but the *LEADER* will always coordinate updates.

Administrative tasks for clustering

Here are some common tasks for managing and monitoring clustering:

1. Modify the **neo4j.conf** files for each core server.
2. Start the core servers in the cluster.
3. Seed the core server (add initial data).
4. Ensure each core server has the data.
5. Modify the **neo4j.conf** files for each read replica server.
6. Start the read replica servers.
7. Ensure each read replica server has the data.
8. Test updates to the database.

In your real application, you set up the core and read replica Neo4j instances on separate physical servers that are networked and where you have installed Enterprise Edition of Neo4j. In a real application, all configuration for clustering is done by modifying the **neo4j.conf** file.

Configuring core servers

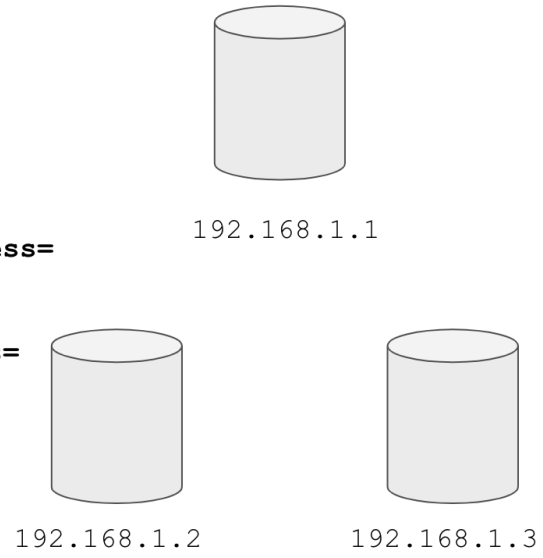
Please refer to the Neo4j Operations Manual for greater detail about the settings for configuring clustering.

Identify core servers

When setting up clustering, you should first identify at least three machines that will host core servers. For these machines, you should make sure these properties are set in **neo4j.conf** where XXXX is the IP address of the machine on the network and XXX1, XXX2, XXX3 are the IP addresses of the machines that will participate in the cluster. These machines must be network accessible.

Example for server: 192.168.1.1:

```
dbms.mode=CORE
causal_clustering.raft_listen_address=
   192.168.1.1:7000
causal_clustering.transaction_listen_address=
   192.168.1.1:6000
causal_clustering.discovery_listen_address=
   192.168.1.1:5000
```



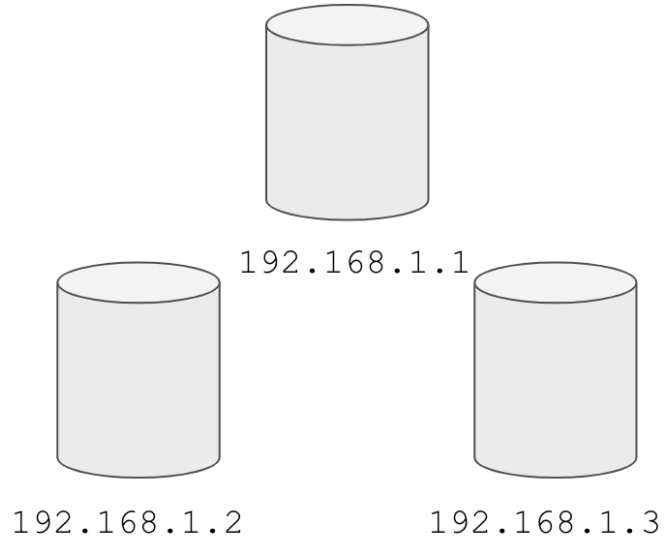
Specify cluster membership

The machines that you designate to run core servers must be reachable from each other. This means that the core machines are part of the membership of the cluster:

On all core servers:

causal_clustering.initial_discovery_members=

```
192.168.1.1:5000,192.168.1.2:5000,192.168.1.3:5000
```



Example: Configuration properties

Here are some of the settings that you may use for your core servers, depending on whether the addresses are known in the network. You may have to specify advertised addresses in addition to the actual addresses.

```
# set this if you want to ensure the host can be accessed from external browsers
dbms.connectors.default_listen_address=0.0.0.0

# these are the default values used for virtually all configs
dbms.connector.https.listen_address=0.0.0.0:7473
dbms.connector.http.listen_address=0.0.0.0:7474
dbms.connector.bolt.listen_address=0.0.0.0:7687

# used by application clients for accessing the instance
dbms.connector.bolt.advertised_address=localhost:18687

causal_clustering.transaction_listen_address=0.0.0.0:6000
causal_clustering.transaction_advertised_address=XXXX:6000

causal_clustering.raft_listen_address=0.0.0.0:7000
causal_clustering.raft_advertised_address=XXXX:7000

causal_clustering.discovery_listen_address=0.0.0.0:5000
causal_clustering.discovery_advertised_address=XXXX:5000

# all members of the cluster must have this same list
causal_clustering.initial_discovery_members=XXX1:5000,XXX2:5000,XXX3:5000,XXX4:5000,XX
X5:5000

# 3 is the default if you do not specify these properties
causal_clustering.minimum_core_cluster_size_at_formation=3
causal_clustering.minimum_core_cluster_size_at_runtime=3

dbms.mode=CORE
```

Minimum cluster sizes

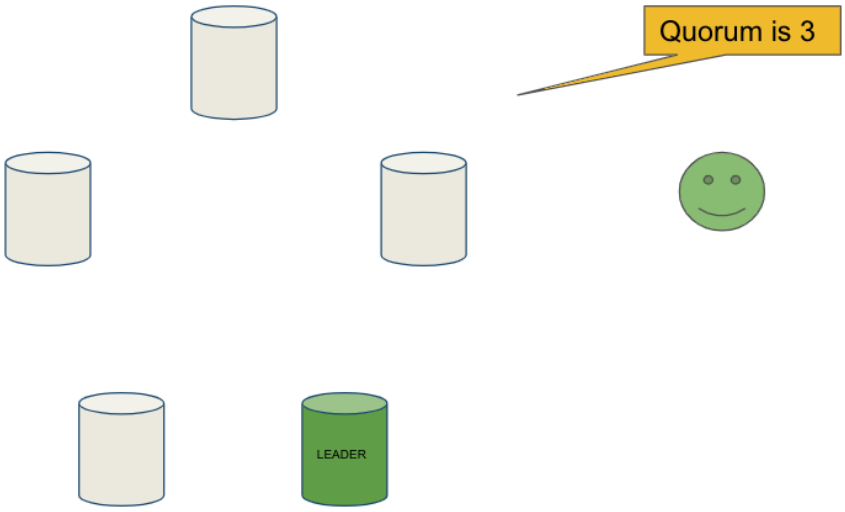
The *minimum_core_cluster_size_at_formation* property specifies the number of core servers that must be running before the database is operable for updates. These core servers, when started, ensure that they are caught up with each other. After all core servers are caught up, then the cluster is operable for updates.

The *minimum_core_cluster_size_at_runtime* property specifies the number of servers that will actively participate in the cluster at runtime.

How runtime minimum is used for a cluster

If the number of core servers started at formation is greater than the number required at runtime, then some started core servers are not considered essential and the cluster can still be operable if some of the core servers stop running. In this example, the size at formation and the runtime minimum are different. Most deployments set these two properties to be the same.

```
causal_clustering.minimum_core_cluster__size_at_formation=5

causal_clustering.minimum_core_cluster__size_at_runtime=3
```



The minimum number of core servers at runtime in a fault-tolerant cluster is three, which is the default setting for clustering. If you require more than three core servers, you must adjust the values in the clustering configuration section where you specify the size and the members of the cluster.

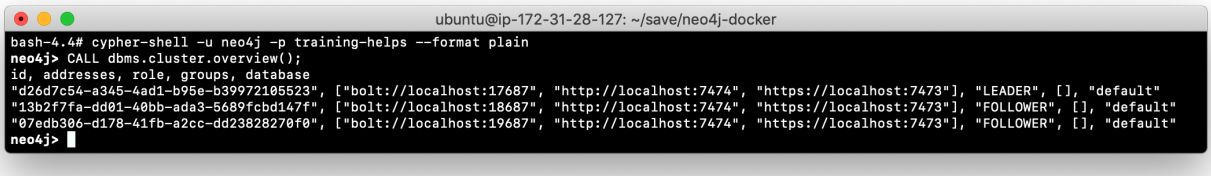
Starting the core servers

After you have modified the **neo4j.conf** files for the cluster, you start each Neo4j instance. When you start a set of core servers, it doesn't matter what order they are started. The cluster is not considered *started* until the number of core servers specified in *causal_clustering.minimum_core_cluster_size_at_formation* have started. One of the members of the core group will automatically be elected as the *LEADER*. Note that which core server is the *LEADER* could change at any time. You should observe the log output for each core server instance to ensure that it started with no errors.

NOTE	There is a configuration property (*causal_clustering.refuse_to_be_leader*) that you can set to true in the **neo4j.conf** file that specifies that this particular core server will never be a leader. It is not recommended that you set this property.

Viewing the status of the cluster

After you have started the core servers in the cluster, you can access status information about the cluster from `cypher-shell` on any of the core servers in the cluster. You simply enter `CALL dbms.cluster.overview();` and it returns information about the servers in the cluster, specifically, which ones are *FOLLOWERS* and which one is the *LEADER*.



Neo4j Enterprise Edition Docker image

For this training, you will gain experience managing and monitoring a Neo4j Causal Cluster using Docker. You will create and run Docker containers using a Neo4j Enterprise Docker image. This will enable you to start and manage multiple Neo4j instances used for clustering on your local machine. The published Neo4j Enterprise Edition 3.5.0 Docker image (from DockerHub.com) is pre-configured so that its instances can be easily replicated in a Docker environment that uses clustering. Using a Docker image, you create Docker containers that run on your local system. Each Docker container is a Neo4j instance.

For example, here are the settings in the **neo4j.conf** file for the Neo4j instance container named *core3* when it starts as a Docker container:

```
#*****************************************************************
# Other Neo4j system properties
#*****************************************************************
dbms.jvm.additional=-Dunsupported.dbms.udc.source=tarball
wrapper.java.additional=-Dneo4j.ext.udc.source=docker
ha.host.data=core3:6001
ha.host.coordination=core3:5001
dbms.tx_log.rotation.retention_policy=100M size
dbms.memory.pagecache.size=512M
dbms.memory.heap.max_size=512M
dbms.memory.heap.initial_size=512M
dbms.connectors.default_listen_address=0.0.0.0
dbms.connector.https.listen_address=0.0.0.0:7473
dbms.connector.http.listen_address=0.0.0.0:7474
dbms.connector.bolt.listen_address=0.0.0.0:7687
causal_clustering.transaction_listen_address=0.0.0.0:6000
causal_clustering.transaction_advertised_address=core3:6000
causal_clustering.raft_listen_address=0.0.0.0:7000
causal_clustering.raft_advertised_address=core3:7000
causal_clustering.discovery_listen_address=0.0.0.0:5000
causal_clustering.discovery_advertised_address=core3:5000
EDITION=enterprise
ACCEPT.LICENSE.AGREEMENT=yes
```

Some of these settings are for applications that use the *high availability (ha)* features of Neo4j. With clustering, we use the core servers for fault-tolerance rather than the high availability features of Neo4j. The setting *dbms.connectors.default_listen_address=0.0.0.0* is important. This setting enables the instance to communicate with other applications and servers in the network (for example, using a Web browser to access the http port for the server). Notice that the instance has a number of *causal_clustering* settings that are pre-configured. These are default settings for clustering that you can override when you create the Docker container for the first time. Some of the other default settings are recommended settings for a Neo4j instance, whether it is part of a cluster or not.

Creating Docker containers

When you create Docker Neo4j containers using `docker run`, you specify additional clustering configuration as parameters, rather than specifying them in the **neo4j.conf** file. Here is an example of the parameters that are specified when creating the Docker container named *core3*:

```
docker run --name=core3 \
        --volume=`pwd`/core3/conf:/conf --volume=`pwd`/core3/data:/data
--volume=`pwd`/core3/logs:/logs  \
        --publish=13474:7474 --publish=13687:7687 \
        --env=NEO4J_dbms_connector_bolt_advertised__address=localhost:13687 \
        --network=training-cluster \
        --env=NEO4J_ACCEPT_LICENSE_AGREEMENT=yes  \
        --env=NEO4J_causal__clustering_minimum__core__cluster__size__at__formation=3 \
        --env=NEO4J_causal__clustering_minimum__core__cluster__size__at__runtime=3 \
        --env=NEO4J_causal__clustering_initial__discovery__members=\
                  core1:5000,core2:5000,core3:5000,core4:5000,core5:5000 \
        --env=NEO4J_dbms_mode=CORE \
       --detach \
        b4ca2f886837
```

In this example, the name of the Docker container is *core3*. We map the conf, data, and logs folders for the Neo4j instance when it starts to our local filesystem. We map the http and bolt ports to values that will be unique on our system (13474 and 13687). We specify the bolt address to use. The name of the Docker network that is used for this cluster is *training-cluster*. *ACCEPT_LICENSE_AGREEMENT* is required. The size of the cluster is three core servers and the names of the [potential] members are specified as *core1*, *core2*, *core3*, *core4*, and *core5*. These servers use port *5000* for the discovery listen address. This instance will be used as a core server (*dbms.mode=CORE*). The container is started in this script detached, meaning that no output or interaction will be produced. And finally the ID of the Neo4j Enterprise 3.5.0 container is specified (*b4ca2f886837*). When you specify the Neo4j parameters for starting the container (`docker run`), you always prefix them with "--env=NEO4J_". In addition, you specify the underscore character for the dot character and a double underscore for the single underscore character instead of what you would use in the Neo4j configuration file.

NOTE When using the Neo4j Docker instance, a best practice is to specify more members in the cluster, but not require them to be started when the cluster forms. This will enable you to later add core servers to the cluster.

Exercise #1: Getting started with clustering

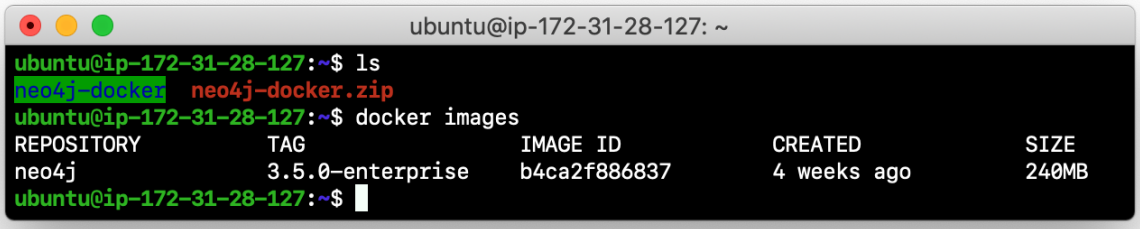
In this Exercise, you will gain experience with a simple cluster using Docker containers. You will not use Neo4j instances running on your system, but rather Neo4j instances running in Docker containers where you have installed Docker on your system.

Before you begin

1. Ensure that Docker Desktop (MAC/Windows) or Docker CE (Debian) is installed (`docker --version`). Here is information about downloading and installing Docker.
2. Download the file neo4j-docker.zip and unzip it to a folder that will be used to saving Neo4j configuration changes for clusters. This will be your working directory for the cluster Exercises in this training. **Hint:** `curl -O https://s3-us-west-1.amazonaws.com/data.neo4j.com/admin-neo4j/neo4j-docker.zip`
3. Download the Docker image for Neo4j (`docker pull neo4j:3.5.0-enterprise`).
4. Ensure that your user ID has docker privileges: `sudo usermod -aG docker <username>`. You will have to log in and log out to use the new privileges.

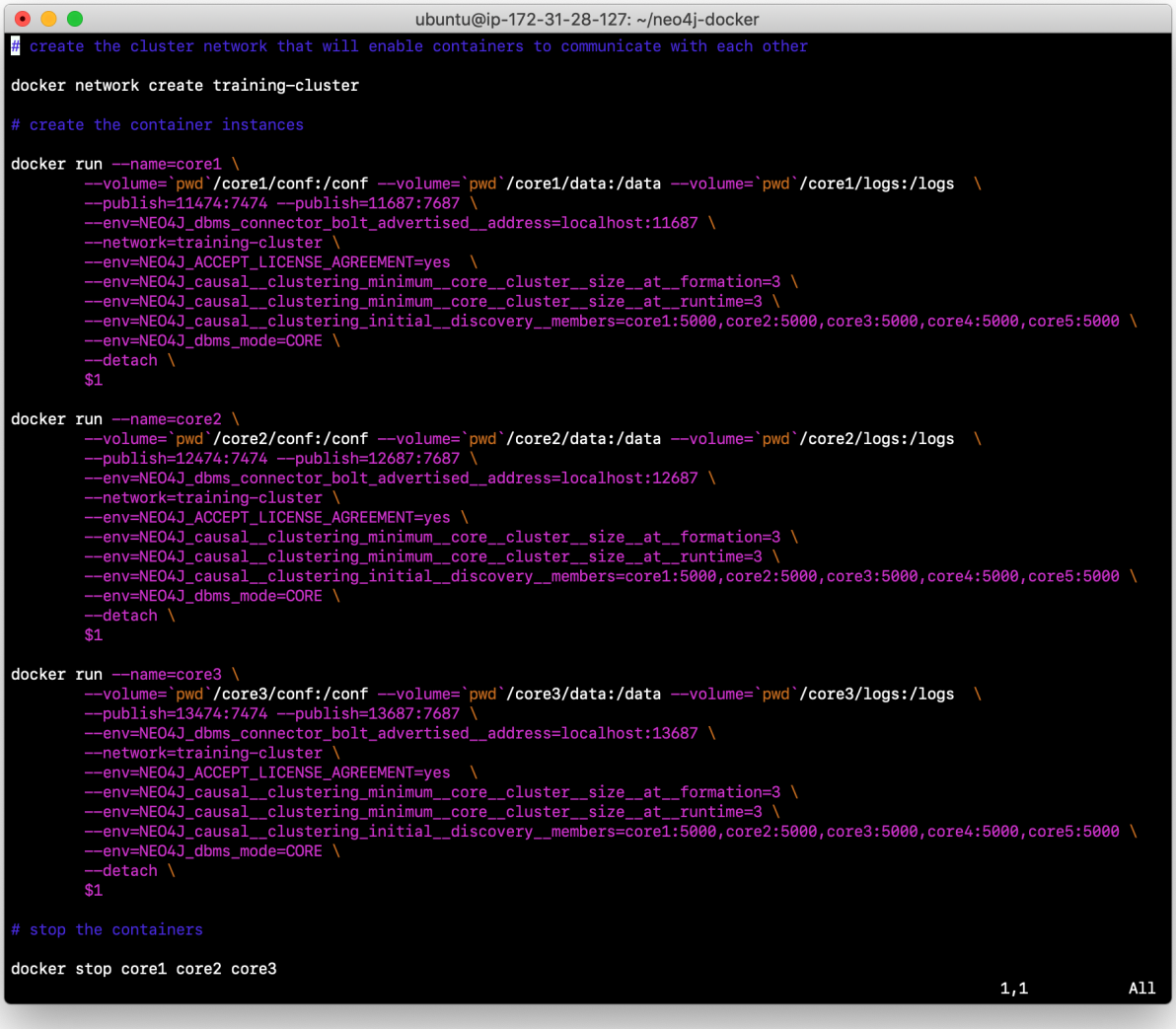
Exercise steps:

1. Open a terminal on your system.
2. Confirm that you have the Neo4j 3.5.0 Docker image: `docker images`. Note that you will have a different Image ID.

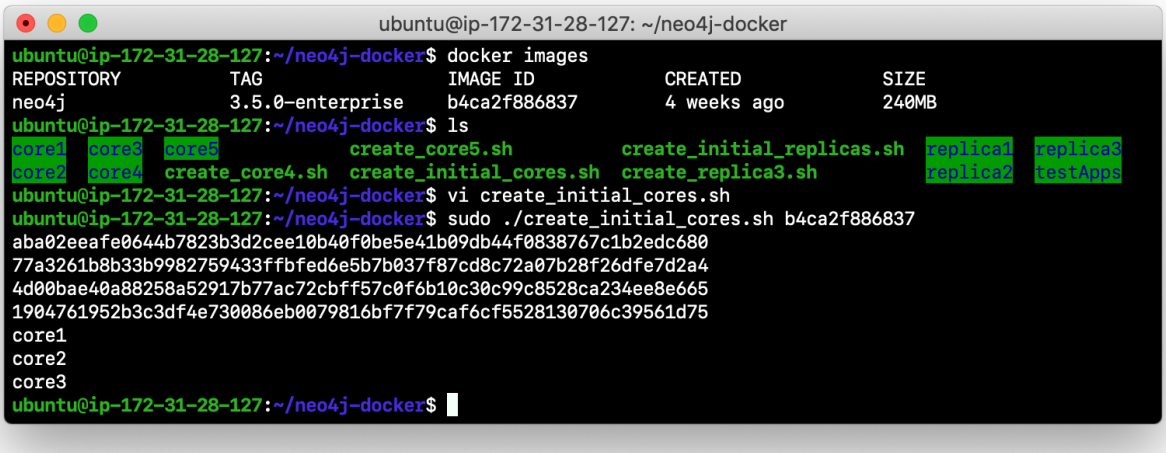


3. Navigate to the **neo4j-docker** folder. This is the folder that will contain all configuration changes for the Neo4j instances you will be running in the cluster. Initially, you will be working with three core servers. Here you can see that you have a folder for each core server and each read replica server.

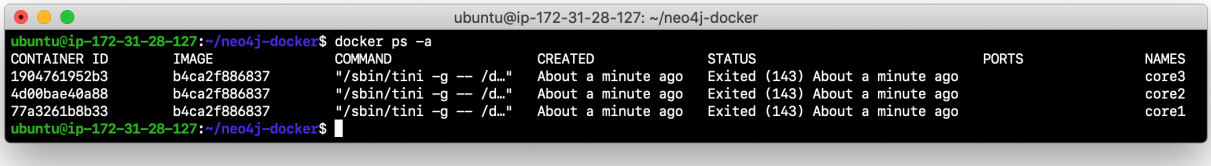
4. Examine the **create_initial_cores.sh** file. This script creates the network that will be used in your Docker environment and then creates three Docker container instances from the Neo4j image. Each instance will represent a core server. Finally, the script stops the three instances.



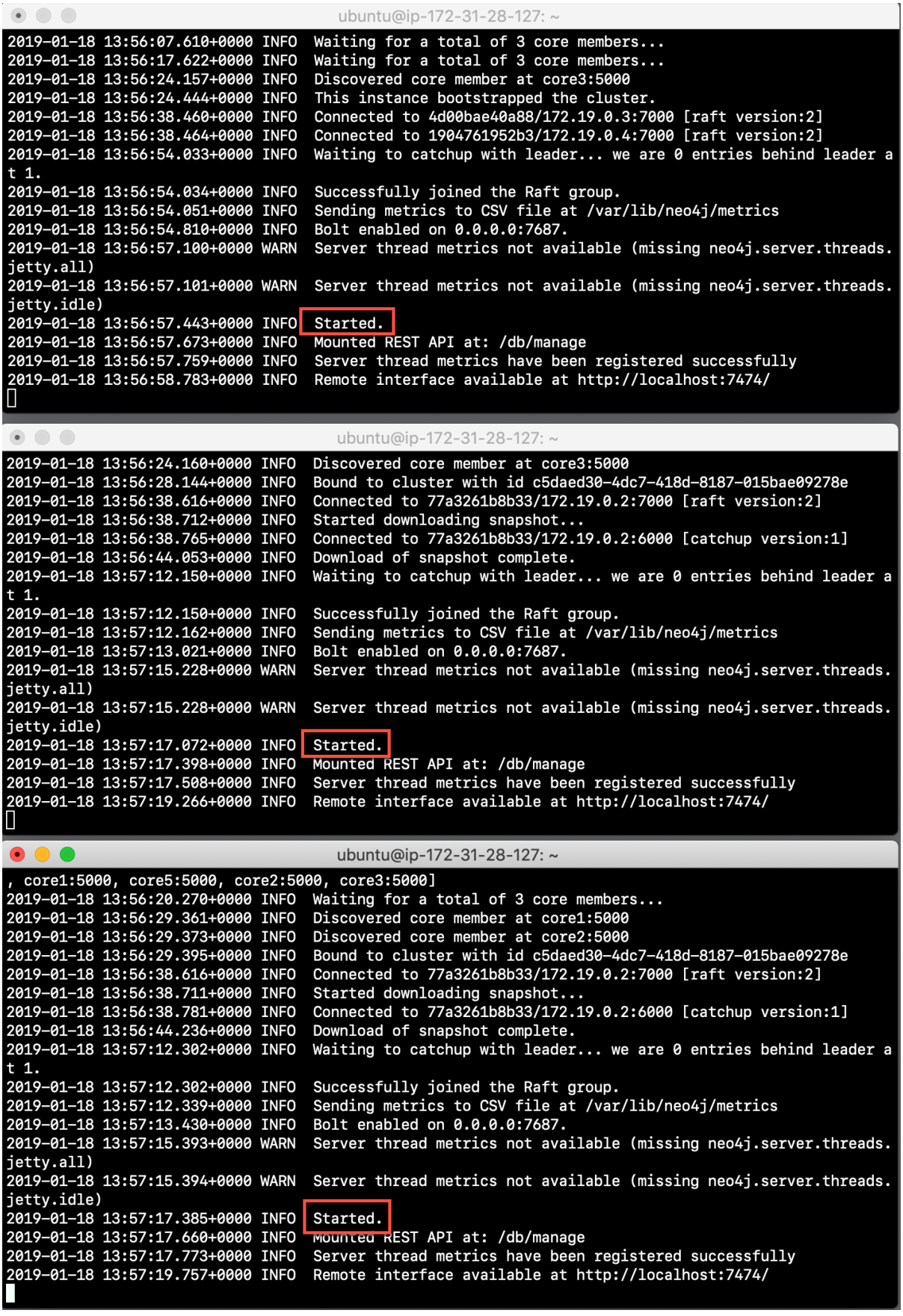
5. Run **create_initial_cores.sh** as root `sudo ./create_initial_cores.sh <Image ID>` providing as an argument the Image ID of the Neo4j Docker image.



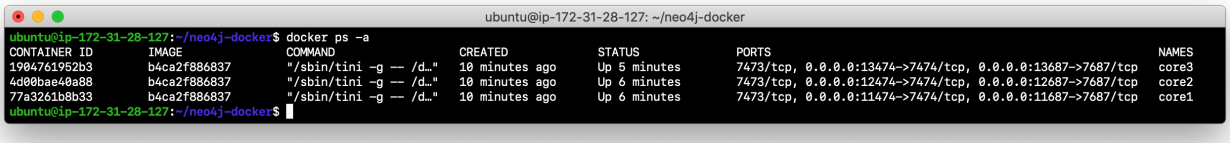
6. Confirm that the three containers exist: `docker ps -a`



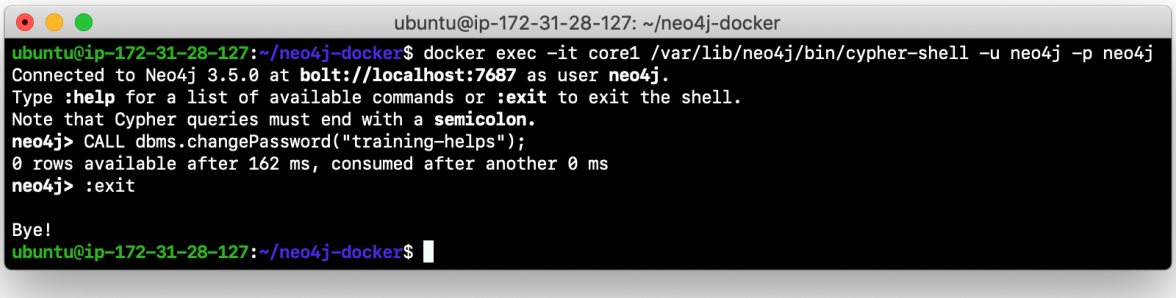
7. Open a terminal window for each of the core servers. (three of them).
8. In each core server window, start the instance: `docker start -a coreX`. The instance should be started. These instances are set up so that the default browser port on localhost will be 11474, 12474, and 13474 on each instance respectively. Notice that each instance uses it's own database as the active database. For example, here is the result of starting the core server containers. Notice that each server starts as part of the cluster. The servers are not fully started until all catchup has been done between the servers and the *Started* record is shown. The databases will not be accessible by clients until *all* core members of the cluster have successfully started.


ubuntu@ip-172-31-28-127: ~
2019-01-18 13:56:07.610+0000 INFO Waiting for a total of 3 core members...
2019-01-18 13:56:17.622+0000 INFO Waiting for a total of 3 core members...
2019-01-18 13:56:24.157+0000 INFO Discovered core member at core3:5000
2019-01-18 13:56:24.444+0000 INFO This instance bootstrapped the cluster.
2019-01-18 13:56:38.460+0000 INFO Connected to 4d00bae40a88/172.19.0.3:7000 [raft version:2]
2019-01-18 13:56:38.464+0000 INFO Connected to 1904761952b3/172.19.0.4:7000 [raft version:2]
2019-01-18 13:56:54.033+0000 INFO Waiting to catchup with leader... we are 0 entries behind leader a
t 1.
2019-01-18 13:56:54.034+0000 INFO Successfully joined the Raft group.
2019-01-18 13:56:54.051+0000 INFO Sending metrics to CSV file at /var/lib/neo4j/metrics
2019-01-18 13:56:54.810+0000 INFO Bolt enabled on 0.0.0.0:7687.
2019-01-18 13:56:57.100+0000 WARN Server thread metrics not available (missing neo4j.server.threads.
jetty.all)
2019-01-18 13:56:57.101+0000 WARN Server thread metrics not available (missing neo4j.server.threads.
jetty.idle)
2019-01-18 13:56:57.443+0000 INFO Started.
2019-01-18 13:56:57.673+0000 INFO Mounted REST API at: /db/manage
2019-01-18 13:56:57.759+0000 INFO Server thread metrics have been registered successfully
2019-01-18 13:56:58.783+0000 INFO Remote interface available at http://localhost:7474/
ubuntu@ip-172-31-28-127: ~
2019-01-18 13:56:24.160+0000 INFO Discovered core member at core3:5000
2019-01-18 13:56:28.144+0000 INFO Bound to cluster with id c5daed30-4dc7-418d-8187-015bae09278e
2019-01-18 13:56:38.616+0000 INFO Connected to 77a3261b8b33/172.19.0.2:7000 [raft version:2]
2019-01-18 13:56:38.712+0000 INFO Started downloading snapshot...
2019-01-18 13:56:38.765+0000 INFO Connected to 77a3261b8b33/172.19.0.2:6000 [catchup version:1]
2019-01-18 13:56:44.053+0000 INFO Download of snapshot complete.
2019-01-18 13:57:12.150+0000 INFO Waiting to catchup with leader... we are 0 entries behind leader a
t 1.
2019-01-18 13:57:12.150+0000 INFO Successfully joined the Raft group.
2019-01-18 13:57:12.162+0000 INFO Sending metrics to CSV file at /var/lib/neo4j/metrics
2019-01-18 13:57:13.021+0000 INFO Bolt enabled on 0.0.0.0:7687.
2019-01-18 13:57:15.228+0000 WARN Server thread metrics not available (missing neo4j.server.threads.
jetty.all)
2019-01-18 13:57:15.228+0000 WARN Server thread metrics not available (missing neo4j.server.threads.
jetty.idle)
2019-01-18 13:57:17.072+0000 INFO Started.
2019-01-18 13:57:17.398+0000 INFO Mounted REST API at: /db/manage
2019-01-18 13:57:17.508+0000 INFO Server thread metrics have been registered successfully
2019-01-18 13:57:19.266+0000 INFO Remote interface available at http://localhost:7474/
ubuntu@ip-172-31-28-127: ~
, core1:5000, core5:5000, core2:5000, core3:5000]
2019-01-18 13:56:20.270+0000 INFO Waiting for a total of 3 core members...
2019-01-18 13:56:29.361+0000 INFO Discovered core member at core1:5000
2019-01-18 13:56:29.373+0000 INFO Discovered core member at core2:5000
2019-01-18 13:56:29.395+0000 INFO Bound to cluster with id c5daed30-4dc7-418d-8187-015bae09278e
2019-01-18 13:56:38.616+0000 INFO Connected to 77a3261b8b33/172.19.0.2:7000 [raft version:2]
2019-01-18 13:56:38.711+0000 INFO Started downloading snapshot...
2019-01-18 13:56:38.781+0000 INFO Connected to 77a3261b8b33/172.19.0.2:6000 [catchup version:1]
2019-01-18 13:56:44.236+0000 INFO Download of snapshot complete.
2019-01-18 13:57:12.302+0000 INFO Waiting to catchup with leader... we are 0 entries behind leader a
t 1.
2019-01-18 13:57:12.302+0000 INFO Successfully joined the Raft group.
2019-01-18 13:57:12.339+0000 INFO Sending metrics to CSV file at /var/lib/neo4j/metrics
2019-01-18 13:57:13.430+0000 INFO Bolt enabled on 0.0.0.0:7687.
2019-01-18 13:57:15.393+0000 WARN Server thread metrics not available (missing neo4j.server.threads.
jetty.all)
2019-01-18 13:57:15.394+0000 WARN Server thread metrics not available (missing neo4j.server.threads.
jetty.idle)
2019-01-18 13:57:17.385+0000 INFO Started.
2019-01-18 13:57:17.660+0000 INFO Mounted REST API at: /db/manage
2019-01-18 13:57:17.773+0000 INFO Server thread metrics have been registered successfully
2019-01-18 13:57:19.757+0000 INFO Remote interface available at http://localhost:7474/

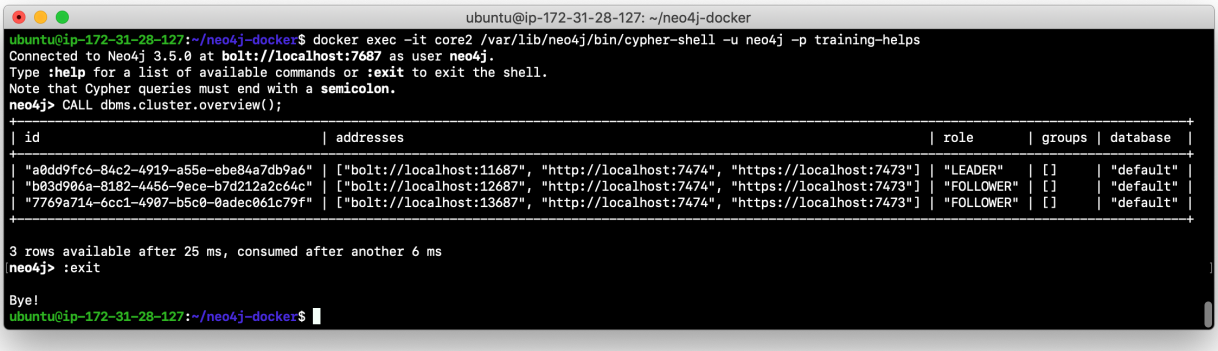
9. In your non-core server terminal window, confirm that all core servers are running in the network by typing `docker ps -a`.



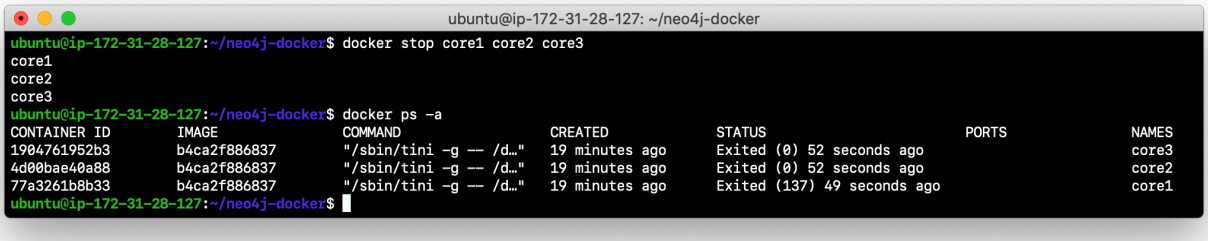
10. In your non-core server terminal window, log in to the core1 server with `cypher-shell` as follows `docker exec -it core1 /var/lib/neo4j/bin/cypher-shell -u neo4j -p neo4j`
11. Change the password. Here is an example where we change the password for core1:



12. Repeat the previous two steps for core2 and core3 to change the password for the *neo4j* user.
13. Log in to any of the servers and get the cluster overview information in `cypher-shell`. In this image, *core1* is the *LEADER*:



14. Shut down all core servers by typing this in a non-core server terminal window: `docker stop core1 core2 core3`



15. You can now close the terminal windows you used for each of the core servers, but keep the non-core server window open.

You have now successfully configured, started, and accessed core servers (as Docker containers) running in a causal cluster.

Seeding the data for the cluster

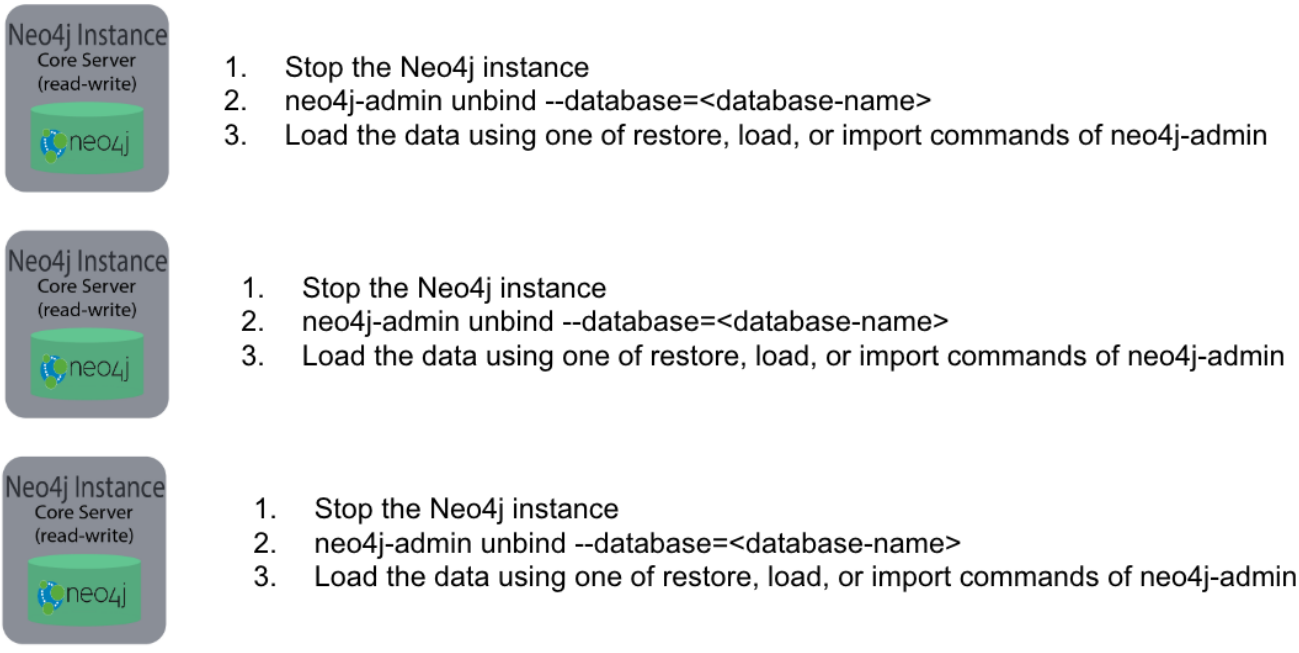
When setting up a cluster for your application, you must ensure that the database that will be used in the cluster has been populated with your application data. In a cluster, each Neo4j instance has its own database, but the data in the databases for each core server that is actively running in the cluster is identical.

Before you seed the data for each core server that is part of a cluster, you must unbind it from the cluster. To unbind the core server, the instance must be stopped, then you run `neo4j-admin unbind --database=<database-name>`.

Loading the data

When you seed the data for the cluster, you can do any of the following, but you must do the same on each of the core servers of the cluster to create the production database. Note that the core servers must be down for these tasks. You learned how to do these tasks in the previous module.

- Restore data using an online backup.
- Load data using an offline backup.
- Create data using the import tool and a set of **.csv** files.



If the amount of application data is relatively small (less than 10M nodes) you can also load **.csv** data into a running core server in the cluster where all core servers are started and actively part of the cluster. This will propagate the data to all databases in the cluster.

Exercise #2: Seeding the cluster databases

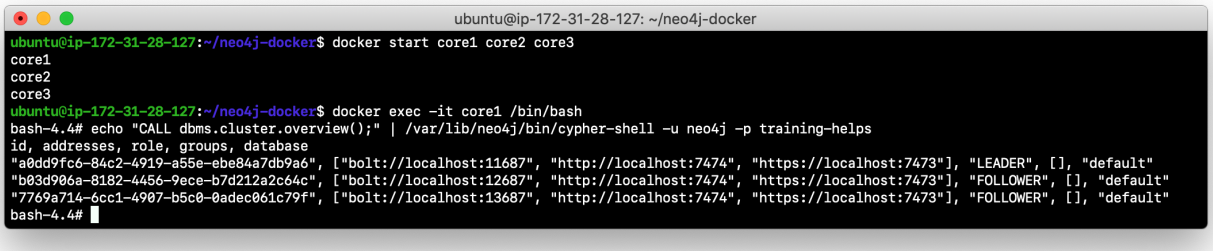
In this Exercise, you will populate the databases in the cluster that you created earlier. Because you are using Docker containers for learning about clustering, you cannot perform the normal seeding procedures as you would in your real production environment because when using the Neo4j Docker containers, the Neo4j instance is already started when you start the container. Instead, you will simply start the core servers in the cluster and connect to one of them. Then you will use `cypher-shell` to load the *Movie* data into the database and the data will be propagated to the other core servers.

Before you begin

Ensure that you have performed the steps in Exercise 1 where you set up the core servers as Docker containers. Note that you can perform the steps of this Exercise in a single terminal window.

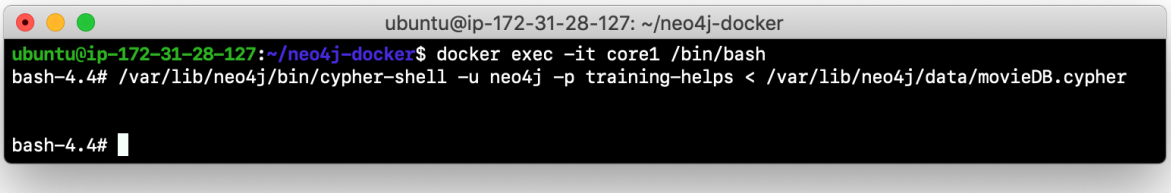
Exercise steps:

1. In a terminal window, start the core servers: `docker start core1 core2 core3`. This will start the core servers in background mode where the log is not attached to STDOUT. If you want to see what is happening with a particular core server, you can always view the messages in **<coreX>/logs/debug.log**.
2. By default, all writes must be performed by the *LEADER* of the cluster. Determine which core server is the *LEADER*. **Hint:** You can do this by logging in to any core server that is running (`docker exec -it <core server> /bin/bash`) and entering the following command: `echo "CALL dbms.cluster.overview();" | /var/lib/neo4j/bin/cypher-shell -u neo4j -p training-helps`. In this example, core1 is the *LEADER*:

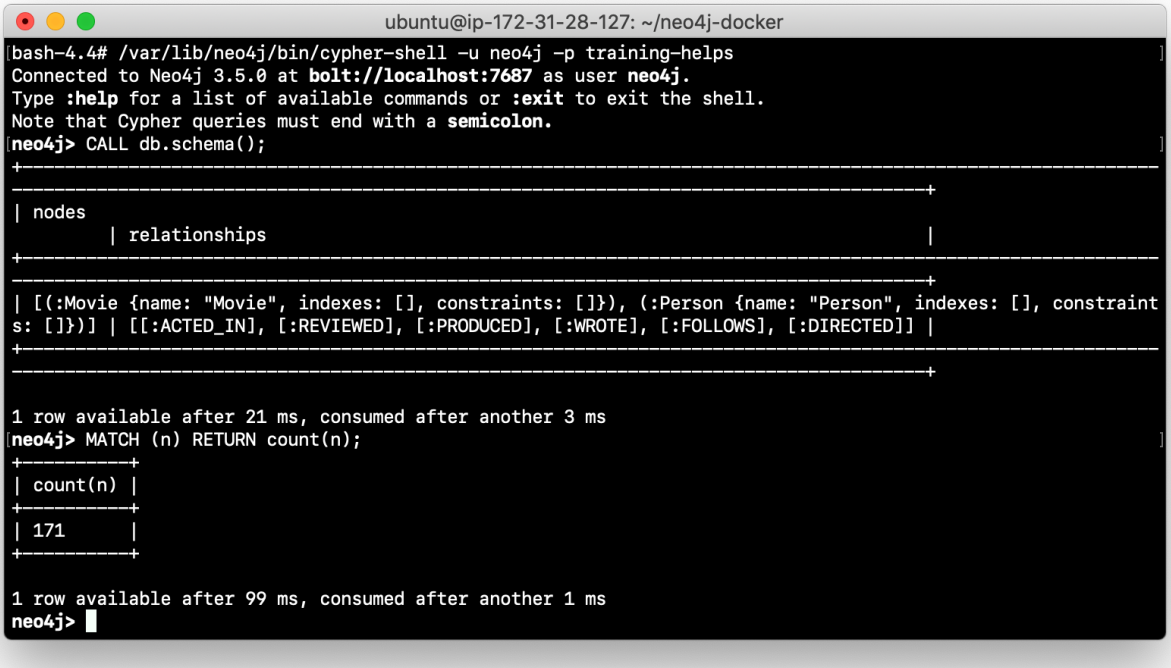


3. Log in to the core server that is the *LEADER*.

4. Run `cypher-shell` specifying that the **movie.cypher** statements will be run. **Hint:** You can do this with a single command line: `/var/lib/neo4j/bin/cypher-shell -u neo4j -p training-helps < /var/lib/neo4j/data/movieDB.cypher`

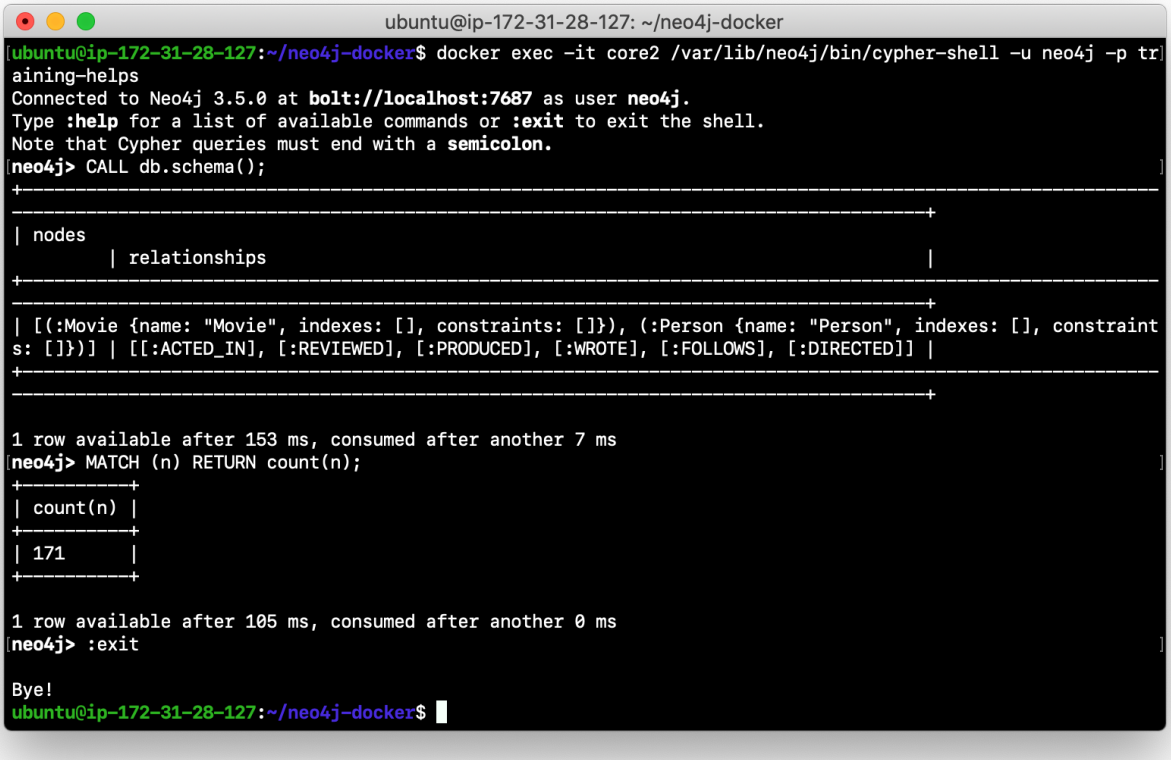


5. Log in to `cypher-shell` and confirm that the data has been loaded into the database.



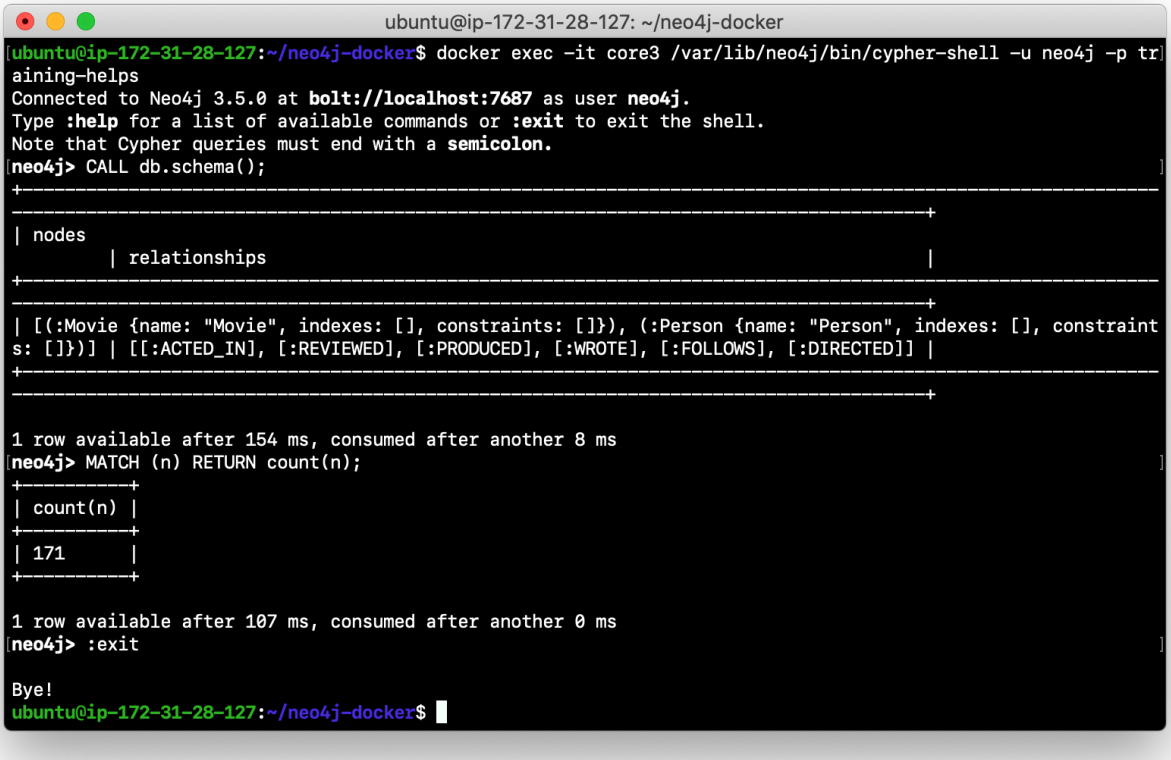
6. Log out of the core server.
7. Log in to a *FOLLOWER* core server with `cypher-shell`. **Hint:** For example, you can log in to core2 with `cypher-shell` with the following command: `docker exec -it core2 /var/lib/neo4j/bin/cypher-shell -u neo4j -p training-helps`

8. Verify that the *Movie* data is in the database for this core server.



9. Log out of the core server.
10. Log in to the remaining core server that is the *FOLLOWER* with `cypher-shell`.

11. Verify that the *Movie* data is in the database for this core server.

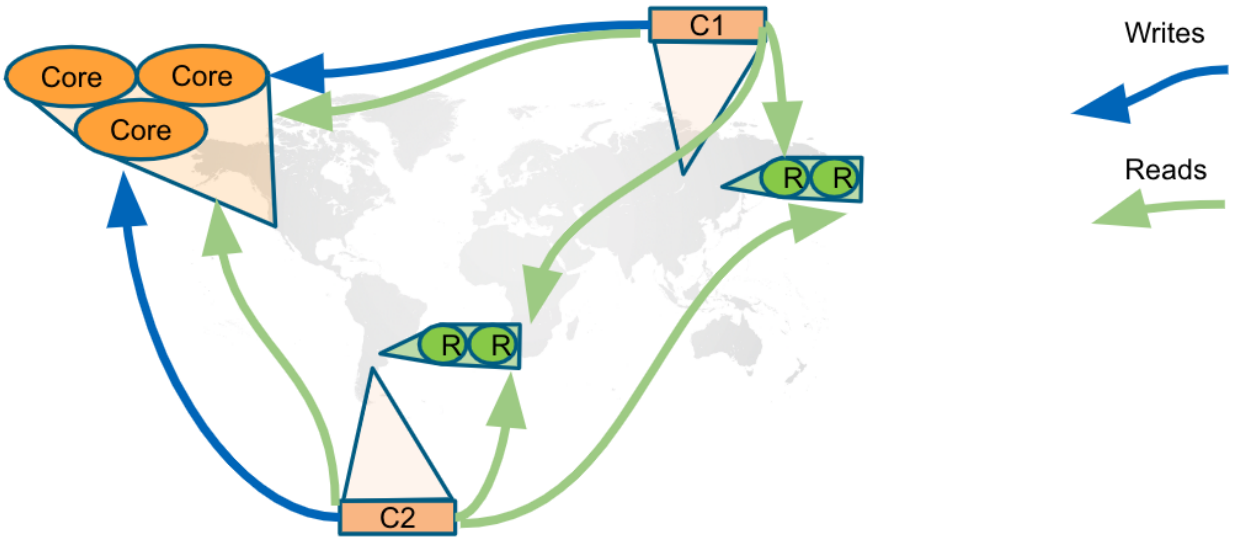


12. Log out of the core server.

You have now seen the cluster in action. Any modification to one database in the core server cluster is propagated to the other core servers.

Basic routing in a cluster

In a cluster, all write operations must be coordinated by the *LEADER* in the cluster. Which core server is designated as the *LEADER* could change at any time in the event of a failure or a network slowdown. Applications that access the database can automatically route their write operations to whatever *LEADER* is available as this functionality is built into the Neo4j driver libraries. The Neo4j driver code obtains the routing table and automatically updates it as necessary if the endpoints in the cluster change. To implement the automatic routing, application clients that will be updating the database must use the *bolt+routing* protocol when they connect to any of the core servers in the cluster.



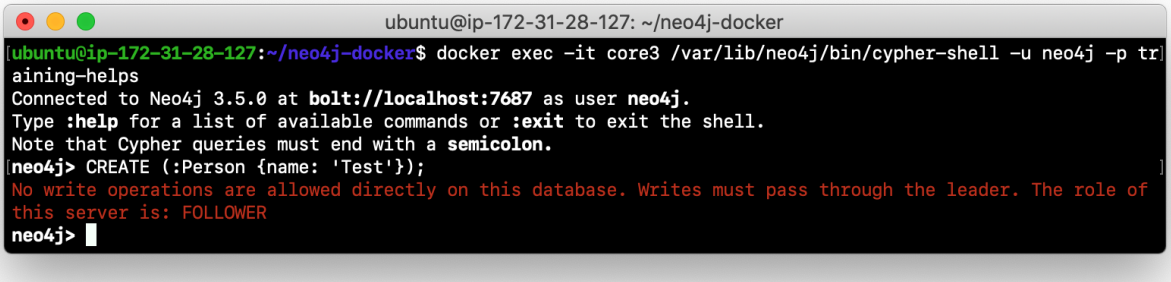
bolt+routing

Applications that update the database should always use *bolt+routing* when accessing the core servers in a cluster. Using this protocol, applications gain:

- Automatic routing to an available server.
- Load balancing of requests between the available servers.
- Automatic retries.
- Causal chaining (bookmarks).

bolt and bolt+routing at runtime

For example, if you have a cluster with three core servers and *core1* is the *LEADER*, your application can only write to *core1* using the *bolt* protocol and bolt port for *core1*. An easy way to see this restriction is if you use the default address for `cypher-shell` on the system where a *FOLLOWER* is running. If you connect via `cypher-shell` to the server on *core2* and attempt to update the database, you receive an error:



When using clustering, all application code that updates the application should use the *bolt+routing* protocol which will enable applications to be able to write to the database, even in the event of a failure of one of the core servers. Applications should be written with the understanding that transactions are automatically retried.

Exercise #3: Accessing the core servers in a cluster

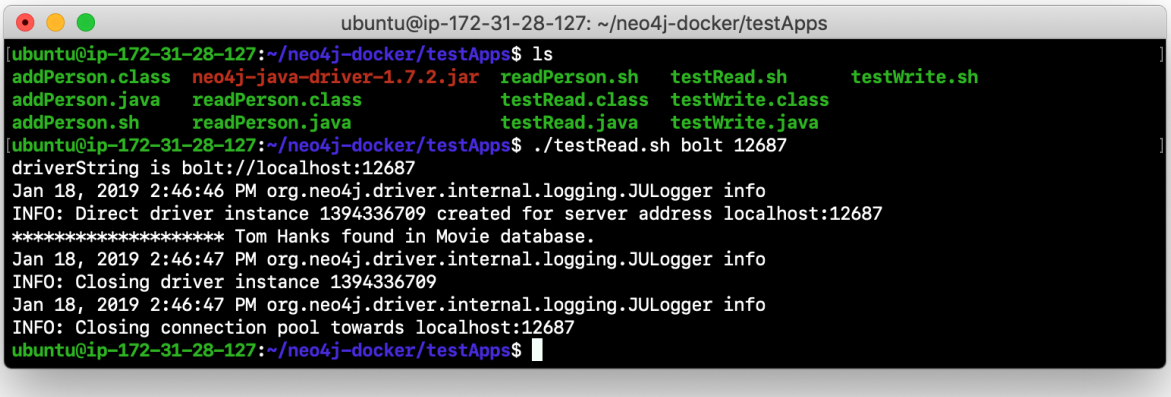
In this Exercise, you gain some experience with *bolt+routing* by running two stand-alone Java applications: one that reads from the database and one that writes to the database.

Before you begin

1. Ensure that you have performed the steps in Exercise 2 where you have populated the database used for the cluster and all three core servers are running. Note that you can perform the steps of this Exercise in a single terminal window.
2. Ensure that the three core servers are started.
3. Log out of the core server if you have not done so already. You should be in a terminal window where you manage Docker.

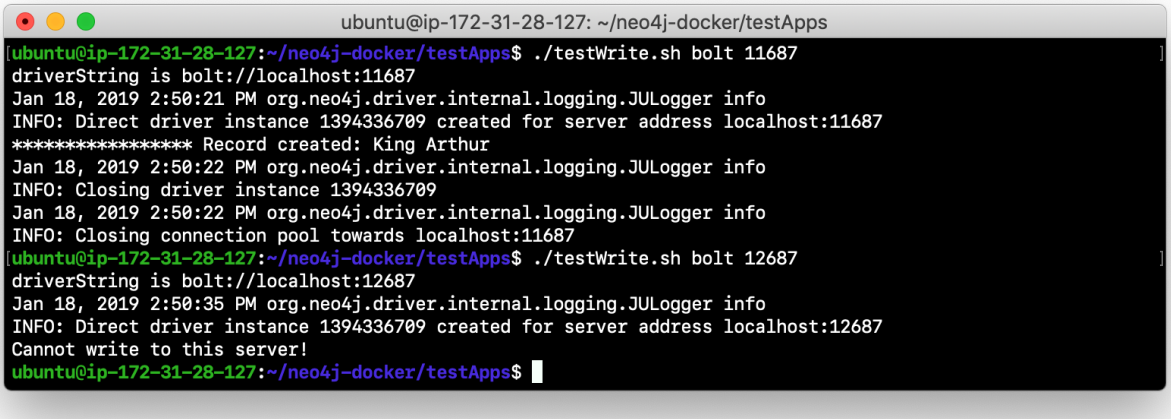
Exercise steps:

1. Navigate to the **neo4j-docker/testApps** folder.
2. There are three Java applications as well as scripts for running them. These scripts enable you to run a read-only client or write client against the database where you specify the protocol and the port for connecting to the Neo4j instance. Unless you modified port numbers in the **create_initial_cores.sh** script when you created the containers, the bolt ports used for core1, core2, and core3 are 11687, 12687, and 13687 respectively. What this means is that clients can read from the database using these ports using the *bolt* protocol. Try running **testRead.sh**, providing bolt as the protocol and one of the above port numbers. For example, type `./testRead.sh bolt 12687`. You should be able to successfully read from each server. Here is an example of running the script against the core2 server which currently is a *FOLLOWER* in the cluster:

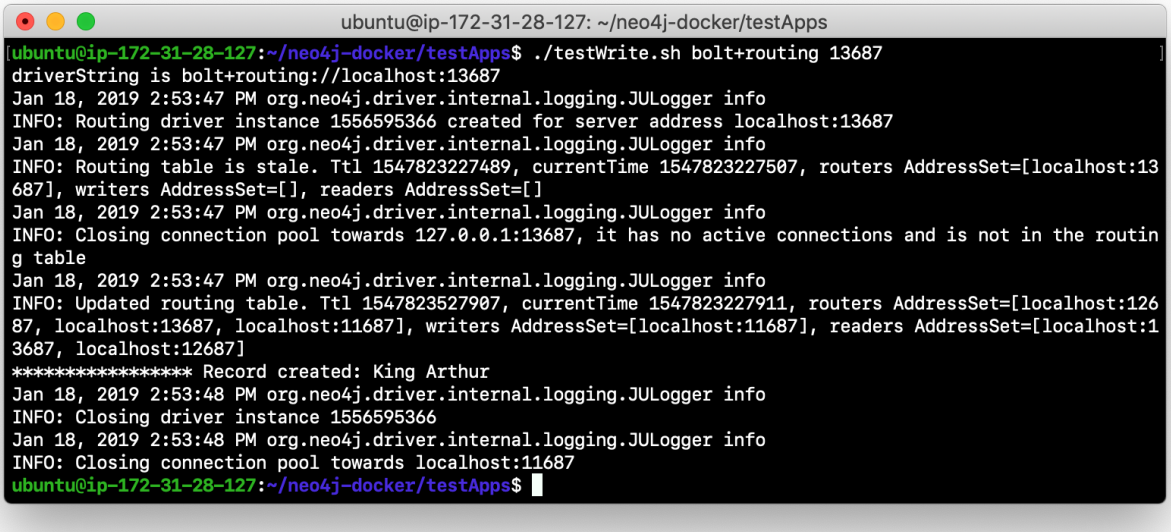


3. Next, run the script against the other servers in the network. All reads should be successful.

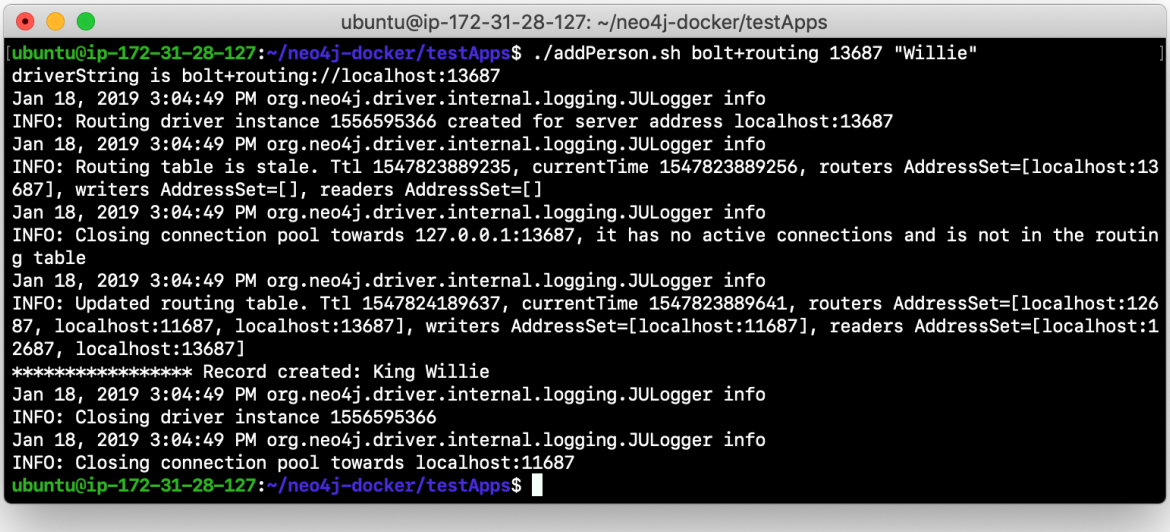
4. Next, run the **testWrite.sh** script against the same port using the *bolt* protocol. For example, type `./testWrite.sh bolt 11687`. What you should see is that you can only use the *bolt* protocol for writing against the *LEADER*.



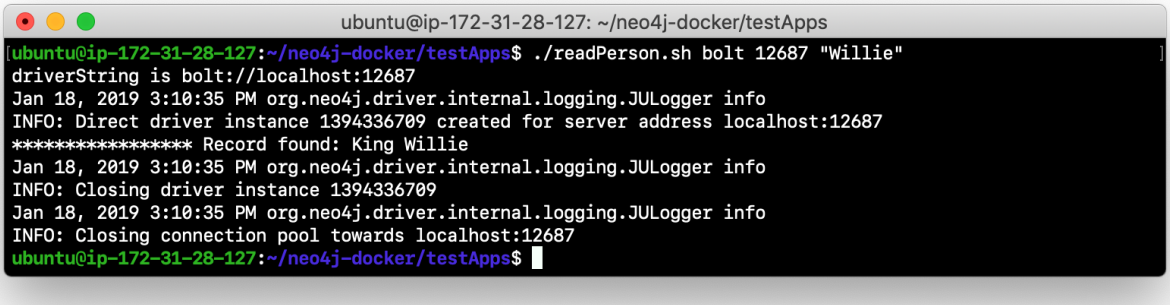
5. Next, change the protocol from *bolt* to *bolt+routing* and write to the core servers that are *FOLLOWER* servers. For example, type `./testWrite.sh bolt+routing 12687`. With this protocol, all writes are routed to the *LEADER* and the application can write to the database.



6. Next, you will add data to the database with a client that sends the request to a *FOLLOWER* core server. Run the **addPerson.sh** script against any port representing a *FOLLOWER* using the *bolt* protocol. For example, type `./addPerson.sh bolt+routing 13687 "Willie"`. This will add a *Person* node to the database for core3.



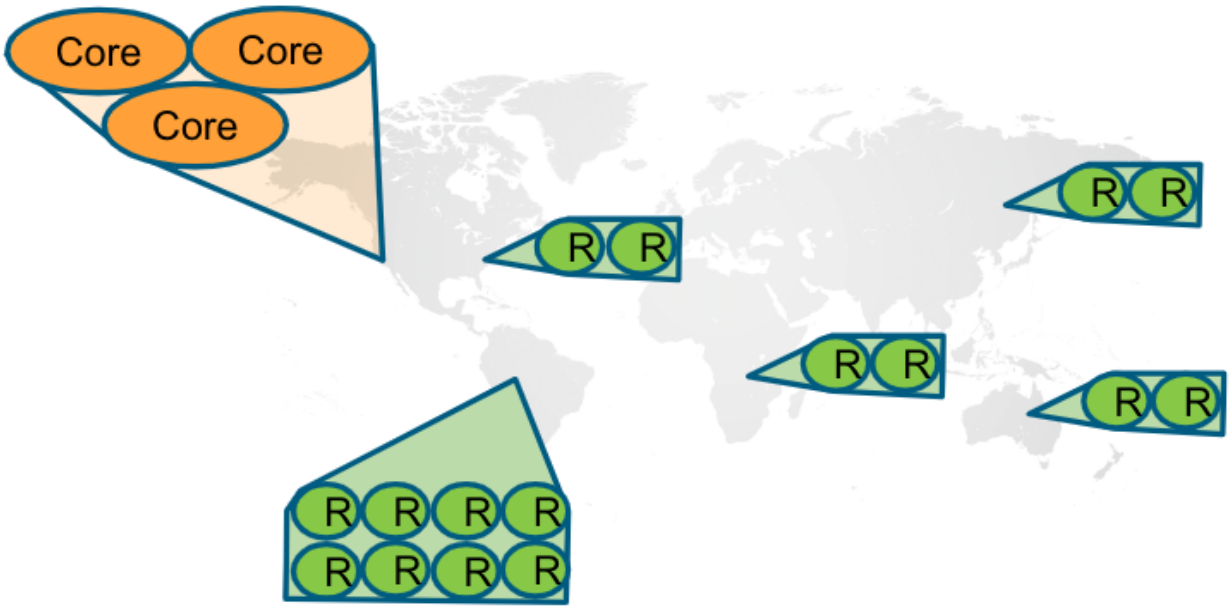
7. Verify that this newly-added *Person* node is written to the other servers in the cluster by using the *bolt* protocol to request specific servers. For example, type `./readPerson.sh bolt 12687 "Willie"` to confirm that the data was added to core2.



You have now seen how updates to the core servers in a cluster must be coordinated by the server that is currently the *LEADER* and how reads and writes are performed in a cluster using the *bolt* and *bolt+routing* protocols.

Configuring read replica servers

You configure read replica servers on host systems where you want the data to be distributed. Read replica servers know about the cluster, but whether they are running or not has no effect on the health of the cluster. In a production environment, you can add many read replicas to the cluster. They will have no impact on the performance of the cluster.



Configuration settings for read replica servers

Here are the configuration settings you use for a read replica server:

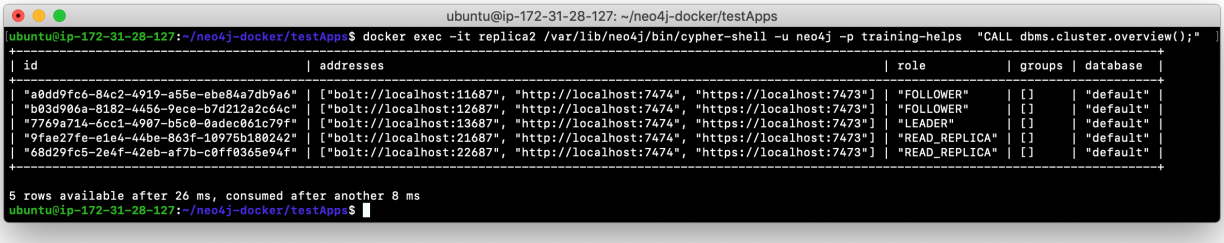
```
dbms.connectors.default_listen_address=0.0.0.0

dbms.connector.https.listen_address=0.0.0.0:7473
dbms.connector.http.listen_address=0.0.0.0:7474
dbms.connector.bolt.listen_address=0.0.0.0:7687

dbms.connector.bolt.advertised_address=localhost:18687

causal_clustering.initial_discovery_members=XXX1:5000,XXX2:5000,XXX3:5000,XXX4:5000,XX
X5:5000

dbms.mode=READ_REPLICA
```

Just like the configuration for a core server, you must specify the bolt advertised address, as well as the addresses for the servers that are the members of the cluster. However, you can add as many read replica servers and they will not impact the functioning of the cluster.

Read replica server startup

There can be many read replica servers in a cluster. When they start, they register with a core server that maintains a shared whiteboard (cache) that can be used by multiple read replica servers. As part of the startup process, the read replica catches up to the core server. The read replicas do not use the *Raft* protocol. Instead, they poll the core servers to obtain the updates to the database that they must apply locally.

Here is what you would see if you had a cluster with three core servers and two read replica servers running:



Unlike core servers where applications use *bolt+routing* to access the database, clients of read replica servers use *bolt*.

Read replica server shutdown

Since the read replica servers are considered "transient", when they shut down, there is no effect to the operation of the cluster. Of course, detection of a shutdown when it is related to a hardware or network failure must be detected so that a new read replica server can be started as clients depend on read access can continue their work.

Exercise #4: Accessing the read replica servers in a cluster

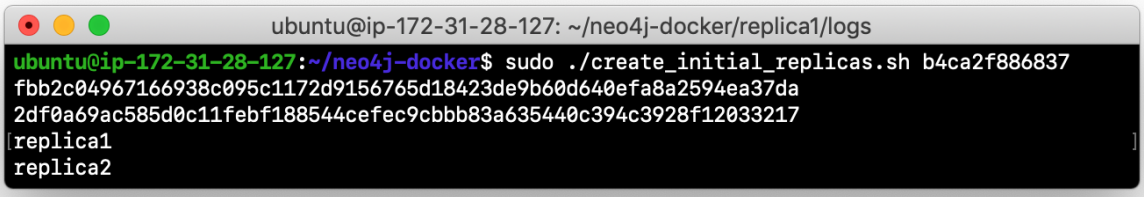
In this Exercise, you will see how read replica servers can be used to retrieve changed data from the core servers.

Before you begin

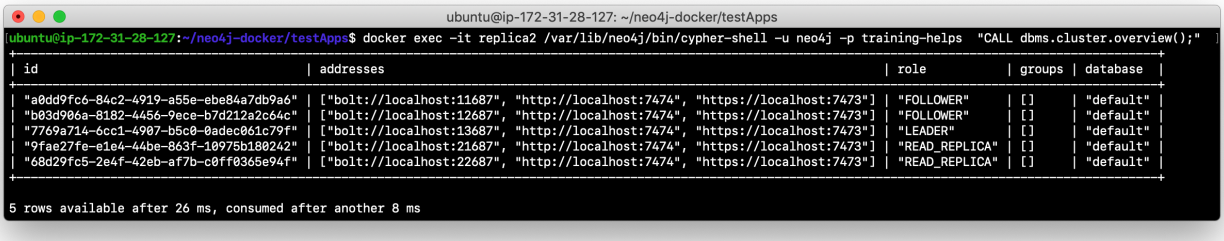
1. Ensure that the three core servers are started.
2. Open a terminal window where you will be managing Docker containers.

Exercise steps:

1. Navigate to the **neo4j-docker** folder.
2. Run the script to create the initial replica servers, providing the Image ID of the Neo4j Docker image.

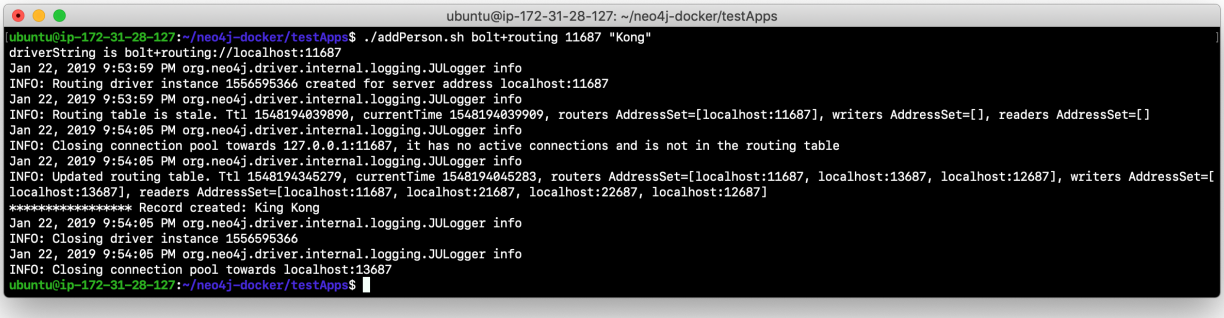


3. Start replica1 and replica2: `docker start replica1 replica2`.
4. Log in to each of read replica servers and change the password.
5. Use Cypher to retrieve the cluster overview. For example in a terminal window type: `docker exec -it replica2 /var/lib/neo4j/bin/cypher-shell -u neo4j -p training-helps "CALL dbms.cluster.overview();"`. Do you see all three core servers and the two read replica servers?

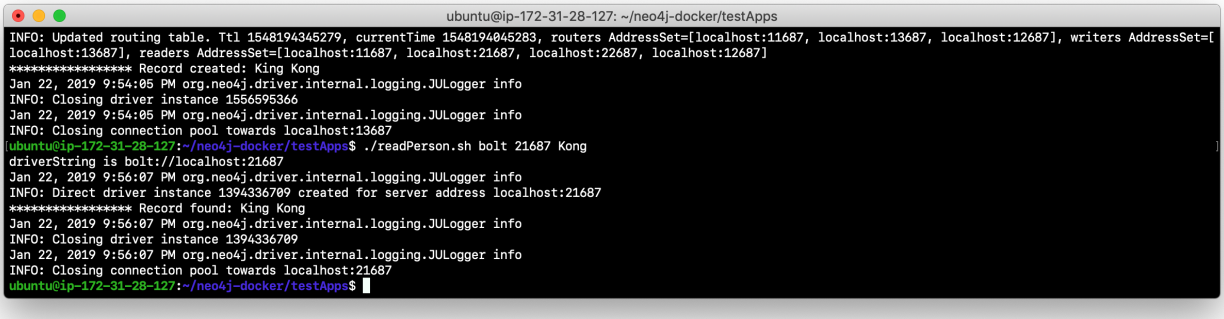


6. Navigate to the **neo4j-docker/testApps** folder.

7. Run the **addPerson.sh** script against any port for a core server using the *bolt+routing* protocol. For example, type `./addPerson.sh bolt+routing 13687 "Kong"`. This will add a *Person* node to the database.



8. Verify that this newly-added *Person* node is readable by a read replica server in the cluster by using the *bolt* protocol to request specific servers. For example, type `./readPerson.sh bolt 22687 "Kong"` to confirm that the data is available.



You have now seen how updates to the core servers in a cluster must be coordinated by the server that is currently the *LEADER* and how reads and writes are performed in a cluster using the *bolt* and *bolt+routing* protocols against the core servers and reads are performed in a cluster using the *bolt* protocol against the read replica servers.

Core server lifecycle

The *minimum_core_cluster_size_at_runtime* property specifies the number of servers that will actively participate in the cluster at runtime. The number of core servers that start and join the cluster is used to calculate what the *quorum* is for the cluster. For example, if the number of core servers started is three, then quorum is two. If the number of core servers started is four, then quorum is three. If the number of core servers started is five, then quorum is three. Quorum is important in a cluster as it dictates the behavior of the cluster when core servers are added to or removed from the cluster at runtime.

As an administrator, you must understand which core servers are participating in the cluster and in particular, what the current *quorum* is for the cluster.

How quorum is used

If a core server shuts down, the cluster can still operate provided the number of core servers is equal to or greater than *quorum*. For example, if the current number of core servers is three, *quorum* is two. Provided the cluster has two core servers, it is considered operational for updates. If the cluster maintains quorum, then it is possible to add a different core server to the cluster since a quorum must exist for voting in a new core server.

If the *LEADER* core server shuts down, then one of the other *FOLLOWER* core servers assumes the role of *LEADER*, provided a quorum still exists for the cluster. If a cluster is left with only *FOLLOWER* core servers, this is because *quorum* no longer exists and as a result, the database is read-only. As an administrator, you must ensure that your cluster always has a *LEADER*.

The core servers that are used to start the cluster (membership) are important. Only core servers that originally participated in the cluster can be running in order to add a new core server to the cluster.

Follow this video to understand the life-cycle of a cluster and how *quorum* is used for fault-tolerance for a cluster:

https://youtu.be/0ug7Cpswjio

Recovering a core server

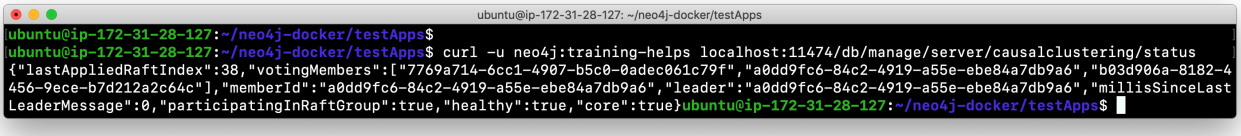
If a core server goes down and you cannot restart it, you have two options:

1. Start a new core server that has not yet been part of the cluster, but is specified in the membership list of the cluster. This will only work if the cluster currently has a quorum so the existing core servers can vote to add the core server to the cluster.
2. Start a new parallel cluster with backup from current read only cluster. This requires that client applications must adjust port numbers they use.

Option (1) is much easier so a best practice is to always specify additional hosts that could be used as replacement core servers in the membership list for a cluster. This will enable you to add core servers to the cluster without needing to stop the cluster.

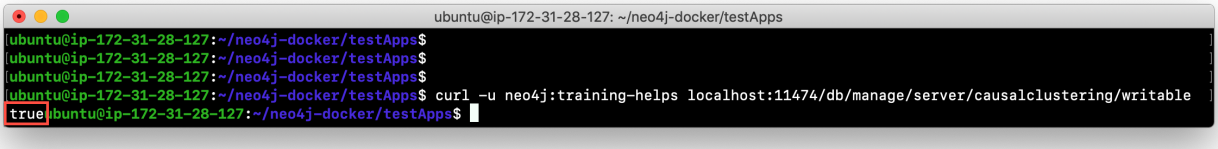
Monitoring core servers

In addition to using Cypher to retrieve the overview state of the cluster, there are also REST APIs for accessing information about a particular server. For example, you can query the status of the cluster as follows: `curl -u neo4j:training-helps localhost:11474/db/manage/server/causalclustering/status` where this query is made against the core1 server:



Is the core server writable?

Or, if you want to see it a particular server is writable (part of a "healthy" cluster). For example, you can get that information as follows: `curl -u neo4j:training-helps localhost:11474/db/manage/server/causalclustering/writable` where this query is made against the core1 server:



Using the REST API enables you as an administrator to script checks against the cluster to ensure that it is running properly and available to the clients.

Helpful configuration settings

The Neo4j Operations Manual documents many properties that are related to clusters. Here are a few you may want to consider for your deployment:

- *causal_clustering.enable_prevoting* set to `TRUE` can reduce the number of *LEADER* switches, especially when a new member is introduced to the cluster.
- *causal_clustering.leader_election_timeout* can be set to a number of seconds (the default is 7s). The default is typically sufficient, but you may need to increase it slightly if your cluster startup is slower than normal.

Exercise #5: Understanding quorum

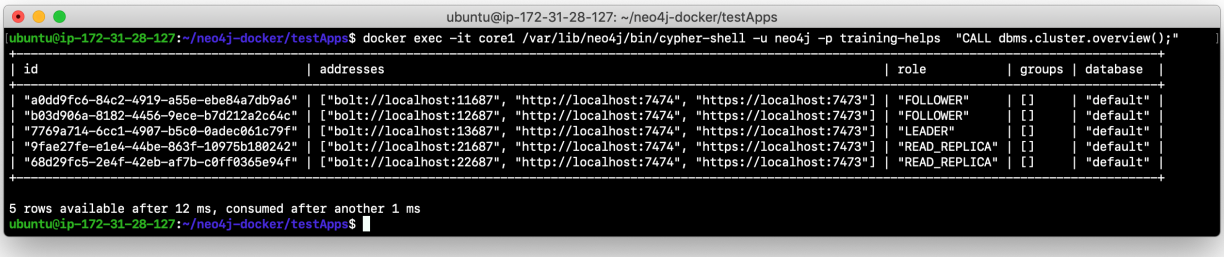
In this Exercise, you gain some experience monitoring the cluster as servers shut down and as servers are added.

Before you begin

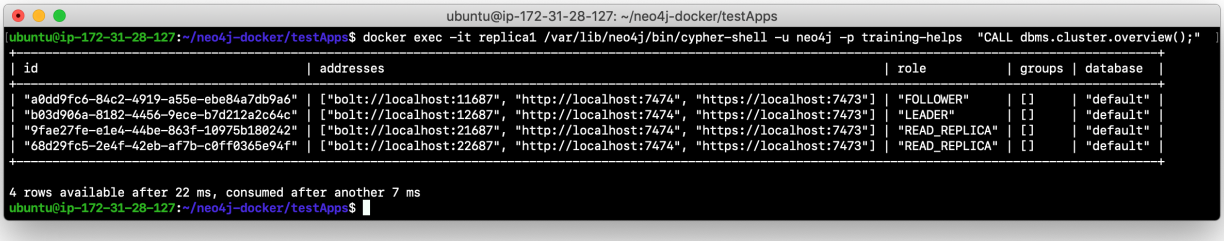
Ensure that you have performed the steps in Exercise 4 and you have a cluster with core1, core2, and core3 started, as well as replica1 and replica2.

Exercise steps:

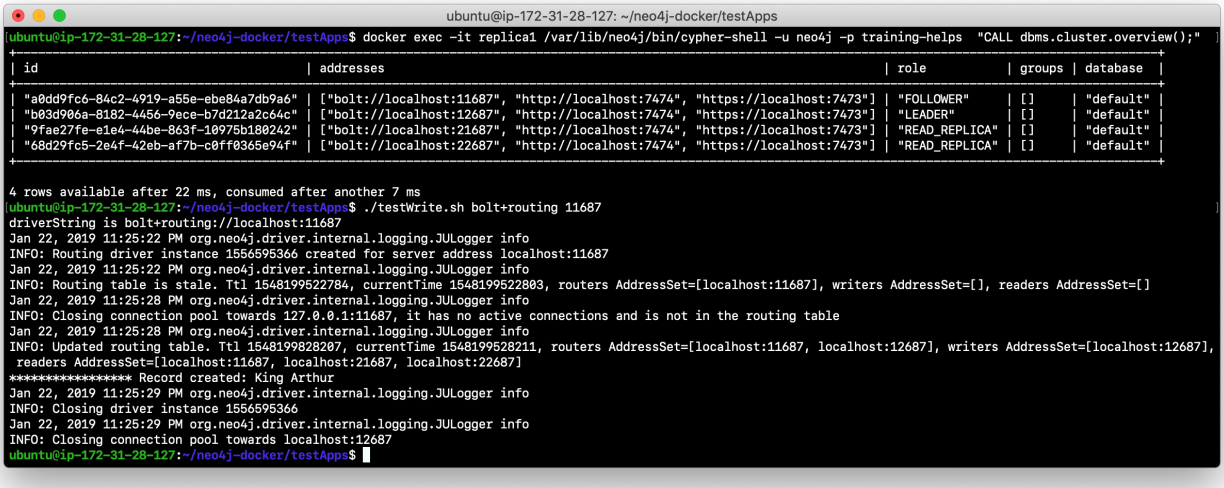
1. View the cluster overview using core1: `docker exec -it core1 /var/lib/neo4j/bin/cypher-shell -u neo4j -p training-helps "CALL dbms.cluster.overview();"`. Make a note of which core server is the *LEADER*. In this example, core3 is the *LEADER*.



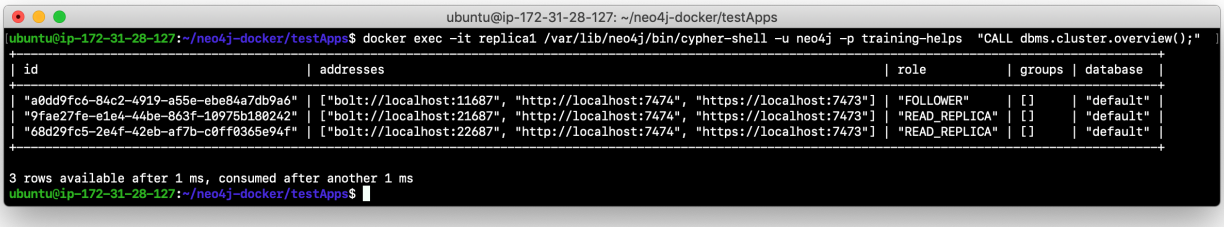
2. Stop the core server that is the *LEADER*.
3. View the cluster overview using replica1: `docker exec -it replica1 /var/lib/neo4j/bin/cypher-shell -u neo4j -p training-helps "CALL dbms.cluster.overview();"`. Do you see that another core server has assumed the *LEADER* role?



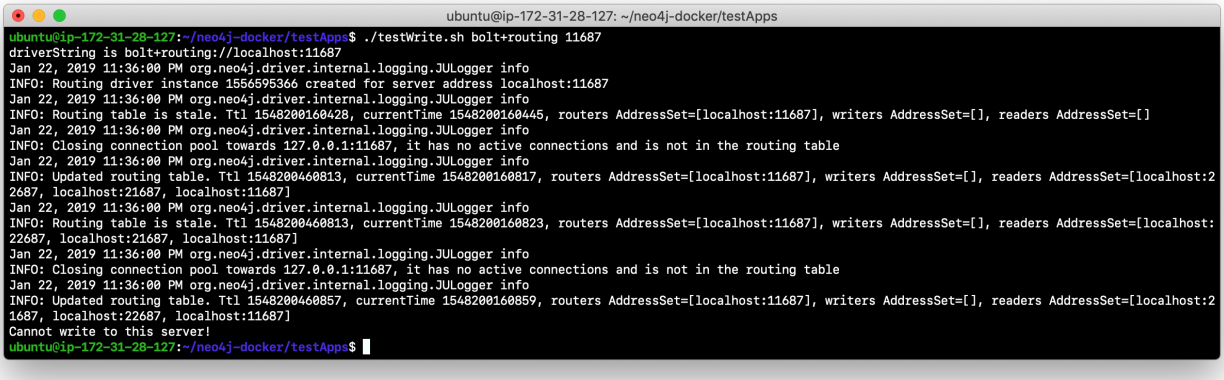
4. In the **testApps** folder, run the script **testWrite.sh** providing the protocol of *bolt+routing* and a port for one of the core servers that is running. Can the client write to the database?



5. Stop the core server that is the *LEADER*.
6. Confirm that the only core server running is now a *FOLLOWER*.

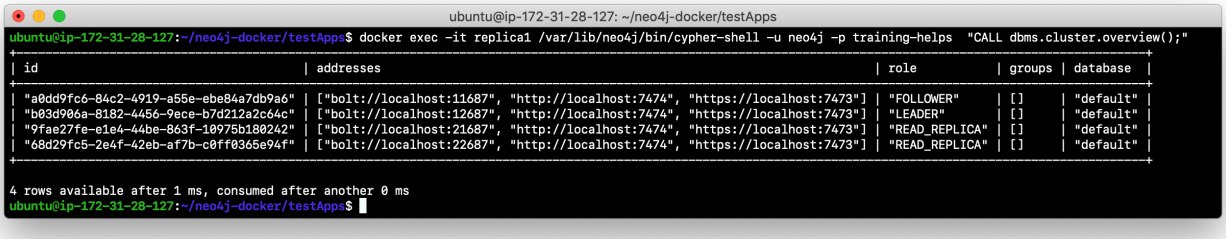


7. Run the script to write to the database using *bolt+routing* and the port number for the remaining core server. Can you write to the database?

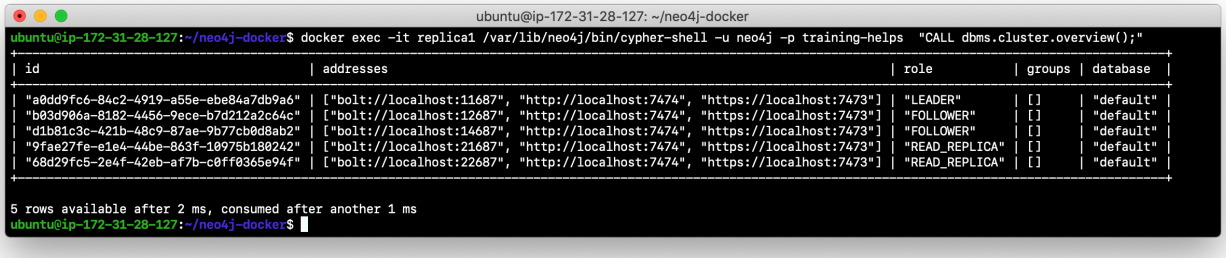


8. Start a core server that you previously stopped.

9. View the cluster overview. Is there now a *LEADER*? This cluster is operational because it now has a *LEADER*



10. The cluster is now back to quorum. What this means is that a new core server can be added (elected) that was not part of the original cluster.
11. Navigate to neo4j-docker and run the script to create core4, providing the Image ID of the Neo4j Docker image.
12. Start the core4 server.
13. Change the password of the core4 server.
14. Retrieve the overview information for the cluster. Does it have two *FOLLOWERS* and one *LEADER*? It was possible to add a new core server to the cluster because the cluster had a quorum and the core5 server was specified in the original configuration of the member list of the cluster.



Clusters in many physical locations

Many large enterprises deploy large datasets that need to be distributed to many physical locations. To deploy a cluster to more than one physical location, a best practice is to host the core servers in one data center, and host the read replicas in another data center. Neo4j also supports hosting core servers in multiple locations. To host Neo4j servers that are geographically distributed, you need a multi-data center license and you must configure it in your **neo4j.conf** file by setting the *multi_dc_license* property to `true`. When doing so, there is more configuration that you must do to ensure that clients are routed to the servers that are physically closest to them. You do this by configuring policy groups for your cluster. If policy groups have been configured for the servers, the application drivers instances must be created to use the policy groups. With policy groups, writes are always routed to the *LEADER*, but reads are routed to any *FOLLOWER* that is available. This enables the cluster and driver to automatically perform load balancing.

Here is a video that shows you why you might want to configure a Causal Cluster to operate in multiple data centers:

https://youtu.be/4q8JklXN7kA

Read more about configuring clusters for multi-data center in the Neo4j Operations Manual

Bookmarks

Both read and write client applications can create bookmarks within a session that enable them to quickly access a location in the database. The bookmarks can be passed between sessions. See the Neo4j Developer Manual for details about writing code that uses bookmarks.

Backing up a cluster

The database for a cluster is backed up online. You must specify `dbms.backup.enabled=true` in the configuration for each core server in the cluster.

The core server can use its transaction port or the backup port for backup operations. You typically use the backup port. Here is the setting that you must add to the configuration:

```
dbms.backup.address=<server-address>:6362
```

Using a read replica for the backup

A best practice is to create and use a read replica server for the backup. In doing so, the read replica server will be in catchup mode with the core servers but can ideally keep up with the committed transactions on the core servers. You can check to see what the last transaction ID is on a core server vs. a read replica by executing the Cypher statement: `CALL dbms.listTransactions() YIELD transactionId;` on each server. Each server will have a different last transaction ID, but as many transactions are performed against the cluster, you should see these values increasing at the same rate. If you find that the read replica is far behind in catching up, you may want to consider using a core server for the backup. If you use a core server for a backup, it could degrade performance of the cluster. If you want to use a core server for backup, you should increase the number of core servers in the cluster, for example from three to five.

Performing the backup

For backing up a cluster, you must first decide which server and port will be used for the backup (backup client). You can backup using either a backup port or a transaction port. In addition, in a real application you will want the backup to be encrypted. For this you must use SSL. Security and encryption is covered later in this training.

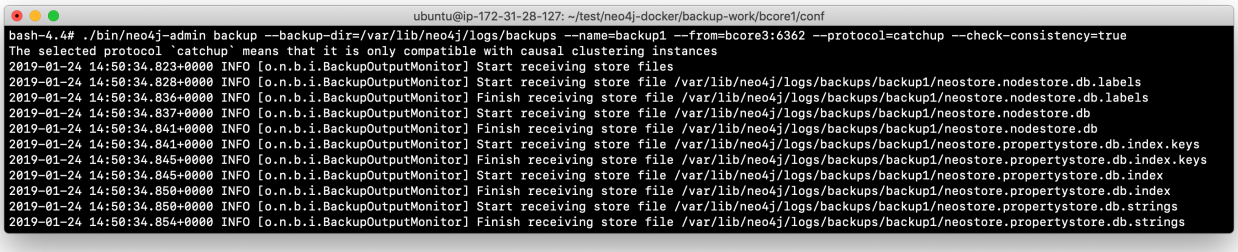
You log in to the server from where you will be performing the backup (typically a read replica server) and then you perform the backup with these suggested settings:

```
neo4j-admin backup --backup-dir=<backup-path> --name=<backup-name> --from=<core-server:backup-port> --protocol=catchup --check-consistency=true
```

You can add more to the backup command as you can read about in the Neo4j Operations Manual.

Example: Backing up the cluster

In this example, we have logged in to the read replica server and we perform the backup using the address and backup port for the *LEADER*, *bcore3*. We also specify the location of the backup files and also that we want the backup to be checked for consistency.



Note that this is not an encrypted backup. You will learn about encryption later in this training when you learn about security.

Exercise #6: Backing up a cluster

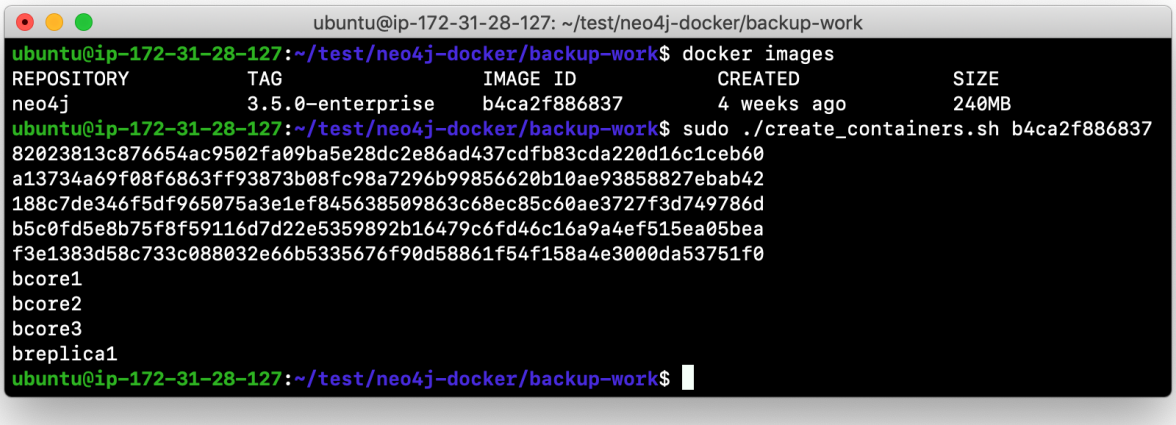
In this Exercise, you gain some experience backing up a cluster. Because the Docker containers are created without backup configured, in order to back up a cluster, you will need to create a different network that will be used for testing backups. Then you will create the core servers and read replica server to work with backing up the database.

Before you begin

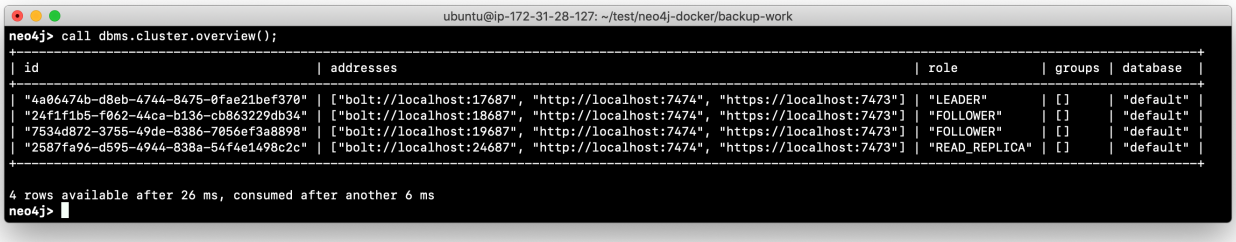
Stop all core and read replica servers.

Exercise steps:

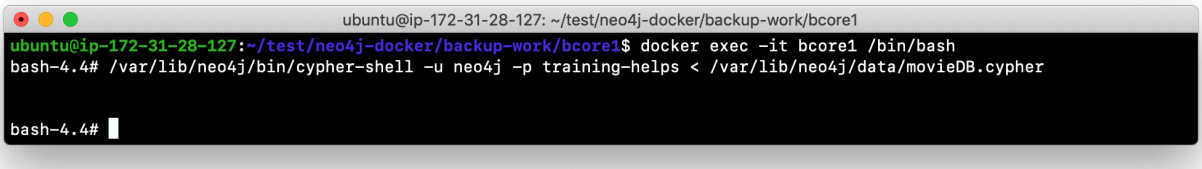
1. In the **neo4j-docker/backup-work** folder, there is a script called **create_containers.sh** that creates a new docker network named *test-backup-cluster*, creates three core servers: *bcore1*, *bcore2*, *bcore3*, and a read replica server, *breplica1*. Examine this script and notice that the core servers and the read replica server is configured for backup using the backup port. Any of the core servers can be used for the backup.
2. Run the script to create the containers specifying the Image ID of the Neo4j image.



3. Start all the servers and change the default password for each server.
4. Confirm that all core servers and the read replica are running in the cluster.



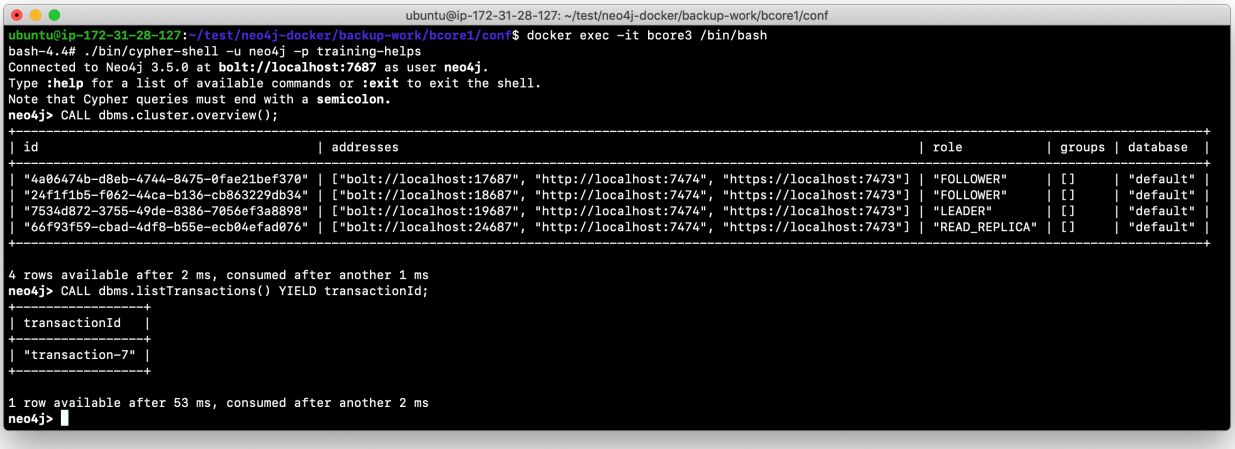
5. Seed the cluster by loading the movie data into one of the core servers.



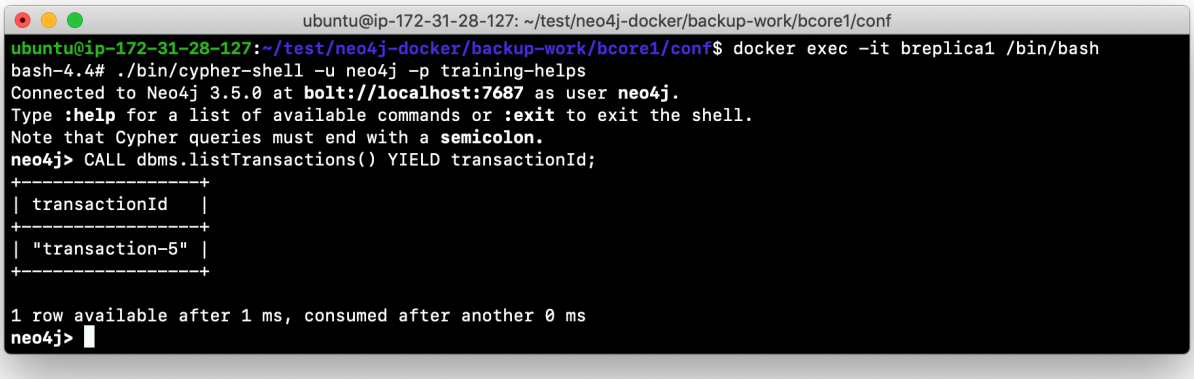
6. Shut down all servers as you will be modifying their configurations to enable backups.
7. For each core servers, add these properties to the end of each **neo4j.conf** file where *X* is the *bcore* number:

```
dbms.backup.enabled=true
dbms.backup.address=bcoreX:6362
```

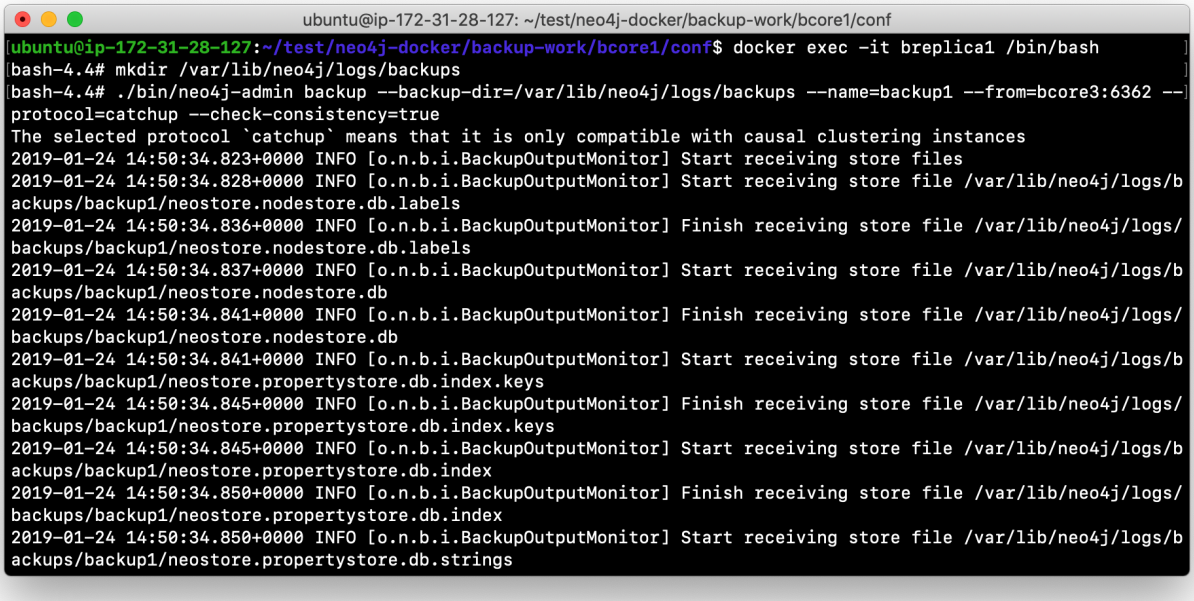
8. Start the core servers and the read replica server.
9. Check the last transaction ID on the core server that is the *LEADER*.

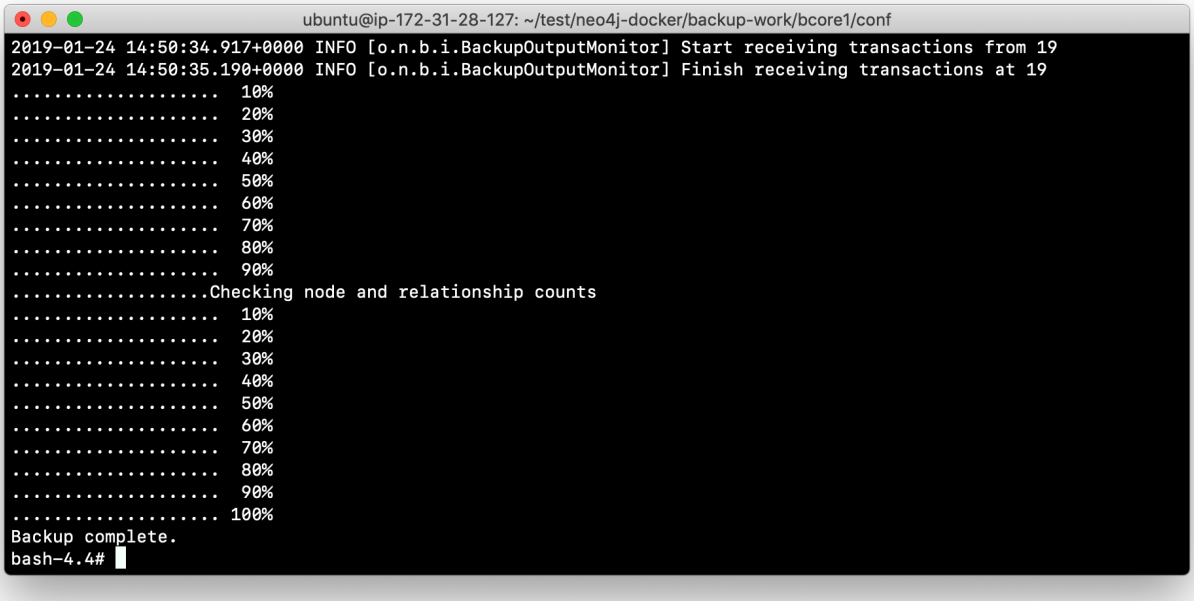


10. Log in to the read replica server and check the last transaction ID. This server will have a different last transaction ID, but in a real application, you will find that this ID value increases at the same rate as it increases in the core servers.

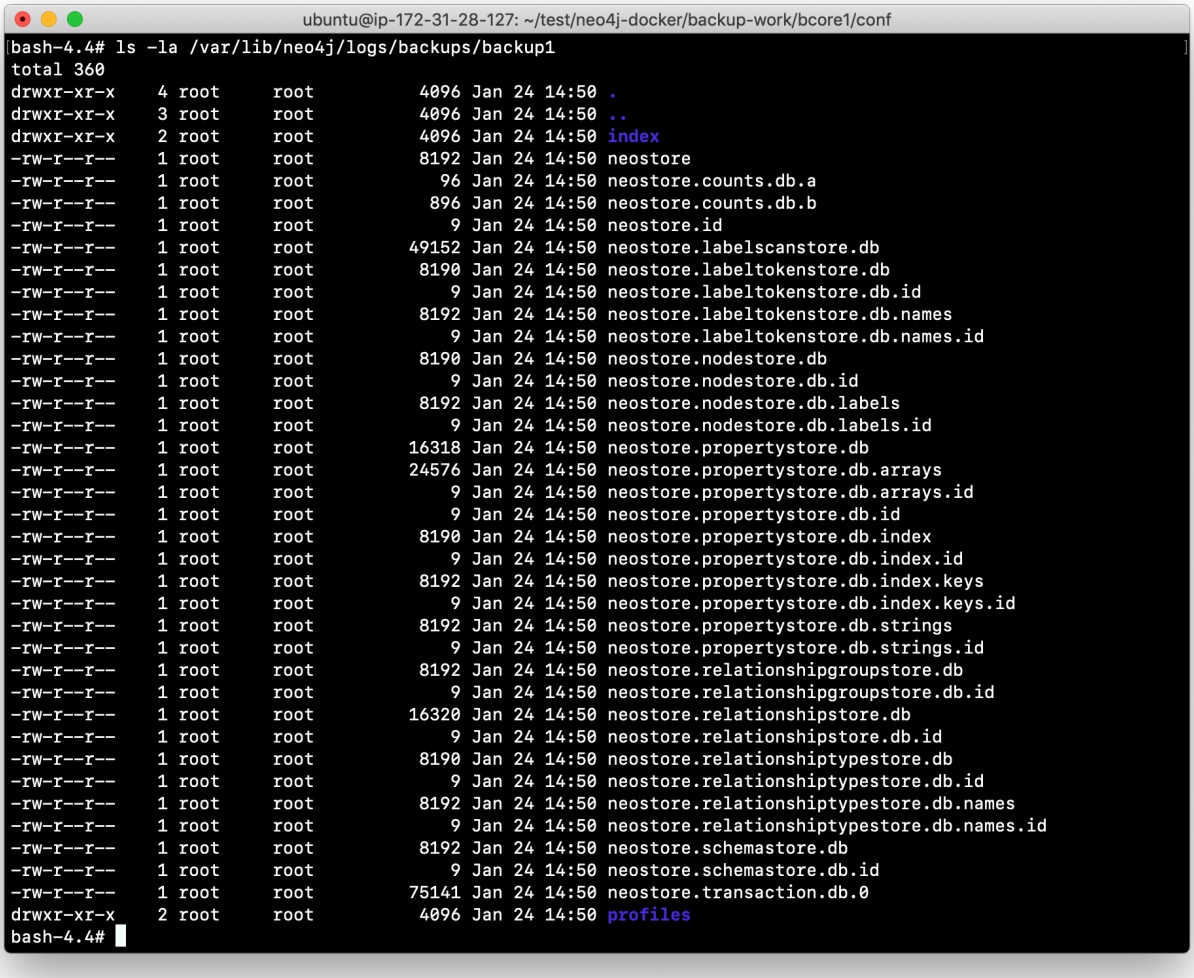


11. While still logged in to the read replica, create a subfolder under **logs** named **backups**.
12. Perform the backup using `neo4j-admin` specifying the *LEADER* port for the backup, use the *catchup* protocol, and place the backup the **logs/backups** folder, naming the backup *backup1*.





13. Confirm that the backup files were created.



Check your understanding

Question 1

Suppose you want to set up a cluster that can survive at least two failures and still be considered fault-tolerant. How many *LEADERS* and *FOLLOWERS* will this cluster have at a minimum?

Select the correct answer.

- ▢ One *LEADER* and two *FOLLOWERS*
- ▢ One *LEADER* and four *FOLLOWERS*
- ▢ Two *LEADERS* and three *FOLLOWERS*
- ▢ Two *LEADERS* and two *FOLLOWERS*

Question 2

What protocol should application clients use to update a database in the cluster?

Select the correct answer.

- ▢ bolt+routing
- ▢ bolt
- ▢ cluster+routing
- ▢ cluster

Question 3

In a cluster, which servers have their own databases?

Select the correct answers.

- ▢ Core servers with the role of *LEADER*
- ▢ Core servers with the role of *FOLLOWER*
- ▢ Read replica servers
- ▢ Primary server for the cluster

Summary

You should now be able to:

- Describe why you would use clusters.
- Describe the components of a cluster.
- Configure and use a cluster.
- Seed a cluster with data.
- Monitor and manage core servers in the cluster.
- Monitor and manage read replica servers in the cluster.
- Back up a cluster.

Security in Neo4j

Table of Contents

About this module

Now that you have gained experience managing a Neo4j instance and database , as well as managing Neo4j Causal Clusters, you will be introduced to how to secure a Neo4j application.

At the end of this module, you should be able to:

- Describe what security means for an application.
- Configure a Neo4j instance to use LDAP as the authentication provider.
- Describe how to secure data-in-transit.
- Describe how to secure data-at-rest.
- Configure and use security auditing.

What is application security?

Many applications today are distributed and provide services and data over the Internet. Users include customers, partners, and off-site employees. In this interconnected world, security is a major concern. There are malicious actors who will attempt to access your application's data, causing disruption and financial loss to your business. There are numerous ways that can be used to attack the data via the Internet, intranet, and even the 'human' back door. Fixing a security breach may require significant updates to your application, especially if the application was not written or implemented to be secure. Deploying a newer version of your application means additional testing and possible down time, both costly to the enterprise.

In addition, your business may need to meet regulatory security requirements or security requirements promised in a customer contract. If it is discovered that your application does not meet these requirements, you could face fines or other serious consequences.

To secure your application, you need to be concerned about 3 things:

- Data security
- Access control
- User privacy

Data security

Some information in an application is private must be protected. Exposing a customer's private information could ruin the reputation of your enterprise, which could take years to recover from. Here are some examples of the types of private data your application needs to secure:

- Company financial data such as salaries, revenue, operating expenses, etc.
- User pass-phrases for the application
- Personal data such as names, credit card numbers, telephone numbers, social security numbers, etc.
- Intellectual property such as code

Access control

Controlling who accesses your application and what they have access to is very important to the business. You want to ensure that only valid users can use the application and malicious actors are kept out. In addition, you want to control which users can access specific application functionality and monitor what users do.

User privacy

Another concern in applications that extend to the Internet is user privacy. When a Web or mobile app requests data from your application, you need to ensure that only the required data is retrieved and sent to the app. That is, if the app requests personal information, such as a driver's license ID, the data returned to the app contains only the ID for the user requested and not the IDs of multiple users that the app, then has to filter for display.

Securing a Neo4j application

Neo4j applications consist of many parts, including databases, static files like images and document scans, application code, application servers and Web servers. These all need to be secured and you use different techniques and technologies to secure them.

There are many things you can do to make your Neo4j application secure. At the highest level, there are 2 areas you must address:

1. Building security into your Neo4j application.
2. Securing the deployed application and infrastructure.

Building security into your Neo4j application

The primary facets of building security into your Neo4j application include:

- Authentication — Is the user who they say they are?
- Authorization — Is a user allowed to do what they are attempting to do?
- Auditing – Create a record of who did what and when (so that you can monitor activity and investigate security breaches).

At a high level, the implementation of how your application performs authentication, authorization, and auditing can be configured by you as an administrator. However, there are aspects of security that may cross into application code. For example, there may be a specific procedure written that can only be executed by certain users. The procedure must be annotated and configured as such. In addition, code may be written to check roles at runtime for authorization.

Securing the deployed application and infrastructure

Securing a deployed Neo4j application and its infrastructure includes securing Neo4j instances and non-Neo4j server processes they communicate with, filesystems, networks, etc. This can include:

- Data-in-transit — securing data transmitted over the network.
- Data-at-rest — securing private data in Neo4j database.
- OS level resources — securing networks and filesystems.
- Server processes — Neo4j instances, application servers, connectors, and Web servers.
- Application-related files—securing application-related files outside of the database.

In this module, you will be introduced to how you as an administrator can secure those parts of the application related to Neo4j. This module will not cover tasks related to securing non-Neo4j resources, networks, and filesystems.

Neo4j software updates

As an administrator, you must be aware of all software on your production systems and keep up-to-date with the software. In particular, you must ensure that the Neo4j version you are using has the the latest versions and patches, especially those that address security.

Authentication in Neo4j

There are three types of authentication frameworks supported by Neo4j:

- Native user authentication
- Custom-built authentication
- Single Point of Authentication (SPA)

Native user authentication means that users are created in the Neo4j database and authentication is performed based upon those values. Most enterprise applications do not use native user authentication in their deployed application.

Your application developers could write a custom authentication plugin. Although this is possible, the underlying internal procedures called by the custom authentication plugin could change in future releases of Neo4j so it is best to avoid this type of authentication for a deployed, secure application that will survive upgrades of Neo4j.

A SPA is highly recommended by Neo4j because it is easier to maintain and is more secure than an enterprise that uses multiple sources of user accounts. Examples of a SPA are Lightweight Directory Access Protocol (LDAP), Active Directory (AD), and Kerberos which can be used for single sign-on. The SPA is the only service in your enterprise that stores user names and passwords. For training purposes, we will use an LDAP provider for authentication, but in your real application, you will need to configure Neo4j for whatever provider your application uses for user authentication.

You will use OpenLDAP for the hands-on Exercises of this module.

If you will be using a different LDAP or authentication provider in your real application, you should consult the Neo4j Operations Manual.

Enabling authentication

A secure Neo4j instance should always have authentication enabled. By default, authentication is enabled for a Neo4j instance. You can explicitly set it in **neo4j.conf** as follows:

```
dbms.security.auth_enabled=true
```

How roles are used for users

Even though roles are not required for authentication, they are for authorization to access Neo4j. Roles must be built into the configuration of a SPA, for example LDAP. Roles are used at runtime to authorize how the current user can access Neo4j resources (data or procedures).

Refer to this table in the Neo4j Operations Manual that describes Neo4j native roles. One of these pre-defined, native roles must be associated with any user that will be connecting to the Neo4j instance for general access. So for example, the user with the role of *reader* will only have read access to the database while a user with the role of *publisher* will be able to create data in the database.

In addition, you can define custom roles that are application-specific and used together with specific custom procedures.

Using LDAP for authentication and authorization

There are many LDAP providers an enterprise can use including Active Directory and Open LDAP.

The tasks for using LDAP for authentication with Neo4j include:

1. Add role information to LDAP (groups).
2. Configure Neo4j for LDAP authentication.
3. Test.

Adding groups for roles in LDAP

In your enterprise, you will need to work with the system administrator responsible for maintaining the user directory (LDAP). Part of the configuration that must be modified for LDAP is the addition of groups for users. The groups added to LDAP and associated with specific users correspond to roles in Neo4j. An easy way to identify a group in LDAP is with a group ID.

For example, the LDAP would have these entries that correspond to the Neo4j native *reader* and *publisher* roles for the training **neo4jtraining.com** domain. The attribute that is used to identify the groups is *gidnumber*.

```
# reader group
dn: cn=reader,ou=groups,dc=neo4jtraining,dc=com
objectClass: posixGroup
objectClass: top
cn: reader
gidnumber: 501

# publisher group
dn: cn=publisher,ou=groups,dc=neo4jtraining,dc=com
objectClass: posixGroup
objectClass: top
cn: publisher
gidnumber: 502
```

Associating users with groups in LDAP

In addition, each user that will be authenticated for connecting to the Neo4j instance, will need to be modified in the LDAP to be associated with a specific group. Here is are examples of the *reader* and *publisher* users that you will be working with in this training:

```
# user reader
objectClass: organizationalPerson
objectClass: person
objectClass: extensibleObject
objectClass: uidObject
objectClass: inetOrgPerson
objectClass: posixAccount
objectClass: top
cn: Reader User
givenName: Reader
sn: reader
uid: reader
gidnumber: 501
uidNumber: 1000
homedirectory: /home/users/reader
mail: reader@neo4jtraining.com
ou: users
userpassword: reader

# user publisher
dn: uid=publisher,ou=users,dc=neo4jtraining,dc=com
objectClass: organizationalPerson
objectClass: person
objectClass: extensibleObject
objectClass: uidObject
objectClass: inetOrgPerson
objectClass: posixAccount
objectClass: top
cn: Publisher User
givenName: Publisher
sn: publisher
uid: publisher
gidnumber: 502
uidNumber: 1001
homedirectory: /home/users/publisher
mail: publisher@neo4jtraining.com
ou: users
userpassword: publisher
```

There are many ways to configure groups and users in LDAP. This is just an example of how the LDAP is configured that you will be working with in this training. Notice that each group has an ID, *gidnumber* and each user has an ID, *uidNumber*. The group ID is used to map to roles in Neo4j.

Note that if a user does not specify a group ID, it must specify an attribute *memberOf* that is set to the designated name, *dn* of the group.

Querying the LDAP Server

The Neo4j instance never writes to the LDAP Server. Before you begin configuring the Neo4j instance to use LDAP, you should confirm that the LDAP Server is properly configured for use with Neo4j. You query the LDAP Server by using the `ldapsearch` utility from the ldap-utils library that is available on OS X and Linux, and by `ldp.exe` on Windows.

To understand the groups that users are associated with in the LDAP, you should perform this type of query:

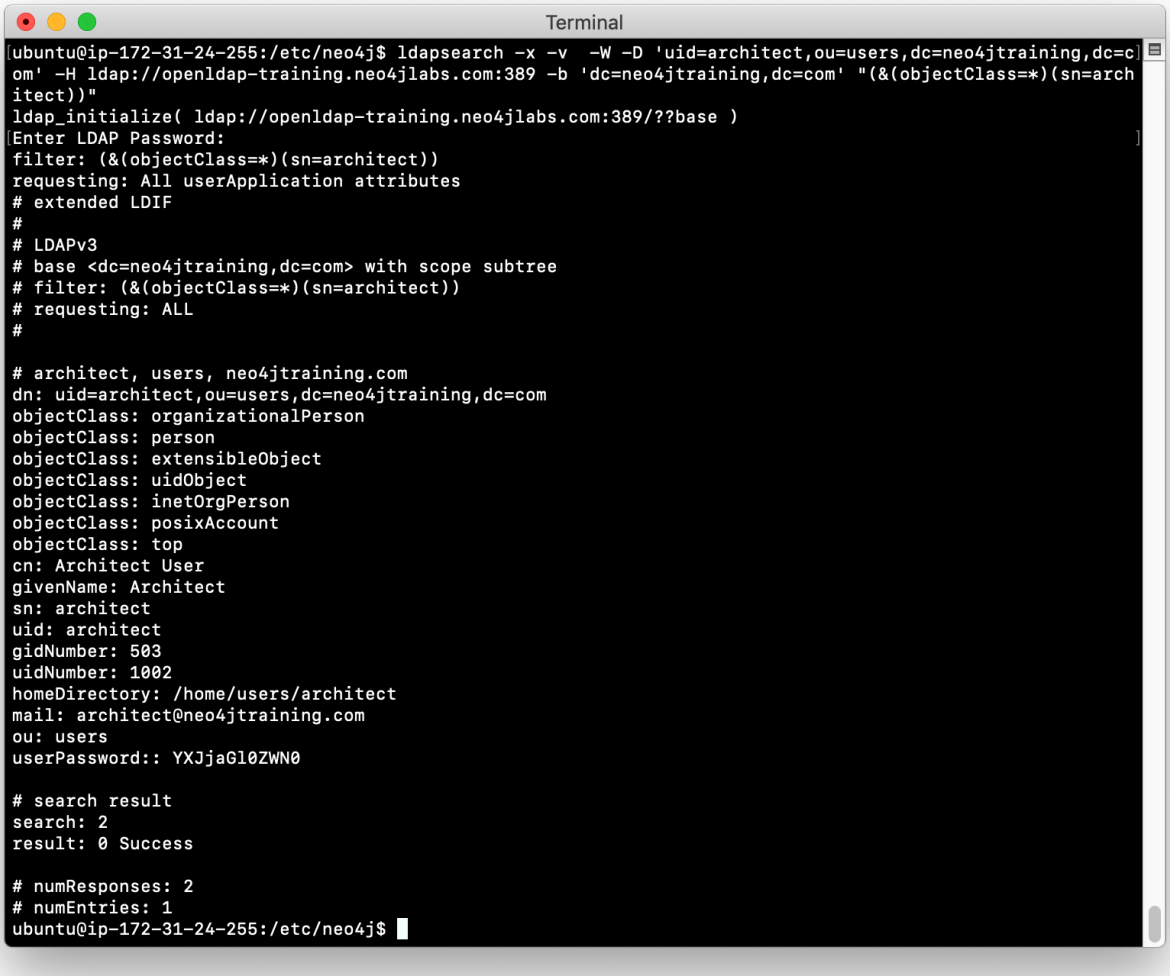
```
ldapsearch -x -v  -W -D '<designated-name-for-user>'
                     -H <ldap-server>:<port>
                     -b '<search-base>'
                     "<filter-for-user>"
```

Example: Retrieving the user, architect

Here is an example for retrieving the user *architect* in the training LDAP Server:

```
ldapsearch -x -v  -W -D 'uid=architect,ou=users,dc=neo4jtraining,dc=com'
           -H ldap://openldap-training.neo4jlabs.com:389
           -b 'dc=neo4jtraining,dc=com'
           "(&(objectClass=*)(sn=architect))"
```

Here is the result of executing this `ldapsearch` command. When you execute this type of search, you must provide the password for the user which in this LDAP is *architect*.



In this example, you see that this particular user is part of many object classes. In a large user directory, you should provide a more specific filter to improve the search.

Example: Optimized retrieval of the user, architect

For example, a user directory will contain entries for non-users such as printers. If you want to focus the search only on people, you would specify something like the following to filter by *person*:

```
ldapsearch -x -v  -W -D 'uid=architect,ou=users,dc=neo4jtraining,dc=com'
           -H ldap://openldap-training.neo4jlabs.com:389
           -b 'dc=neo4jtraining,dc=com'
           "(&(objectClass=person)(sn=architect))"
```

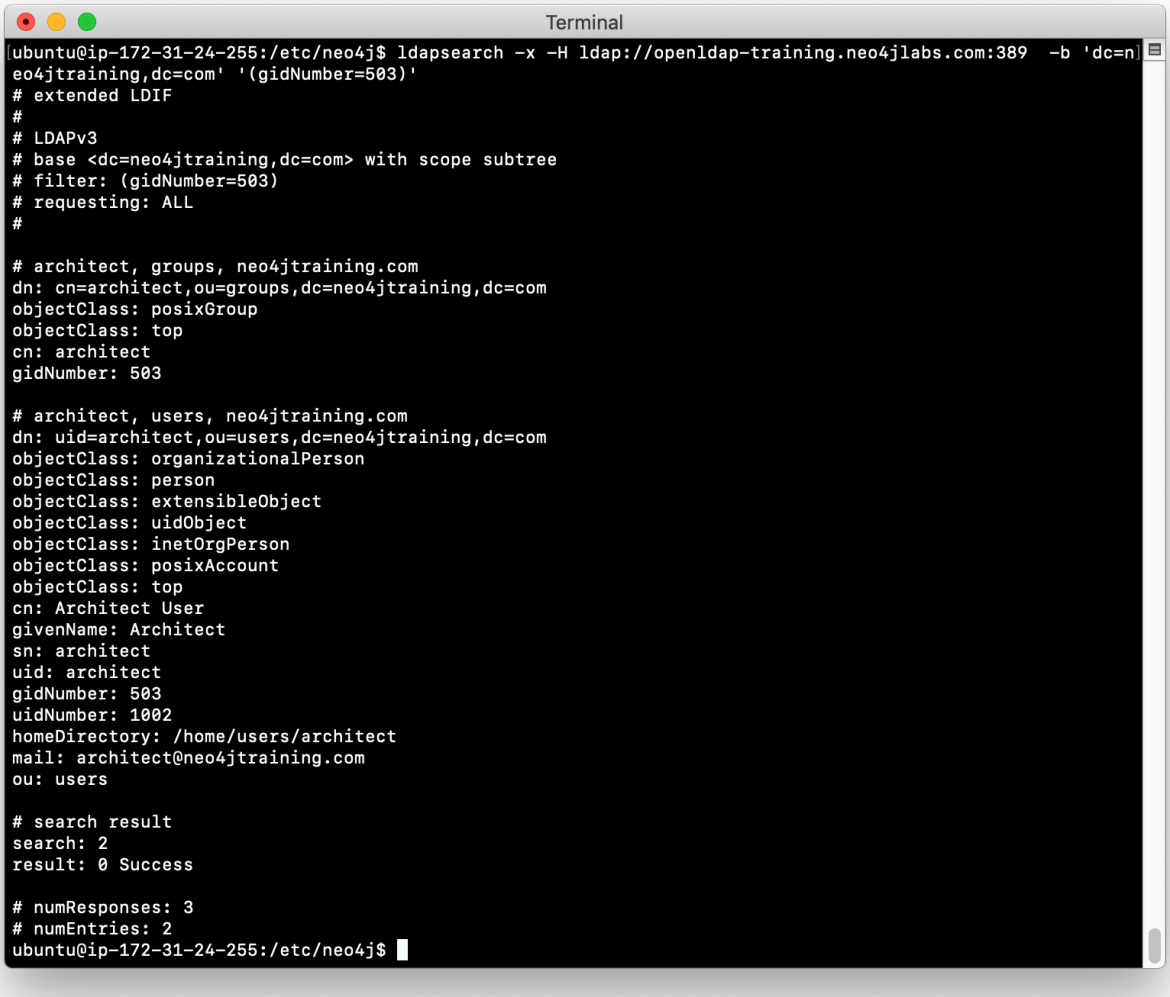
Example: Retrieving entries for a group

For integration with Neo4j, you must examine the LDAP entries to understand what groups are defined as they will need to be mapped to Neo4j native roles and possibly custom roles. In the previous image, you saw that the *gidNumber* attribute is assigned a value of 503. In LDAP, a user can be a member of more than one group.

For example, once you determine the attribute that is used to define group membership in the LDAP, you can perform a query to retrieve all entries that use that same *gidNumber* of *503*:

```
ldapsearch -x -H ldap://openldap-training.neo4jlabs.com:389  -b
'dc=neo4jtraining,dc=com' '(gidNumber=503)'
```

And here we see an entry for a user and an entry for a group.

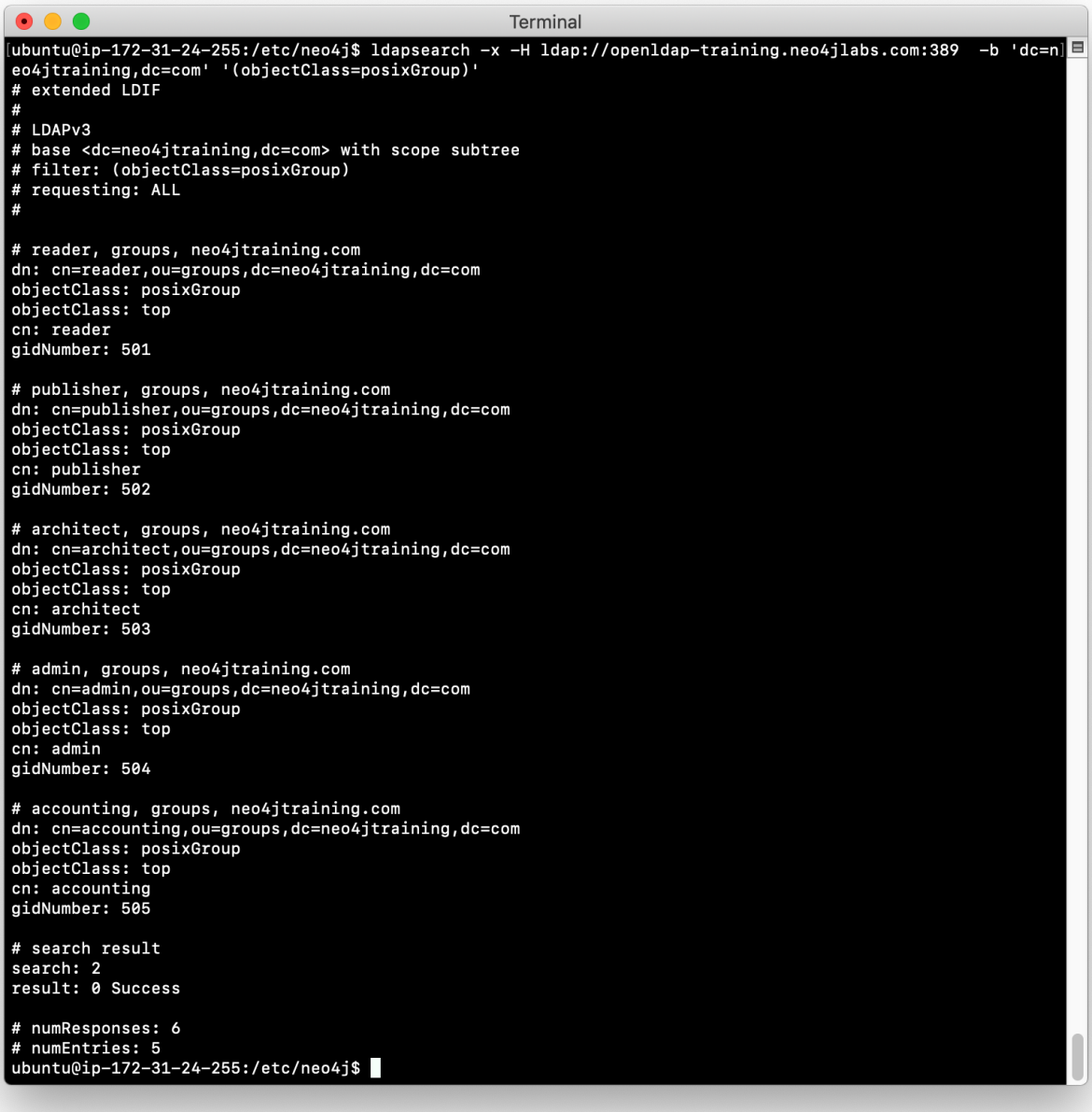


Notice that the group entry has an *objectClass* of *posixGroup*. This tells you that all groups are defined with this attribute in this particular LDAP.

Example: Retrieving group entries

So, for example, you can then do a query to find all entries that have this attribute to learn about which groups are defined in the LDAP:

```
ldapsearch -x -H ldap://openldap-training.neo4jlabs.com:389  -b
'dc=neo4jtraining,dc=com' '(objectClass=posixGroup)'
```



What queries you perform on your LDAP provider will depend on the discovery of entries in the LDAP.

Exercise #1: Exploring the training User Directory

In this Exercise, you will simply execute a couple of commands to retrieve data from an existing LDAP server that will be used for the Exercises of this training.

Before you begin:

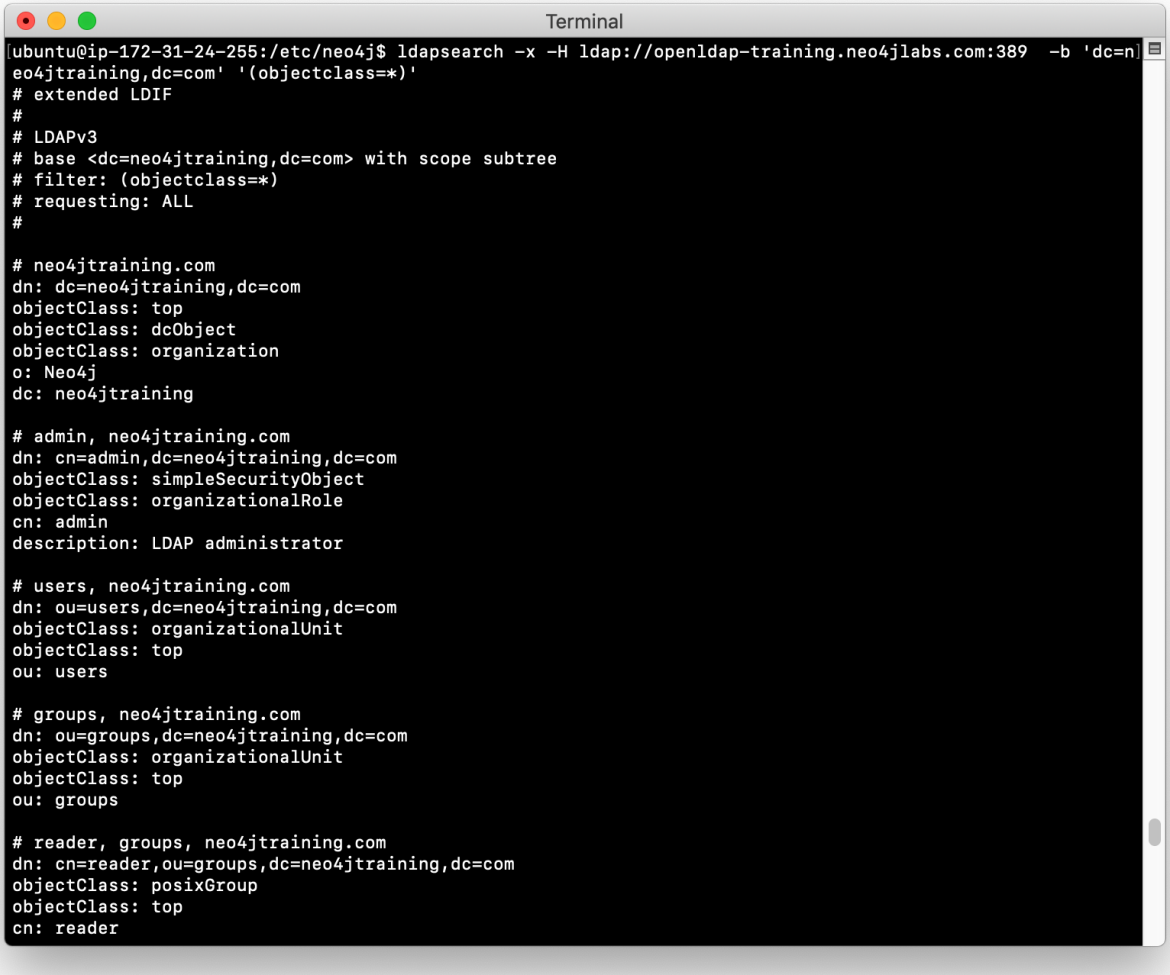
1. Open a terminal window on your system where you have worked with a stand-alone Neo4j instance in the *Managing a Neo4j Database* module.
2. Ensure that you have the LDAP utilities package installed on your system. (For example on Debian: `sudo apt-get install ldap-utils`).

Exercise steps:

In this example, and for the LDAP provider (an EC2 instance) you will be using for the Exercises in this module, the users are all part of the **neo4jtraining.com** domain.

1. Execute this `ldapsearch` command to retrieve all entries from the LDAP Server:

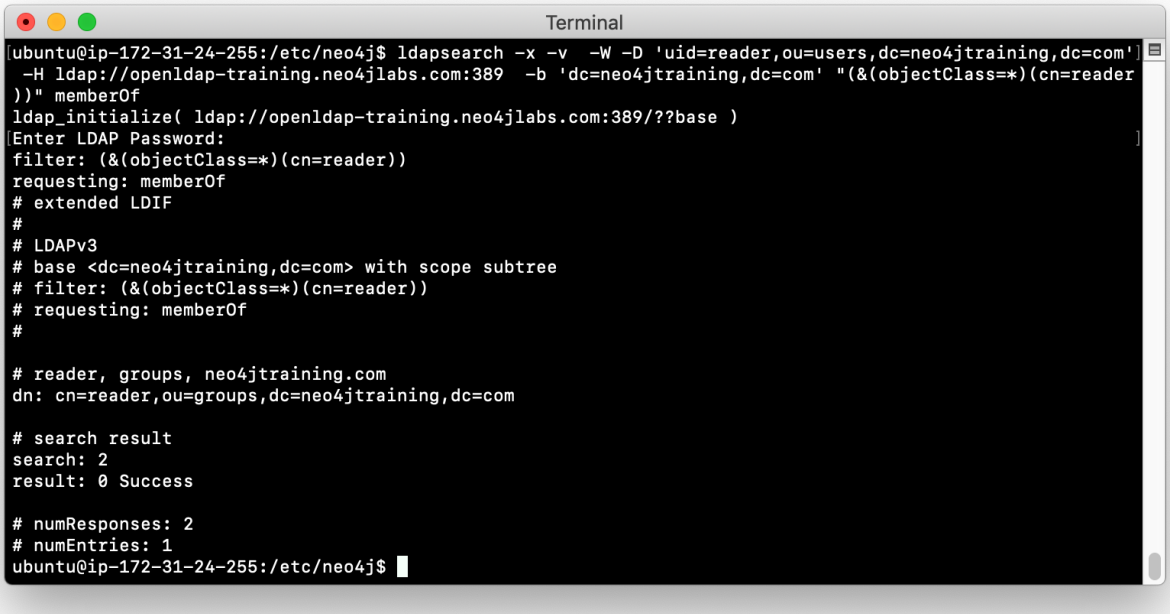
```
ldapsearch -x -H ldap://openldap-training.neo4jlabs.com:389  -b
'dc=neo4jtraining,dc=com' '(objectclass=*)'
```



NOTE In an enterprise LDAP, you will probably not want to perform this type of query as it will return too much information, but in our training environment, you can perform this type of query to understand want is defined in our training LDAP Server.

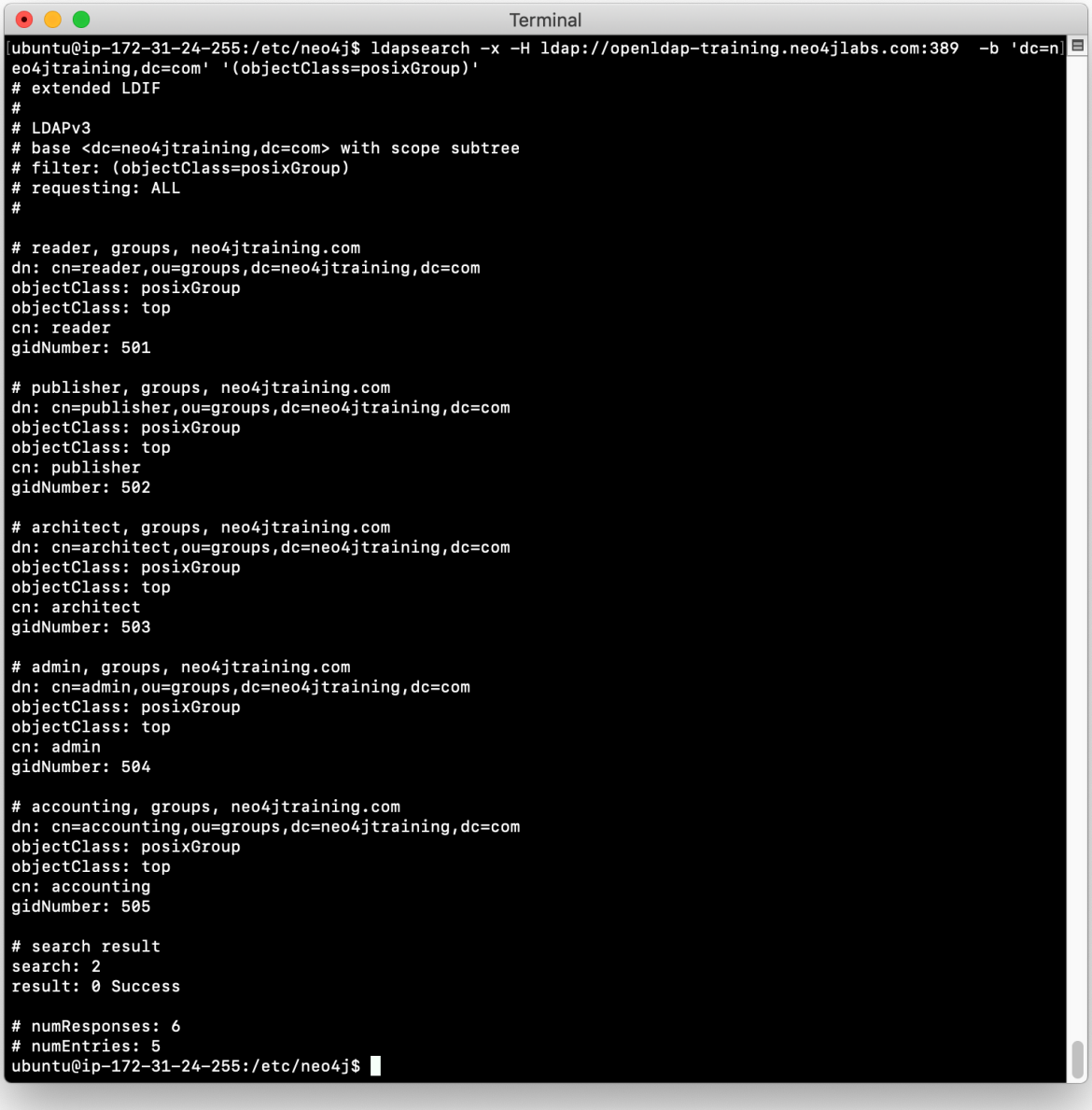
2. Execute this `ldapsearch` command to retrieve the *reader* entry from the LDAP Server. When prompted for the password, enter *reader*.

```
ldapsearch -x -v  -W -D 'uid=reader,ou=users,dc=neo4jtraining,dc=com' -H
ldap://openldap-training.neo4jlabs.com:389  -b 'dc=neo4jtraining,dc=com'
"(&(objectClass=*)(cn=reader))" memberOf
```



3. Execute the search for another user in the LDAP, for example *publisher*.

4. Execute the search for returning all group entries in the LDAP.



You have now confirmed that you can access the LDAP Server that will be used for authentication with the Neo4j instance and you have explored the entries in the LDAP.

Configuring Neo4j for LDAP authentication and authorization

There are many configuration settings related to authentication and authorization in the **neo4j.conf** file. The Neo4j Operations Manual describes how you can configure Neo4j to connect to and use an LDAP provider.

You can configure Neo4j to work with:

- Active Directory
- Active Directory using a sAMAccountName
- openLDAP

Each of these LDAP providers require different settings for authentication. For example using *AD* and *openLDAP*, you specify a value for *dbms.security.ldap.authentication.user_dn_template* and for *AD* with *sAMAccountName*, you specify values for other properties.

Using *AD* with *SAMAccountName* gives you the greatest flexibility because with this configuration, you do not need to specify a fixed *dn* template for authentication. Most enterprise deployments use this option.

There are several properties you specify for authorization and how you set these properties will depend on your LDAP provider.

Settings for LDAP group membership

In summary, there are three options for authorization with LDAP to establish the user's group membership:

Settings	Description
use_system_account = false	The user will search their own group membership during authentication.
use_system_account = true *use_samaccountname=false*	The system account will log in and search for the users group membership.
use_system_account = true *use_samaccountname=true*	The system account will log in and search for the user based upon their *samaccountname* and will do both group membership as well as return the users *dn* so that bind (authentication) can be completed for the user.

Configuration for openLDAP

For *openLDAP*, you set up a stand-alone or core and read replica servers in a cluster for authentication using an LDAP Server and configure the following:

```
dbms.security.auth_enabled=true
dbms.security.auth_provider=ldap
#---
dbms.security.ldap.host=<host IP address where LDAP Server runs>
#---
dbms.security.ldap.authentication.mechanism=simple
dbms.security.ldap.authentication.user_dn_template=uid={0},<top-level entity for
users>
#---
dbms.security.ldap.authorization.use_system_account=false
dbms.security.ldap.authorization.user_search_base=<top-level entity for users>
dbms.security.ldap.authorization.user_search_filter=(&(objectClass=*)(uid={0}))
dbms.security.ldap.authorization.group_membership_attributes=<attribute-used-to-
define-groups-or-members>
dbms.security.ldap.authorization.group_to_role_mapping=\
    <group-or-member >  = reader; \
    <group-or-member >  = publisher; \
    <group-or-member >  = architect; \
    <group-or-member >  = admin; \
    <group-or-member > = <custom-role>
```

Example: Training configuration

For example, with the LDAP Server that you will be using in this training, you specify:

```
dbms.security.auth_enabled=true
dbms.security.auth_provider=ldap
#---
#LDAP Server running as EC2 instance
dbms.security.ldap.host=openldap-training.neo4jlabs.com
#---
dbms.security.ldap.authentication.mechanism=simple
# users are defined under ou=users,dc=neo4jtraining,dc=com in LDAP
dbms.security.ldap.authentication.user_dn_template=uid={0},ou=users,dc=neo4jtraining,d
c=com
#---
dbms.security.ldap.authorization.use_system_account=false
# limit where the search starts so entire LDAP is not searched
dbms.security.ldap.authorization.user_search_base=ou=users,dc=neo4jtraining,dc=com
# limit the entries that are searched to be people
dbms.security.ldap.authorization.user_search_filter=(&(objectClass=person)(uid={0}))
# in this LDAP the gidnumber attribute is used to define groups
dbms.security.ldap.authorization.group_membership_attributes=gidnumber
# each group is mapped to a Neo4j native role and to the custom accounting role
dbms.security.ldap.authorization.group_to_role_mapping=\
    501  = reader; \
    502  = publisher; \
    503  = architect; \
    504  = admin; \
    505 = accounting
```

NOTE	Just as the LDAP can be configured to allow users to be members of multiple groups. You can also specify multiple groups to be mapped to the same role.

After you have made these configuration changes for the Neo4j instance, you restart the instance and then test that users in the LDAP can access the Neo4j instance.

The **security.log** file contains log records of all users that connected to or attempted to connect to the Neo4j instance. Later in this training, you will learn more about monitoring and logging. If you need to troubleshoot an authentication issue, you can set *dbms.log.security.level=DEBUG* in your Neo4j configuration to see more information about the login attempts. In most cases the troubleshooting will need to occur on the LDAP Server side to better understand the requests received from the Neo4j instance.

Multiple authentication providers

It is possible to use native and another authentication provider together in situations where a small number of users would be maintaining the database, but most of the users would use the enterprise authentication provider such as LDAP. To to this, your configuration setting would be:

```
dbms.security.auth_providers=native,ldap
```

Turning off role mapping (temporary)

In a troubleshooting situation, you may need to give all users access to the database. You can start the Neo4j instance where all role mapping is bypassed. In this example, all groups with the ID of user would have the reader role:

```
dbms.security.ldap.authorization.group_membership_attributes=objectClass
dbms.security.ldap.authorization.group_to_role_mapping="user" = reader
```

In the next Exercise, you will configure a Neo4j instance to use authentication with the LDAP Server you accessed in the previous exercise.

Exercise #2: Configuring authentication for a stand-alone instance

In this Exercise, you will modify the configuration for the stand-alone Neo4j instance that you have worked with in the *Managing a Neo4j Database* module, prior to using Docker for Causal Clustering. You will use the **movies3.db** that you worked with previously with the Neo4j stand-alone instance. Then you will test that the authentication is working.

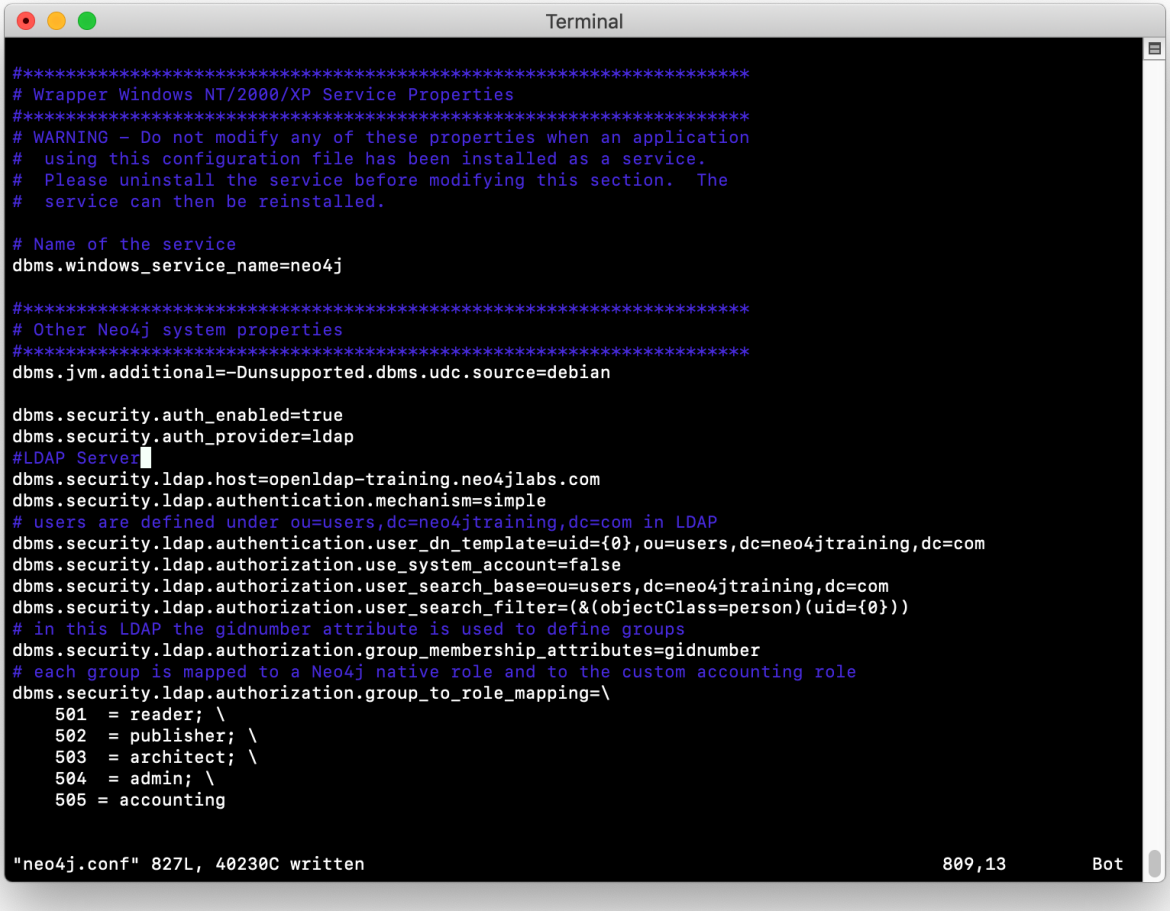
Before you begin

1. Make sure you have completed Exercise 1 that confirms that you can access the LDAP Server.
2. Modify **neo4j.conf** to use **movie3.db** as the active database.
3. Start or restart the Neo4j instance and confirm that it starts without error.
4. Open a terminal window on your system where you have worked with a stand-alone Neo4j instance using the **movie3.db** database.
5. Stop the Neo4j instance.

Exercise steps:

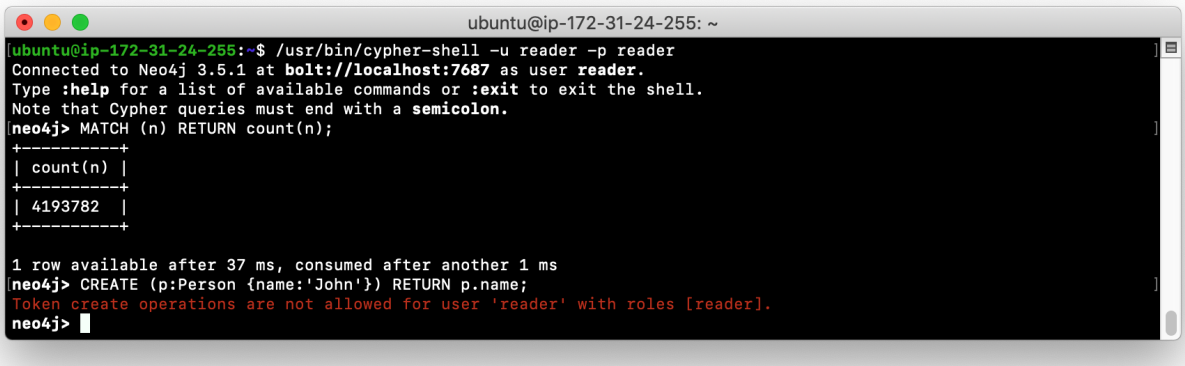
1. Make a copy of **neo4j.conf**, which is the last good configuration for this Neo4j instance.

2. Modify the properties in this file to use the training LDAP Server as the LDAP provider (openldap-training.neo4jlabs.com). You can set the properties in their locations in the **neo4j.conf** file, but a useful way to work is to add the properties you are setting to the end of the file so you can see all of your modifications in one place.



3. Start the Neo4j instance.
4. Examine the log file to ensure that the instance started without errors. If it does not start, review/adjust the properties you set in **neo4j.conf**.
5. Start cypher-shell specifying the user *reader/reader*. Can you log in?
6. Enter a Cypher statement that reads from the database `MATCH (n) RETURN count(n);`

7. Enter a Cypher statement that writes to the database `CREATE (p:Person {name:'John'}) RETURN p.name;` Did you see an error? This is because a user with the *reader* role cannot modify the database.



8. Exit out of cypher-shell.
9. Start cypher-shell specifying the user *publisher/publisher*. Can you log in?
10. Enter a Cypher statement that reads from the database `MATCH (n) RETURN count(n);`
11. Enter a Cypher statement that writes to the database `CREATE (p:Person {name:'John'}) RETURN p.name;`

12. Look at the log records in the **security.log** file. Do they correspond to your activities against the Neo4j instance?

Using LDAP for authentication, you can control which users have different types of access to the database based upon the groups defined in the LDAP. If you need to perform administrative operations against the database, you would log in as *admin/admin* which is what currently is defined for this LDAP Server used for training.

Exercise 2: Taking if further:

Configure and test the core and read replica servers you used for Causal Clustering in the previous module for authentication using the same LDAP Server.

LDAP caching

By default, the Neo4j instance caches security-related interactions between a client and the LDAP Server as an optimization. A best practice is to leave this setting. If you set it to false, the client experience slower as the authentication/authorization needs to be done with every request to the Neo4j instance. See the documentation about the *dbms.security.ldap.authentication.cache_enabled* settings and how to clear the Auth cache.

Using secure LDAP

If the LDAP provider uses encryption (LDAPS or StartTLS=true), then you must set some additional properties in **neo4j.conf** to specify that encryption will be used and what port to use. In addition, the LDAP Server Certificate needs to be added to the JVM keystore for each Neo4j instance:

```
keytool -import -alias <alias-name> -keystore ..\lib\security\cacerts -file <path-to-
cert-file>
```

This is commonly used with Active Directory. See the documentation about encrypting with your particular authentication provider.

All production environments should use an SSL certificate issued by a Certificate Authority when accessing their LDAP provider. You will work with the system administrator responsible for providing the certificate. You can, however, set up and configure a self-signed test certificate while you are setting up your Neo4j Instance to use the LDAP provider. To do this you specify a value for *dbms.jvm.additional* in **neo4j.conf** as specified in the documentation. Once all of your tests are complete, you can switch to the real SSL certificate issued by the Certificate Authority for the LDAP Server.

Securing data-in-transit

Users “in the building”, such as employees can access the application from an internal network. These networks are secure. In addition, “off site” employees typically access the application using a Virtual Private Network (VPN) which is also secure. Customers or other users who access the application over the Internet, by default, do not use a secure connection. If these customers send private data like pass-phrases and credit card numbers, it can be examined, stolen, or altered by a malicious actor somewhere on the Internet. To secure their connection, you must provide SSL/TLS access your application. SSL/TLS is a software layer that encrypts data sent and received over the Internet.

How to secure data-in-transit

If your application has users that access the application over the Internet, you should implement SSL/TLS by doing the following.

- Identify which server processes are accessed by users over the Internet.
- Identify server ports that will be used for these server processes.
- Configure each server process to use SSL ports.
- Create and publish secure digital certificates for each server process.

A digital certificate is used by client applications when they communicate with the server processes using SSL/TLS. A digital certificate must have an expiration date.

NOTE	You must ensure that the version of SSL/TLS software your application uses is up-to-date and that it supports all the types of clients that will be accessing your application. For example, not all browsers are compatible with the latest SSL/TLS versions.

Recommendations for external users

You should recommend that external users store the following artifacts securely:

- Digital certificates
- User credentials (encrypt if they are included in code or a configuration file; never use clear-text credentials)

Configuring the Neo4j instance for SSL/TLS

Because the use of certificates is secure and requires a domain that you own as well as certificates from a trusted Certificate Authority (CA), you will learn how to configure a Neo4j instance, but will not perform the steps in your training environment. The creation of certificates and keys should be used in your real production environment.

Here are the steps you should take for your production system.

Step 1: Install JCE

1. To use SSL/TLS with the Neo4j instance, you must first ensure that the Java Cryptography Extension (JCE) is installed on the host machine.
2. Download the JCE zip file from the above location or from here
3. Unzip the file to create **local_policy.jar** and **US_export_policy.jar**.
4. Navigate to the Java 8 **jre/lib/security/policy/unlimited** folder.
5. Save a copy of the existing files.
6. Copy the two **.jar** files you unzipped into this **unlimited** folder.

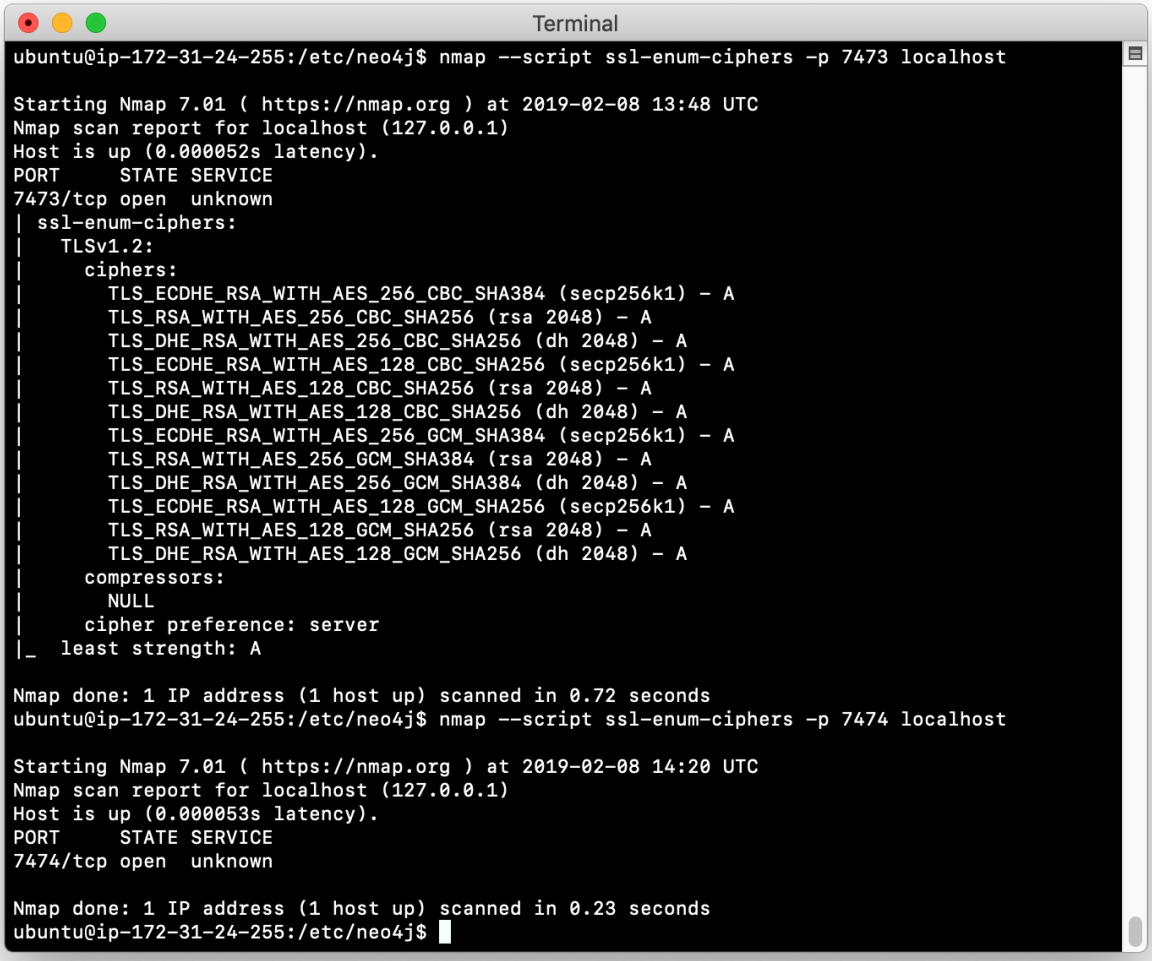
Confirming ports are SSL-enabled

Your Neo4j instance can only provide SSL if you have installed JCE. On Linux/Debian, you can confirm that your specific ports of the running Neo4j instance are SSL-capable by executing one of the following commands:

```
# a causal cluster raft listen address
nmap --script ssl-enum-ciphers -p 7000 localhost
# for HTTPS
nmap --script ssl-enum-ciphers -p 7473 localhost
```

Example: Ports used

Suppose we have installed JCE for use with our Neo4j instance. Here is an example where we see that the HTTPS port uses these ciphers and the HTTP port does not:



Step 2: Obtain digital certificates

Determine how many security policies you will use. For example, you may have policies for clients, backups, and cluster communications. Each policy will have its own private key (**private.key**) and public certificate (**public.crt**). You must obtain the certificates from a trusted Certificate Authority in PEM format. If your application requires multiple certificates, they can be combined into a single file.

To obtain a certificate you must first have a domain name that you own. You can buy a fixed domain name or you can buy a more flexible one from a site such as dyn.com. There are many providers of certificates you can use. Here is an article that explains how to use LetsEncrypt which works well with Neo4j instances.

Step 3: Create folders for the security policies

The default location for certificates is defined in **neo4j.conf** with the *dbms.directories.certificates* property.

In this location, create a folder for each security policy your application requires, for example **client_policy**, **backup_policy**, **cluster_policy**. Then, in each of these folders, create the **revoked** and **trusted** sub-folders.

Step 4: Place keys and certificates into your Neo4j installation

Place the **.key** and **.cert** files in the top-level folder for the security policy. Place a copy of the **.cert** file in the **trusted** folder for the security policy.

NOTE	If you are using *LetsEncrypt*, follow the instructions in the article regarding links to the certificates and keys rather than copying them.

Step 5: Configure SSL for the Neo4j instance

Here is an example of recommended settings to configure SSL for a Neo4j instance that will support SSL for all clients:

```
# client policy
dbms.ssl.policy.client_policy.base_directory=/var/lib/neo4j/certificates/client_policy
dbms.ssl.policy.client_policy.private_key=/var/lib/neo4j/certificates/client_policy/pr
ivate.key
dbms.ssl.policy.client_policy.public_certificate=/var/lib/neo4j/certificates/client_po
licy/public.cert
dbms.ssl.policy.client_policy.client_auth=REQUIRE
bolt.ssl_policy=client_policy
https.ssl_policy=client_policy

#all bolt communication must be encrypted
dbms_connector_bolt_tls_level=REQUIRED

#prevent use of http port
dbms.connector.http.enabled=false

# Web access only using HTTPS for Neo4j Browser
dbms.security.http_strict_transport_security=Strict-TransportSecurity: max-
age=31536000; includeSubDomains
```

The directory **client_policy** is where the certificate and key is available to the Neo4j instance. All bolt communication must use *tls_level*, which ensures that all credentials used for authentication and clear-text data are encrypted. Additionally, we ensure that all Web access for the Neo4j Browser must be via HTTPS.

SSL for Causal Clusters

For each Neo4j instance that will serve as a core server or read replica, you should configure SSL as you do for a stand-alone Neo4j instance. You must configure a separate policy for the cluster, for example *cluster_policy*. Each server will use both the *client_policy* and the *cluster_policy*.

Here is an example of the SSL configuration for a member of a cluster that you add to the Neo4j configuration in addition to the SSL configuration settings you saw earlier for the client policy:

```
# cluster policy
dbms.ssl.policy.cluster_policy.base_directory=/var/lib/neo4j/certificates/cluster_poli
cy
dbms.ssl.policy.cluster_policy.private_key=/var/lib/neo4j/certificates/cluster_policy/
private.key
dbms.ssl.policy.cluster_policy.public_certificate=/var/lib/neo4j/certificates/cluster_
policy/public.cert
dbms.ssl.policy.cluster_policy.client_auth=REQUIRE

# specify TLS version
dbms.ssl.policy.cluster_policy.tls_version=TLSv1.2

# specify ciphers used
dbms.ssl.policy.cluster_policy.ciphers=TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384

causal_clustering.ssl_policy=cluster_policy

# use a specfic backup policy for the cluster
dbms.backup.ssl_policy=backup_policy
```

SSL for backups

Whether you are using Neo4j stand-alone or in a cluster, a best practice is to also ensure that SSL is used for backups. To do so, you can create and configure a policy, for example *backup_policy*, for backups that also requires a certificate.

Additional settings related to data-in-transit

Here are some additional settings that you should consider to help reduce unintended access to your Neo4j instances:

```
#prevent browser from caching logging information
browser.retain_connection_credentials=false

#prevent browser staying open
browser.credential_timeout=15m

# Using a specific CORS response header
dbms.connectors.access_control_allow_origin=neo4j.com

# disable data usage collection by third parties
dbms.udc.enabled=false
```

Best practices for developers

You should work with your security administrator to come up with a set of guidelines for developers of your application to ensure that data-in-transit is secure. For example, developers should never include private data in a Cypher statement, but use parameters to avoid Cypher injections at runtime.

Securing data-at-rest

Securing data-at-rest means securing private data in a Neo4j installation including databases and backups.

An enterprise database typically contains private customer, personal, and financial data. If you do not secure the private data in the database, a malicious actor could access or steal the database and the private data would be exposed.

Securing files at the OS level

OS-level security restricts access to application files such as:

- Database files
- Application-related files outside the database
- Configuration files for the Neo4j instance
- Neo4j executables
- Application executables and libraries
- Security-related files (keys, digital certificates)

The system administrator is responsible for setting the access permissions for all files stored in the filesystem. If a malicious actor can access a directory in a filesystem that contains application files, they can steal, modify, or delete files in that directory. In addition, if the files in that directory contain private information, this information could be exposed.

Ensure that Neo4j does not run as root

Using a non-privileged, dedicated service account restricts the database from accessing the critical areas of the operating system that are not required by the Neo4j instance. This will also mitigate the potential for unauthorized access via a compromised, privileged account on the operating system.

You can determine which processes are owned by *neo4j* using: `ps -ef |grep -E “neo4j”`

By default the user, *neo4j* owns the processes running that are related to the Neo4j instance.

Neo4j file permissions

The following directories should be read-only:

- **conf**
- **import**
- **bin**
- **lib**
- **plugins**

The following directories should be read/write:

- **data**
- **logs**
- **metrics**

And the **bin** directory should be execute.

You should also set the log files to only be writable by *neo4j* and readable by *root*.

Security auditing

You must have a plan for analyzing a suspected security breach so you can respond quickly and effectively. As part of that plan, you should implement a security audit trail that captures which users logged in successfully and those that failed.

Implementing an audit trail does not secure your application against malicious actors, but the information that is captured can be used to alert you of a potential security breach or to aid in the investigation of a suspected security breach.

Here are some recommended settings for security auditing in Neo4j:

```
dbms.directories.logs=logs
# Log level for the security log. One of DEBUG, INFO, WARN and ERROR.
dbms.logs.security.level=INFO
# Threshold for rotation of the security log.
dbms.logs.security.rotation.size=20m
# Minimum time interval after last rotation of the security log before
it may be rotated again.
dbms.logs.security.rotation.delay=300s
# Maximum number of history files for the security log.
dbms.logs.security.rotation.keep_number=7
```

In a troubleshooting situation or if you want to more closely monitor access, you would set the level to *DEBUG*.

Exercise #3: Security auditing

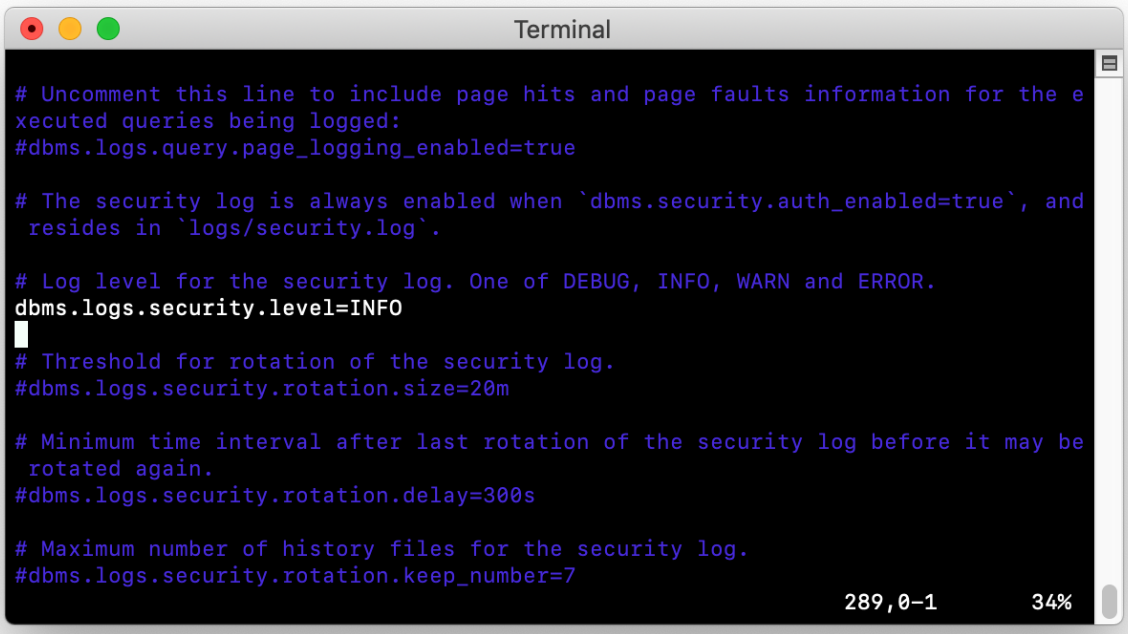
In this Exercise, you will modify the security auditing configuration for the stand-alone Neo4j instance that you have worked earlier in this module.

Before you begin

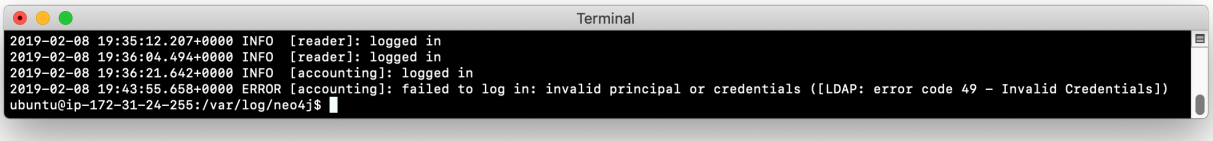
1. Make sure you have completed Exercise 2 where you have configured and tested the Neo4j instance to use the training LDAP Server.
2. Open a terminal window on your system.
3. Stop the Neo4j instance.

Exercise steps:

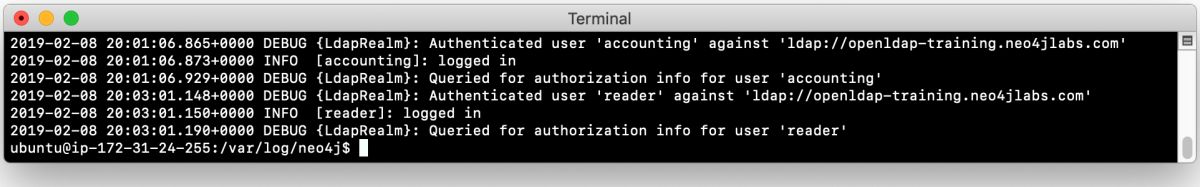
1. Make a copy of **neo4j.conf**, which is the last good configuration for this Neo4j instance.
2. Modify the property in this file log security events at the INFO level.



3. Start the Neo4j instance.
4. Examine the log file to ensure that the instance started without errors. If it does not start, review/adjust the properties you set in **neo4j.conf**.
5. Start cypher-shell specifying the user *reader/reader*.
6. Exit out of cypher-shell and log in again specifying *accounting/accounting*.
7. Exit out of cypher-shell and log in again specifying *accounting/foo*.
8. View the **security.log** file. Do you see these logins and the failed login attempt? The default level of logging is *INFO* if you do not set the property in the configuration file.



9. Modify the configuration for the security level to be DEBUG.
10. Restart the Neo4j instance.
11. Start cypher-shell specifying the user *accounting/accounting*.
12. Enter a statement that reads from the database, for example: `MATCH (n) RETURN count(n);`. You should receive an error because this user does not have the *reader* role.
13. View the log file. There is information written stating that the read access was checked against the LDAP. Even though access was denied, this information is not written to the log file.
14. Exit out of cypher-shell.
15. Start cypher-shell specifying the user *reader/reader*.
16. Enter a statement that reads from the database, for example: `MATCH (n) RETURN count(n);`.
17. View the log file. Again, you should see that the instance contacted the LDAP Server for role information.



18. Exit out of cypher-shell.
19. Stop the Neo4j instance.

So as you have seen, you can audit for logins and failed logins, even with the default security level of *INFO*.

Additional security measures

You may want to consider not using any default ports for your Neo4j instances. Hackers frequently scan IP addresses for commonly used ports, so it's not uncommon to use a different port to "fly under the radar".

The `neo4j-shell` utility has been deprecated and you should not allow any users to use it. By default, it is not enabled in a Neo4j instance.

JMX is sometimes used for monitoring system activity. JMX is not secure and by default is disabled in Neo4j.

You should work closely with Neo4j Professional Services and Technical Support to ensure that your production system is secure. See this Security Checklist will also help you to determine if you have set up your systems correctly.

Check your understanding

Question 1

Suppose you will be using an enterprise LDAP provider and you must work with the LDAP administrator to ensure that all users who will be accessing the Neo4j database will have read or write access. What needs to be added to the LDAP?

Select the correct answer.

- ▢ The Neo4j plugin for LDAP
- ▢ The Neo4j LDIF file that is installed with Neo4j
- ▢ New user entries with the *neo4j* attribute defined of with a value of *reader* or *writer*
- ▢ New group entries for neo4j roles and modifications to existing user entries to be a group member

Question 2

Suppose that you want to ensure that all data in or out of a Neo4j instance is secure (data-in-transit). What must you configure?

Select the correct answers.

- ▢ SSL policy
- ▢ Enable bolt to use TLS
- ▢ Restrict HTTP port
- ▢ Require HTTPS for Neo4j Browser

Question 3

In order to configure a security policy for SSL in Neo4j, what must you obtain?

Select the correct answers.

- ▢ Encrypted pass-phrase file to use from Neo4j support
- ▢ Digital certificate from a Certificate Authority (public.crt file)
- ▢ SSL library (.jar) from the Certificate Authority
- ▢ Private key from the Certificate Authority (private.key)

Summary

You should now be able to:

- Describe what security means for an application.
- Configure a Neo4j instance to use LDAP as the authentication provider.
- Describe how to secure data-in-transit.
- Describe how to secure data-at-rest.
- Configure and use security auditing.

Monitoring Neo4j

Table of Contents

About this module

Now that you have gained experience managing a Neo4j instance and database, managing Neo4j Causal Clusters, and the steps you must take to secure your deployed Neo4j application, you will learn how to monitor the Neo4j instance as it is used by applications.

At the end of this module, you should be able to:

- Describe the categories of monitoring and measurement you can perform with Neo4j.
- Monitor:
  - queries
  - transactions
  - connections
  - memory usage
- Manage log files.
- Manage the collection of Neo4j metrics.
- Use JMX queries.

Monitoring in Neo4j

As an administrator, you should configure your deployed Neo4j application so that you can perform routine monitoring of activity, as well as being prepared to more deeply monitor and possibly re-configure Neo4j.

The Neo4j instance writes events to log files where you can configure the level of logging you want to perform. In addition, Neo4j writes metrics to a directory that is dedicated for collecting runtime data. The Neo4j Operations Manual describes these files and locations that you will be working with in this module.

Some linux installations may use a system service to collect log files and allow you to access log data with `journalctl` as described in the Neo4j Operations Manual. You have already seen some of the events that are written to the **neo4j.log** file (`journalctl -u neo4j` on Debian) when the Neo4j instance starts and you want to confirm that it started successfully. In addition, you have seen error events written to **debug.log** when attempting to start a Neo4j instance in a Causal Cluster. You have also seen authentication events that are written to the **security.log** file when users connect to the Neo4j instance.

Monitoring log files

In the previous module, *Security in Neo4j*, you learned about the authentication events that are written to the **security.log** file. The categories of events that you configure for and monitor in log files that you will learn about in this module include:

- Queries
- Transactions
- Connections
- Memory

NOTE	You have learned how to monitor core and read replica servers in a Neo4j Causal Cluster in the module, *Causal Clustering in Neo4j*.

Collecting metrics

In addition, you can configure the Neo4j instance to collect metrics that are related to events, but can be viewed in tools (such as Grafana) that use the Graphite or Prometheus protocols to help you monitor your application. In most cases, you will want to configure a tool such as Nagios to provide alerts when certain metrics or events are detected in Neo4j. Note that you can also set up alerts in Grafana, but Nagios is a better choice for alerts. CloudWatch is another UI that is commonly use for monitoring and alerting with AWS deployments.

Monitoring queries

In a production environment, you want to know if a query is taking a long time and using too many resources. A user/application may not even be aware that their query is hung. For example, if they started a query and then walked away from their computer.

As an administrator, you can configure Neo4j to write information about queries that completed to the **query.log** file. You can provide settings that will log information about queries that took a long time to complete. You can also monitor currently running queries and if need be, kill them if they are taking too long.

Configuring query logging

You can configure Neo4j to log an event if a query runs more than xx milliseconds. There is no standard for what a reasonable period of time is for a query, but in most databases, a query that runs for minutes is not a good thing! At a minimum, you should enable logging for queries and set a threshold for the length of time queries take. Then, as part of your monitoring, you could regularly inspect the **query.log** file to determine if a certain set of queries or users are possibly performing queries that tax the resources of the Neo4j instance.

For example, here are the properties you would set in the Neo4j configuration to log a message and provide information when a query takes more than 1000ms to complete:

```
dbms.logs.query.enabled=true
dbms.logs.query.threshold=1000ms
dbms.logs.query.parameter_logging_enabled=true
dbms.logs.query.time_logging_enabled=true
dbms.logs.query.allocation_logging_enabled=true
dbms.logs.query.page_logging_enabled=true
dbms.track_query_cpu_time=true
dbms.track_query_allocation=true
```

The Neo4j Operations Manual has a section on the configuration settings you can specify to log query events to the **query.log** file.

NOTE	If you have Neo4j Desktop installed on a system, you can use it to run an application called Query Log Analyzer, written by an engineer from Neo4j. This application is a useful tool for viewing queries on your Neo4j Enterprise instance. Here is the Medium article that describes the tool and how to install it.

Long-running queries

Inspecting the log file for queries that completed in more than XX milliseconds provides historical information, but what if you suspect that a query is running too long or is hung?

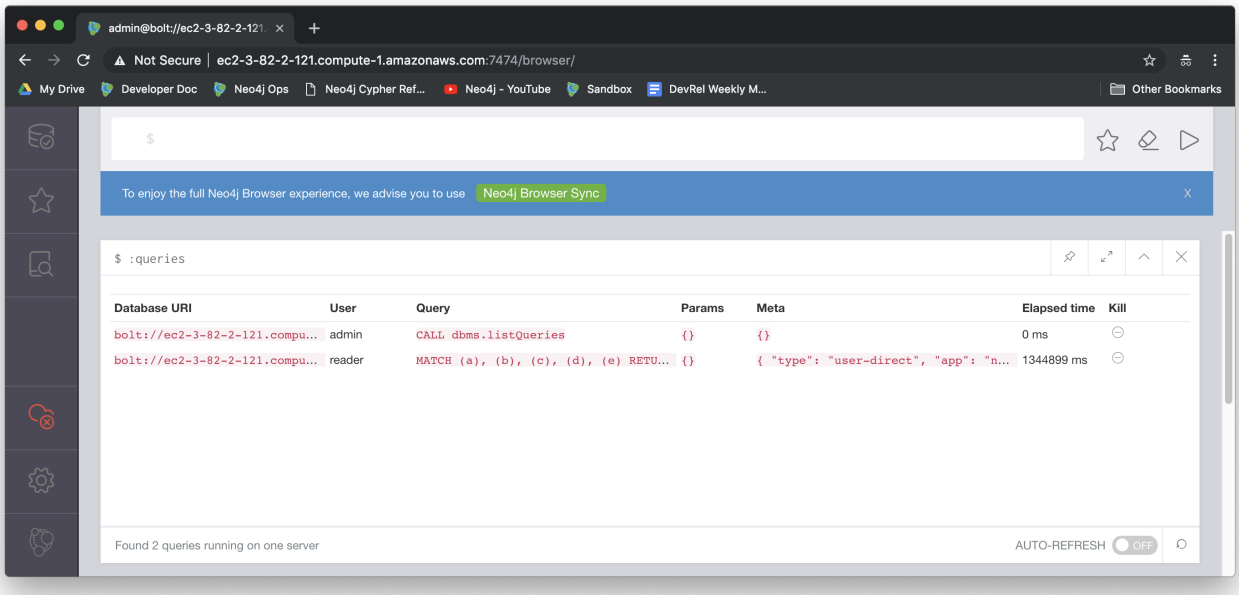
There are two reasons why a Cypher query may take a long time:

- The query returns a lot of data. The query completes execution in the graph engine, but it takes a long time to create the result stream.
  - Example: `MATCH (a)--(b)--(c)--(d)--(e)--(f) RETURN a`
- The query takes a long time to execute in the graph engine.
  - Example: `MATCH (a), (b), (c), (d), (e) RETURN count(id(a))`

If the query executes and then **returns a lot of data**, there is no way to monitor it or kill the query. If the Neo4j instance has many of these **rogue** queries running, it will slow down considerably so you should aim to limit these types of queries.

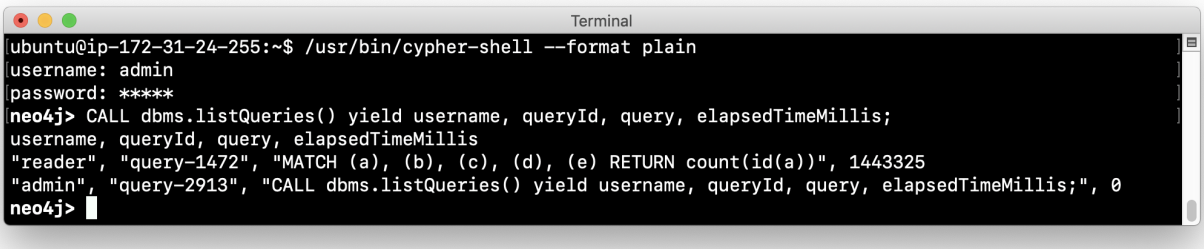
Viewing currently running queries (Neo4j Browser)

In Neo4j Browser you can use the `:queries` command to see all currently running queries:



Viewing currently running queries (cypher-shell)

In `cypher-shell` you execute `CALL dbms.listQueries() YIELD username, queryId, query, elapsedTimeMillis;`.



NOTE Due to a limitation in the database, there is a very small possibility that not all queries will be returned when you run `dbms.listQueries()`.

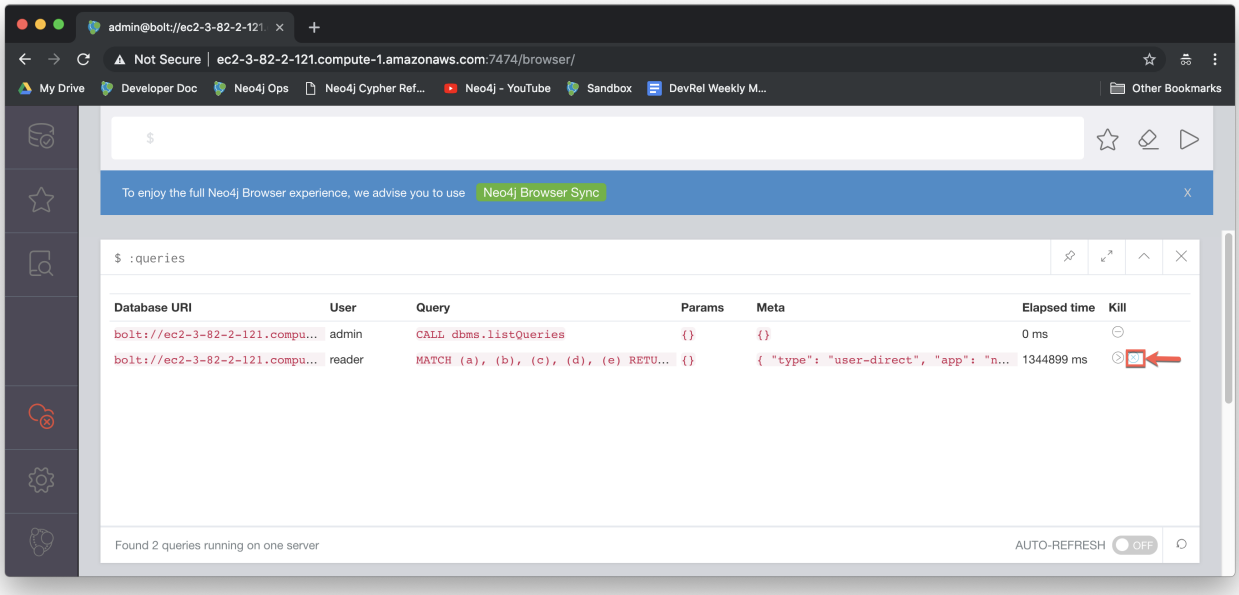
Another statement you can use to view long-running queries and any type of transaction running in the Neo4j instance is `CALL dbms.listTransactions();` which you will use in the next Exercise.

If you have the *admin* role, you can view (and kill) queries from all users. If you do not have an *admin* role, you will only be able to view your own queries.

Killing a long-running query

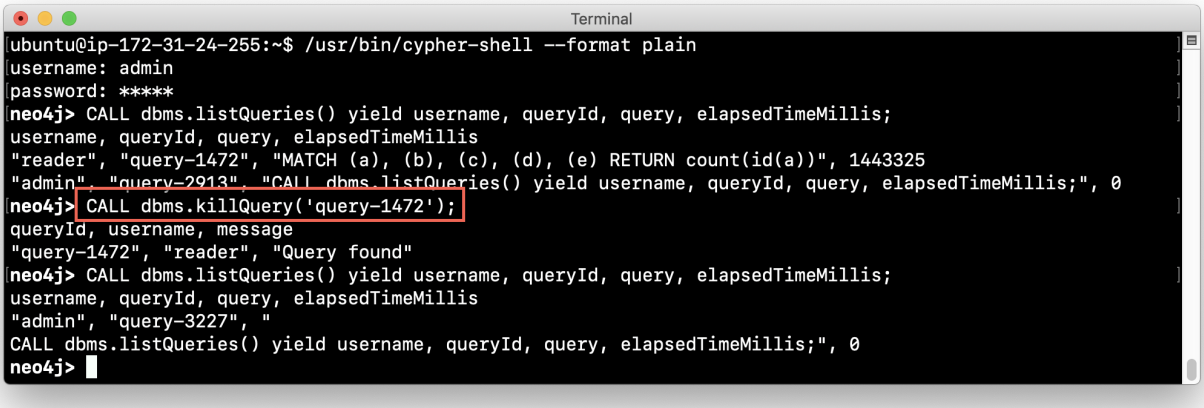
Recall that a user (or application) that issues a long-running query may not be able to stop the query. You would need to intervene and kill the query for the user.

Once you have identified the long-running query that you want to kill, in Neo4j Browser, you can kill it by double-clicking the icon in the *Kill* column.



Killing a long-running query (cypher-shell)

Alternatively, in `cypher-shell` you can execute the statement `CALL dbms.killQuery('query-id');`.



Exercise #1: Monitoring queries

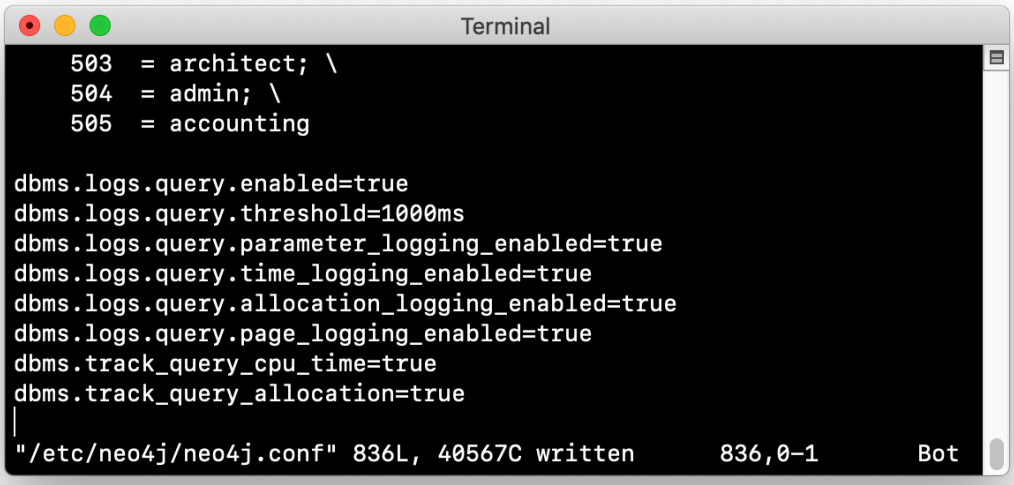
In this Exercise, you enable query logging where an event will be written to the **query.log** file for a query that took more than 1000ms to complete. Then you will monitor and detect a long-running query and kill it.

Before you begin:

1. For this Exercise, you will be using the stand-alone Neo4j instance that you configured for authentication in the previous module, *Security in Neo4j*.
2. Ensure that the database you are using is **movie3.db**.

Exercise steps:

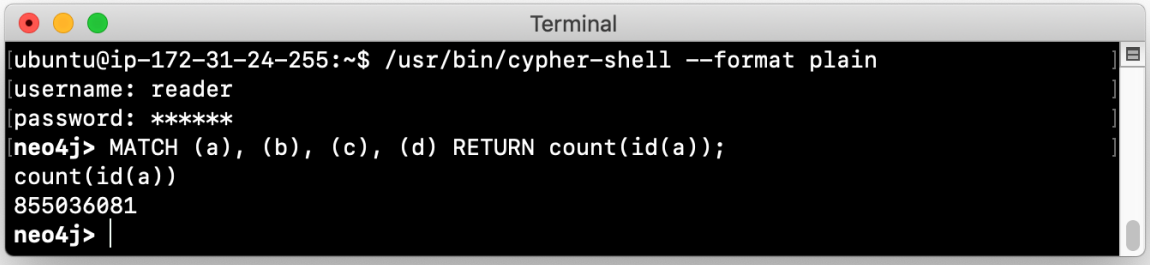
1. Modify the **neo4j.conf** file to create a log record if a query exceeds 1000 ms.



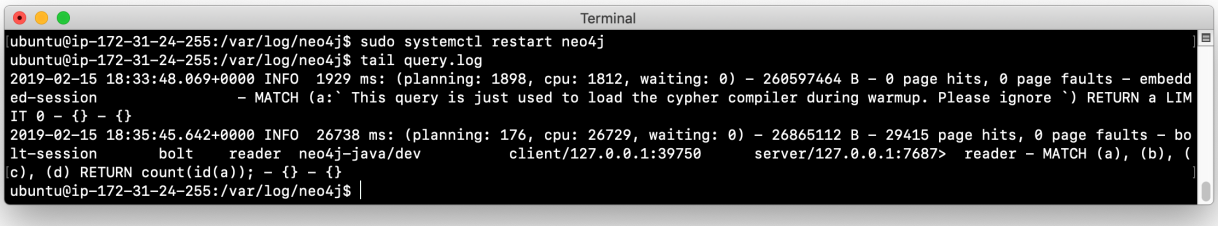
2. Start/restart the Neo4j stand-alone instance.
3. Open a new terminal window and log in to `cypher-shell` with the *reader/reader* credentials. (**Suggestion**: specify --format plain)
4. In this `cypher-shell` session, enter the following statement which will execute a query that runs for longer than 1000 ms: `MATCH (a), (b), (c), (d) RETURN count(id(a));`

NOTE	You should never run this type of query against a large database. It is only presented here for illustrative purposes to show a long-running query.

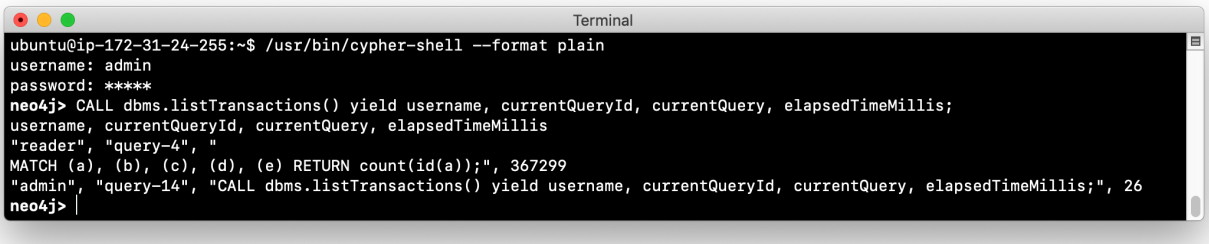
5. Wait about a minute, it should complete.



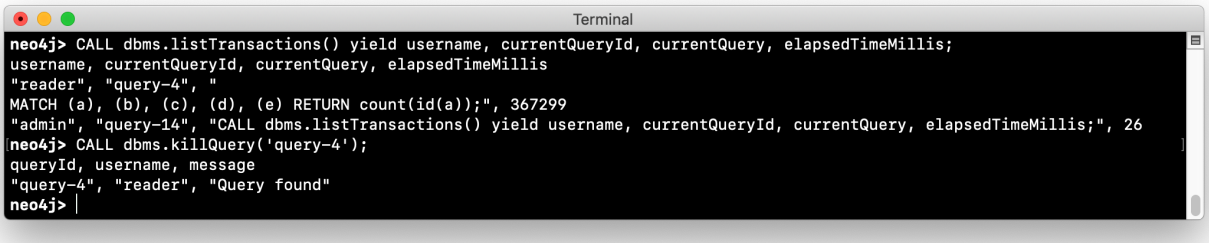
6. In the terminal window where you started the Neo4j instance, view the **query.log**. Is there a record for this query?



7. In `cypher-shell` session for *reader*, enter a query that will execute for an even longer period of time: `MATCH (a), (b), (c), (d), (e) RETURN count(id(a));`.
8. Open a new terminal window and log in to cypher-shell with the *admin/admin* credentials. (**Suggestion**: specify --format plain)
9. In this second *admin* `cypher-shell` session, execute the Cypher statement to list transactions. Do you see the query from *reader*?
10. Then execute the same statement returning the username, currentQueryId, currentQuery, and elapsedTimeMillis.



11. In the second *admin* `cypher-shell` session, execute the Cypher statement to kill the long-running query.



11. Observe in the *reader_* `cypher-shell` session that the query has been killed.

Automating monitoring of queries

Some queries against the Neo4j instance are not simply queries, but are Cypher statements that load data from CSV files. These types of Cypher statements could take a considerable amount of time to complete. One option for you to help automate the killing of long-running queries is to create a script that executes a Cypher statement such as the following:

```
CALL dbms.listQueries() YIELD query, elapsedTimeMillis, queryId, username
WHERE  NOT query CONTAINS toLower('LOAD')
AND elapsedTimeMillis > 30000
WITH query, collect(queryId) AS q
CALL dbms.killQueries(q) YIELD queryId
RETURN query, queryId;
```

This Cypher statement will retrieve all queries that are running for longer than 30000 ms that do not perform a LOAD operation and kill them. You could place this code into a script that is run at regular intervals.

Monitoring transactions

In the previous Exercise, you saw that you can query the Neo4j instance for currently running queries, as well as currently running transactions. Transactions and their successful completion are important for any production Neo4j instance. As an administrator, you must be able to confirm through monitoring and configuration settings that transactions are completing within a specified period of time.

A transaction is either a read-only transaction or a read-write transaction. Read-only transactions never block other clients as they acquire *share* locks, but can take a long period of time to execute as you saw in the previous Exercise. A read-write transaction acquires *exclusive* locks during the transaction and may be blocked by other transactions that have acquired *exclusive* locks on the same resources. In some scenarios, a deadlock could occur if one transaction is blocked and is also blocking another transaction from acquiring the exclusive locks it needs.

In a multi-user read-write transactional application, you should should configure the Neo4j instance so that a transaction will be aborted if it cannot obtain *exclusive* locks after a certain period of time. This will eliminate a deadlock situation.

In addition, you should configure an upper limit for how long a transaction can run. This will depend on your particular application, but it should be set to a value that is greater than the lock timeout value. This is called a *transaction guard* which is a good thing in a production system. In fact, you can use *transaction guard* to automatically kill queries that take longer than xx minutes to execute.

Example: Configuring transaction guard

Here is an example of the configuration settings for lock acquisition timeout and *transaction guard* where the transaction will fail if it exceeds one second or the request waits more than 10 milliseconds to acquire a write lock:

```
# transaction guard: max duration of any transaction
dbms.transaction.timeout=1s
# max time to acquire write lock
dbms.lock.acquisition.timeout=10ms
```

When a lock timeout occurs or when a transaction times out, the client will receive an error and a record is written to the **debug.log** file.

NOTE	If you set a transaction timeout without setting the lock timeout, the client session may be deadlocked and the transaction cannot be terminated. This is why it is important to set both of these properties in your Neo4j configuration.

Exercise #2: Monitoring transactions

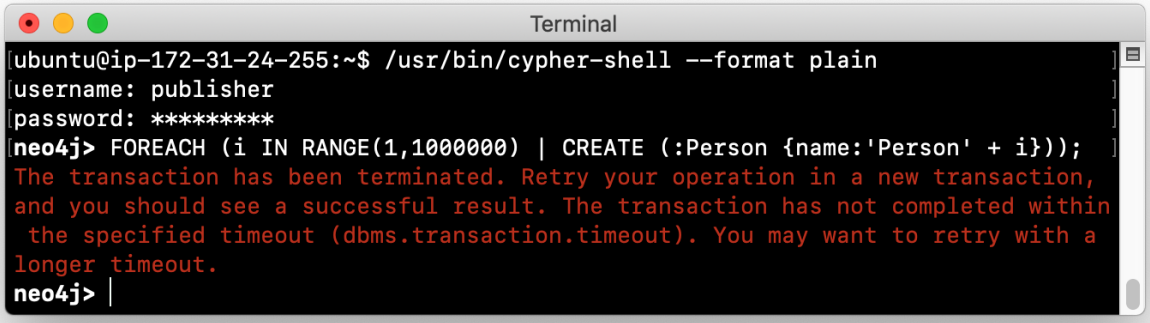
In this Exercise, you configure Neo4j to not allow transactions that take longer than one second to complete.

Before you begin:

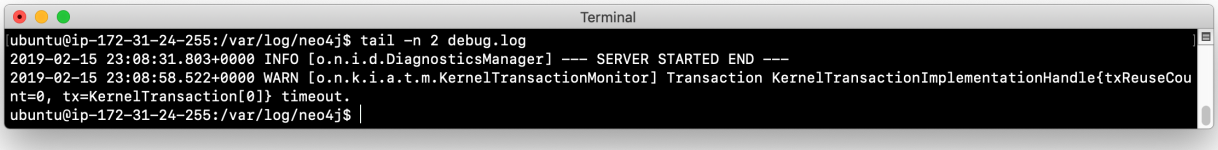
For this exercise, you will be using the stand-alone Neo4j instance that you used in the previous Exercise.

Exercise steps:

1. Modify the **neo4j.conf** file to terminate transactions where the client cannot obtain a write lock after 10 milliseconds or the transaction time exceeds 1 second.
2. Start or restart the Neo4j instance.
3. In a terminal window, log in to `cypher-shell` with the credentials *publisher/publisher*.
4. Enter this Cypher statement which will attempt to execute a write transaction to create a million *Person* nodes: `FOREACH (i IN RANGE(1,1000000) | CREATE (:Person {name:'Person' + i}));`. Do you receive an error?



5. View the record written to **debug.log**.



NOTE If you attempt to create more than a million *Person* nodes, you will run into other problems, most notably, running out of virtual memory in the Neo4j instance. You will learn about configuring virtual memory later in this module.

Monitoring locks

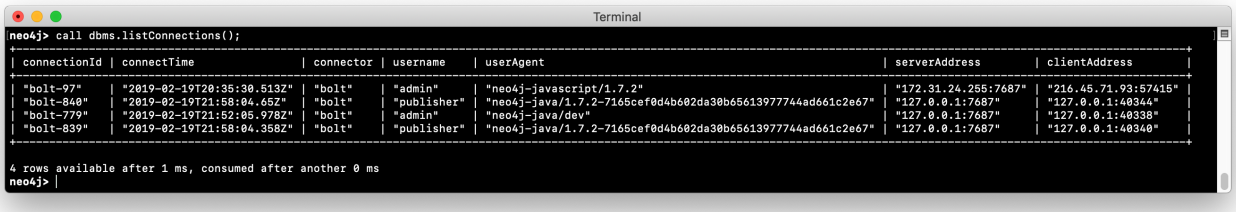
You can query the Neo4j instance's currently running transactions. If you see transactions that are running for a long time, you can further query the Neo4j instance to determine what locks each long-running query is holding. To read more about monitoring locks, see this Neo4j Support Knowledge Base article.

Monitoring connections

A Neo4j instance (stand-alone or cluster mode) uses a set of ports for inter-cluster communication and a set of ports for client communication. When you configure the Neo4j instance, you should ensure that the configured ports are available and are not blocked by a firewall.

The default ports used by a Neo4j instance are documented in the Neo4j Operations Manual. And you have learned that you can modify the port numbers used by a Neo4j instance. As you learned in a previous lesson about securing Neo4j, for a secure Neo4j application, you should not use any default port numbers.

As an administrator, you can view the current connections to a Neo4j instance from `cypher-shell` by executing the call to `dbms.listConnections()`:



The connection with the userAgent value of *neo4j-java/dev* is the cypher-shell session. Any connections that are *javascript* are from the Web interface to Neo4j Browser. The other connections are for a *java* application. You could write a query to screen for connections from certain IP addresses that are forbidden. How you identify these IP addresses will depend on your security administrator for your application.

Monitoring and terminating connections

With `dbms.listConnections()`, you can identify a connection that:

- has been connected to the Neo4j instance for too long a time period.
- is from a user that you do not want connecting to the Neo4j instance.
- is from a suspect IP address.

You terminate the connection to the Neo4j instance with a call to `dbms.killConnection()` where you can provide the connection ID or a comma-separated list of connection IDs with the format `['connectID-xx','connectID-yy']`.

Exercise #3: Monitoring connections

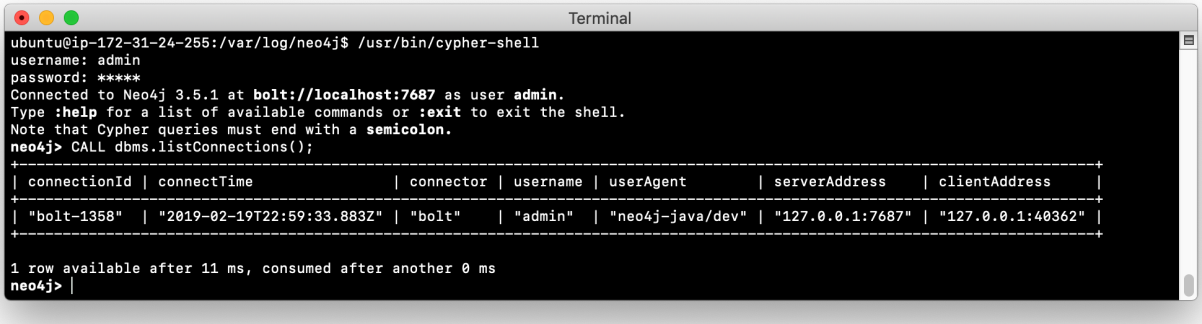
In this Exercise, you access the Neo4j instance from multiple clients and monitor the connections.

Before you begin:

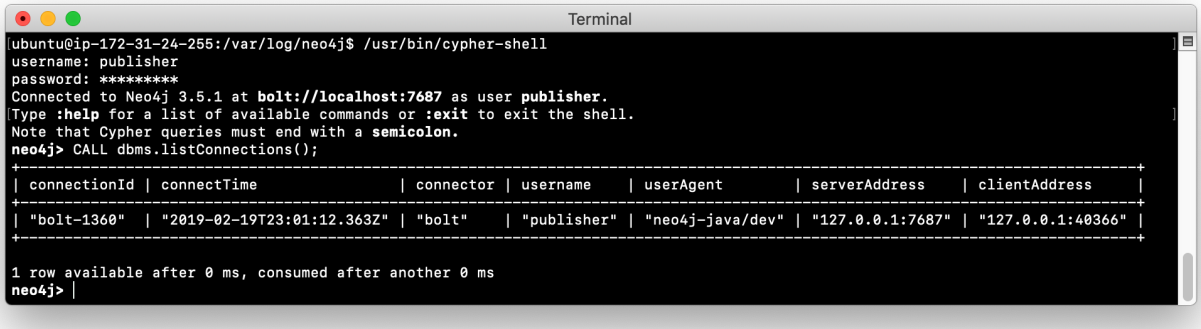
1. Make sure that you have exited out of any `cypher-shell` sessions.
2. Download the writeApp java application zip file located here. **Hint**: Enter `wget https://s3-us-west-1.amazonaws.com/data.neo4j.com/admin-neo4j/writeApp.zip`.
3. Unzip **writeApp.zip** which will create the folder **writeApp**.
4. Make sure that **write.sh** has execute permissions (`chmod +x write.sh`)

Exercise steps:

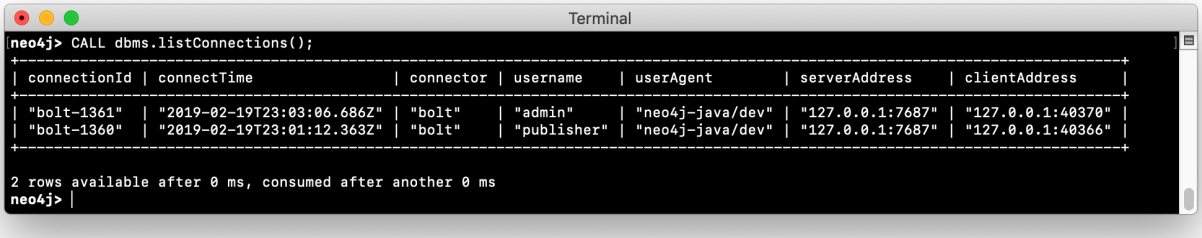
1. In a terminal window, log in to `cypher-shell` with the credentials *admin/admin*.
2. Enter the Cypher statement to list all connections to the Neo4j instance.



3. In a different terminal window, log in to `cypher-shell` with the credentials *publisher/publisher*.
4. Enter the Cypher statement to list all connections to the Neo4j instance. Do you only see the connections for your user ID?

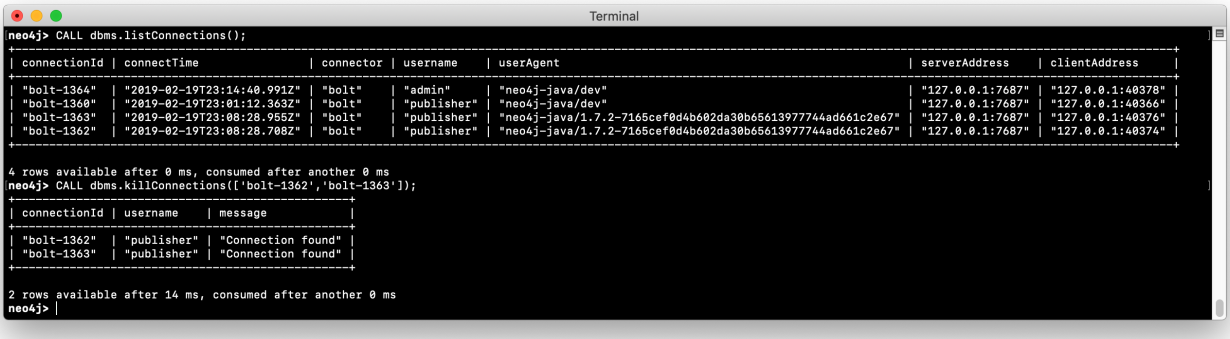


5. In the first *admin* `cypher-shell` session, enter the Cypher statement to list all connections to the Neo4j instance. Do you see all of the connections?

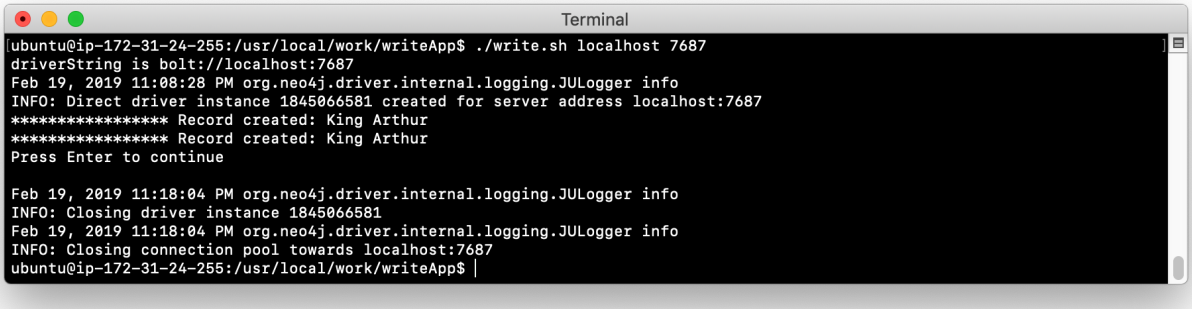


6. In a third terminal window navigate to the **writeApp** folder you created when you unzipped the java application.
7. Enter `./write.sh localhost 7687`. This java application will open a connection to the Neo4j instance and will ask you to press **Enter** to continue. Do <u>not</u> press **Enter**.
8. In the *admin* `cypher-shell` session, enter the Cypher statement to list all connections.

9. In the *admin* `cypher-shell` session, enter the Cypher statement to kill the java client connections for *publisher*.



10. In the window where the write Java application is waiting for you to press **Enter**, press the **Enter** key. You should see a message that the connection was closed.



Logging HTTP requests

You may want to monitor requests that come into the Neo4j instance from browser clients as these types of requests are typically not part of an application, but rather a user connecting to the server with their credentials.

You can set this property in **neo4j.conf** to log these requests:

```
# To enable HTTP logging, uncomment this line
dbms.logs.http.enabled=true
```

With HTTP logging enabled, you will see records for each HTTP request so you should also limit the number of log files to keep and their sizes. Part of your monitoring might be to look for certain patterns in the **http.log** file(s) and in particular, requests made from IP addresses that you may not want accessing the instance.

Exercise #4: Monitoring HTTP requests

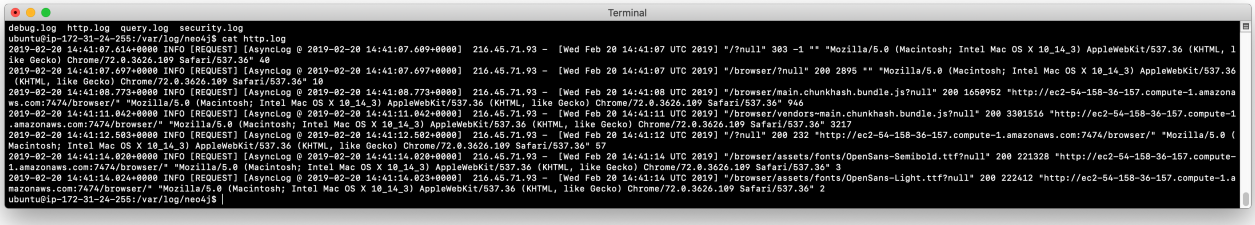
In this Exercise, you enable the Neo4j instance for logging HTTP requests and monitor them.

Before you begin:

1. Make sure that you have exited out of any cypher-shell sessions.
2. Stop the Neo4j instance.

Exercise steps:

1. In a terminal window, modify the Neo4j configuration to log HTTP requests.
2. Start the Neo4j instance.
3. In a browser, connect to the Neo4j instance using port 7474. Connect to the server as *reader/reader*.
4. View the schema of the database by executing: `CALL db.schema();`
5. View the records in the **http.log** file.



Monitoring memory usage

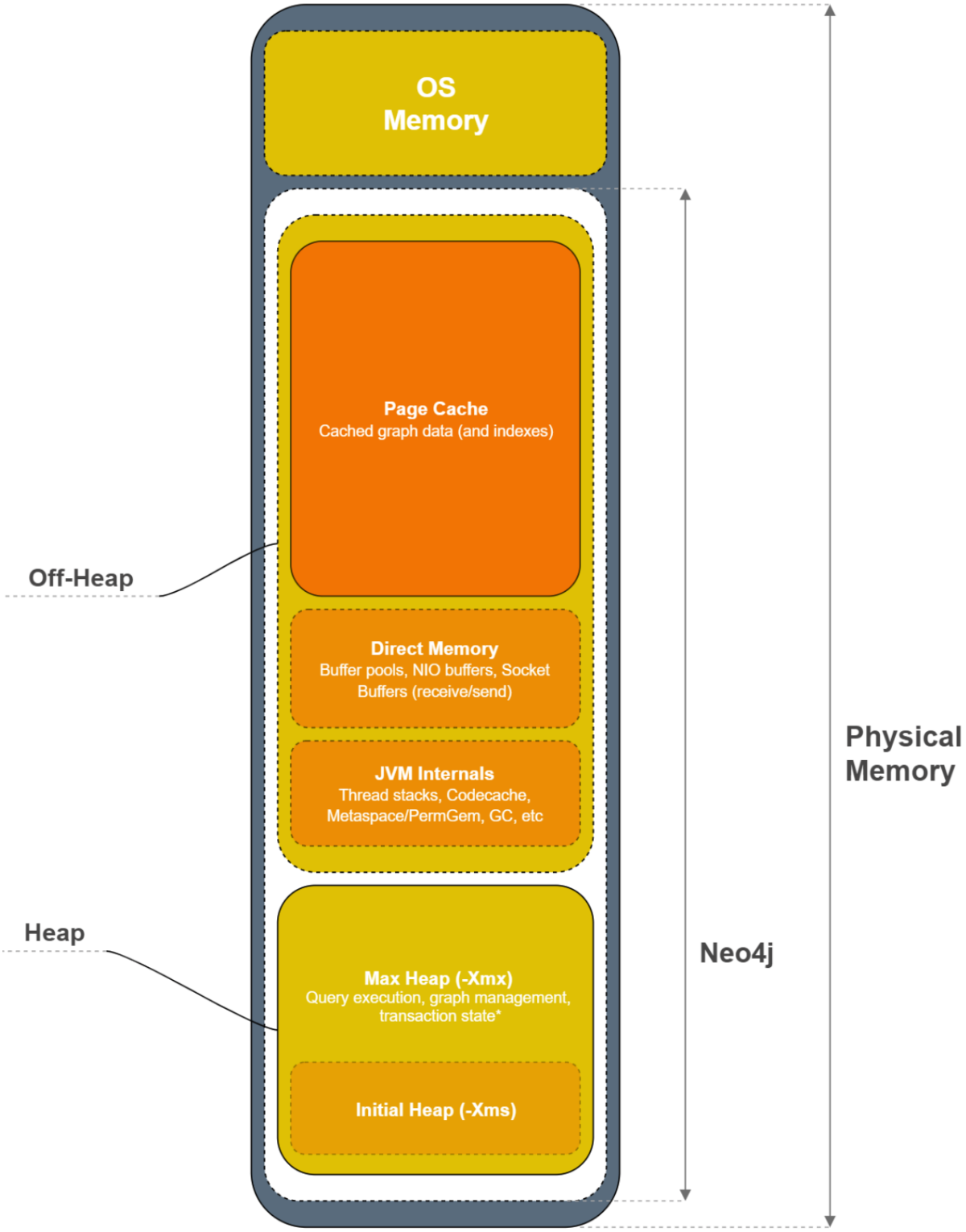
There are many properties that you can set to control how the Neo4j instance executes at runtime. The default values provided in the **neo4j.conf** file are useful for a small database with a small number of connections. In a production environment and in a Causal Cluster environment, you must make sure that the settings for the JVM are the best ones for your particular application.

This training does not teach about performance tuning, but it introduces you to how memory is used by a Neo4j instance and how you can perform basic monitoring of memory usage.

In a JVM, memory is consumed by a number of internal resources:

JVM Memory Usage	**Description**
Heap	The heap is where your Class instantiations or “Objects” are stored.
Thread stacks	Each thread has its own call stack. The stack stores primitive local variables and object references along with the call stack (list of method invocations) itself. The stack is cleaned up as stack frames move out of context so there is no GC performed here.
Metaspace	Metaspace stores the Class definitions of your Objects, and some other metadata.
Code cache	The JIT compiler stores native code it generates in the code cache to improve performance by reusing it.
Garbage Collection	In order for the GC to know which objects are eligible for collection, it needs to keep track of the object graphs. So this is one part of the memory lost to this internal bookkeeping.
Buffer Pools	Many libraries and frameworks allocate buffers outside of the heap to improve performance. These buffer pools can be used to share memory between Java code and native code, or map regions of a file into memory.

Memory consumption of a Neo4j instance



A Neo4j instance consumes memory as follows:

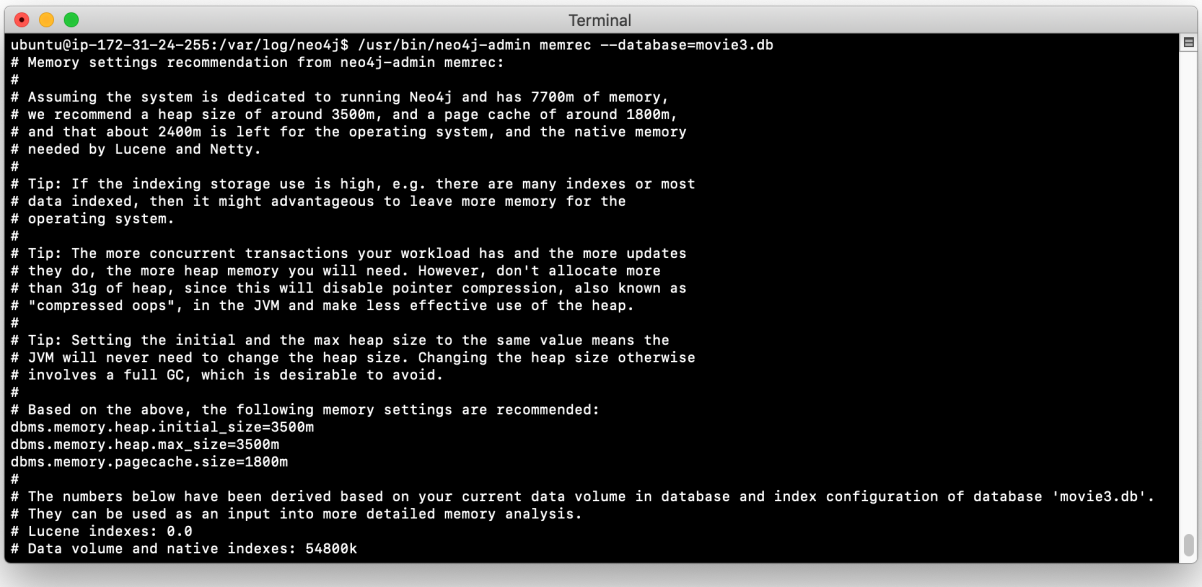
Neo4j Instance Memory Usage	Description
Heap	The JVM has a heap that is the runtime data area from which memory for all class instances and arrays are allocated. Heap storage for objects is reclaimed by an automatic storage management system (known as a garbage collector or GC).
Off-heap	Off-heap refers to objects that are managed by EHCache, but stored outside the heap (and also not subject to GC). As the off-heap store continues to be managed in memory, it is slightly slower than the on-heap store, but still faster than the disk store.
Page cache	The page cache lives off-heap and is used to cache the Neo4j data (and native indexes). The caching of graph data and indexes into memory will help avoid costly disk access and result in optimal performance.

Heap allocation is where the runtime data resides including query execution, graph management, and transaction state.

Initial memory settings for a database

The amount of memory the Neo4j instance will need may change over time and will depend on the growth of the database, as well as the number and types of queries against the database.

Initially, you can obtain a recommendation for property settings related to memory from information in the database using the `memrec` command of `neo4j-admin`:



This tool provides recommended memory settings based upon information in your database and also information about available memory on your system.

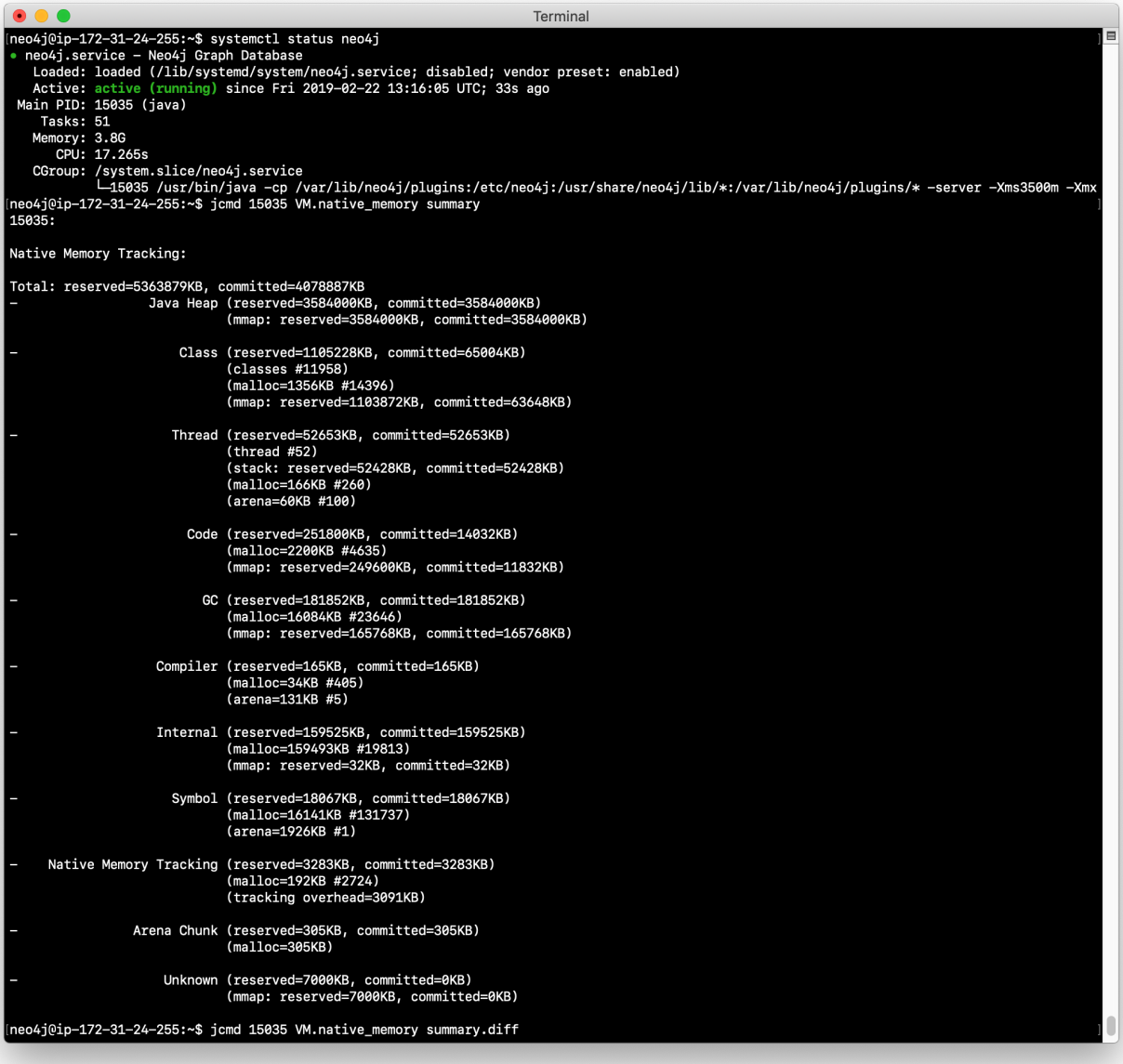
Monitoring memory consumption

If you suspect that there is a memory issue with your Neo4j instance, you should temporarily turn on GC logging in the Neo4j configuration: `dbms.logs.gc.enabled=true`. In addition, records will be written to **debug.log** if an out of memory event occurs in the Neo4j instance. When trying to resolve out of memory issues with your application, you should work with Neo4j Technical Support to determine the cause and solution for the problem.

Using jcmd to obtain summary information

One way that you can monitor memory usage for a running Neo4j instance is with the `jcmd` utility which is described in this Neo4j KB article. To monitor memory usage with this utility, you must set `dbms.jvm.additional=-XX:NativeMemoryTracking=detail` in your Neo4j configuration.

Here is an example of a `jcmd` execution to get summary information about memory usage on the system:



Using jcmd in practice

If you suspect that certain parts of the application or a transaction is consuming too much memory, you can run `jcmd` to get a baseline, and then run it again to compare the differences in memory consumption as follows:

```
jcmd <PID for Neo4j instance> VM.native_memory baseline
// wait for some time during transaction
jcmd <PID for Neo4j instance> VM.native_memory summary.diff
```

NOTE	In order to use `jcmd` for a Neo4j instance, you must ensure that the instance is started with the `dbms.jvm.additional` property set and you must run it as the user *neo4j*. **Hint**: `sudo su - neo4j`.

Refer to the Neo4j Operations Manual for guidance about configuring memory, indexes, etc. for the Neo4j instance. In a production environment, you should work with Neo4j Technical Support to ensure that you are monitoring memory usage and have the appropriate settings. The *Performance* section of the documentation has guidelines that you should consider when configuring your Neo4j instance that are beyond the scope of this training.

Exercise #5: Monitoring a memory issue

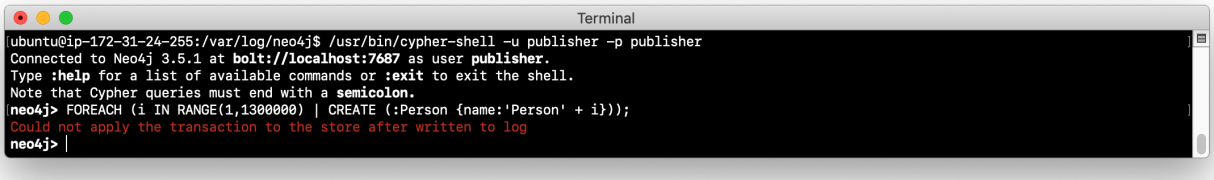
In this Exercise, you will execute a query that exhausts memory, then you will configure memory settings for the Neo4j instance and execute the query again.

Before you begin:

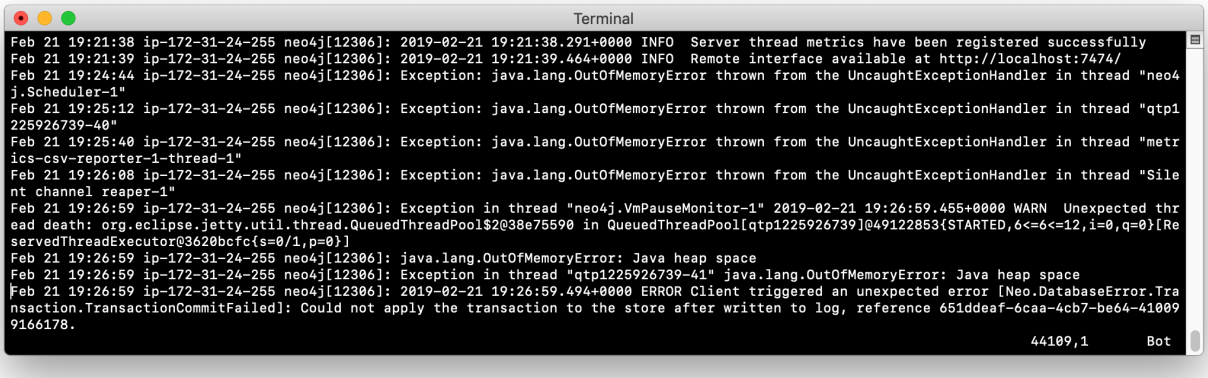
1. Make sure that you have exited out of any `cypher-shell` sessions.
2. Stop the Neo4j instance.
3. Modify the Neo4j configuration to not time out if a query takes a long time to execute. Simply comment out the settings you set previously in Exercise 2.

Exercise steps:

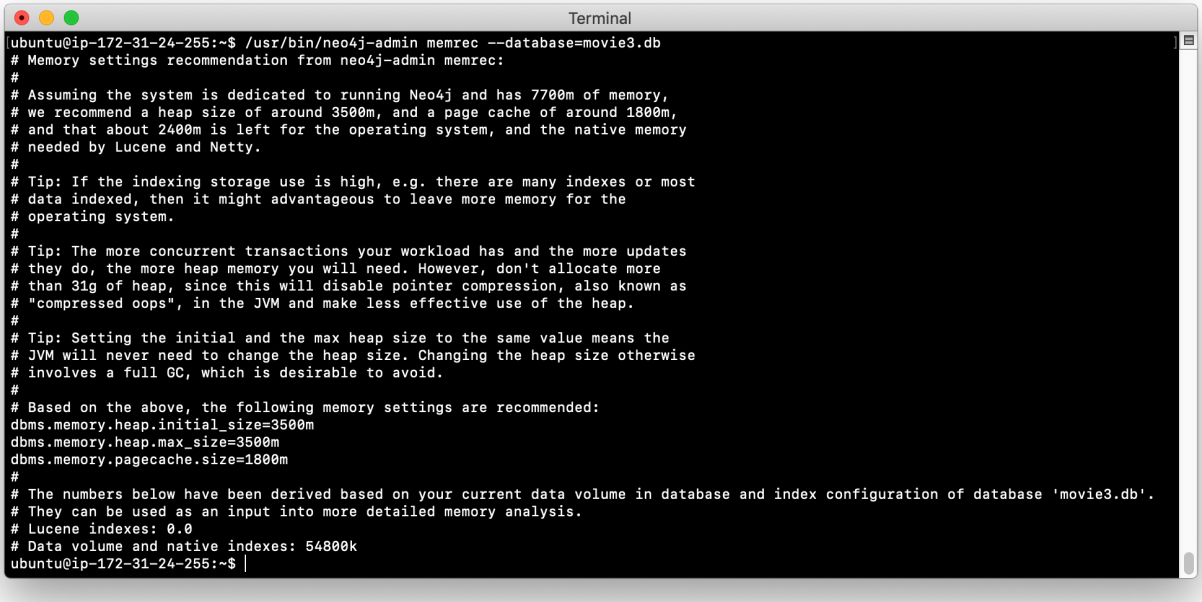
1. Start the Neo4j instance.
2. In `cypher-shell`, connect to the Neo4j instance as *publisher/publisher*.
3. Enter the following Cypher statement that will attempt to create 1.3 million *Person* nodes: `FOREACH (i IN RANGE(1,1300000) | CREATE (:Person {name:'Person' + i}));`.
4. Wait a few minutes. Eventually, you should receive an error.



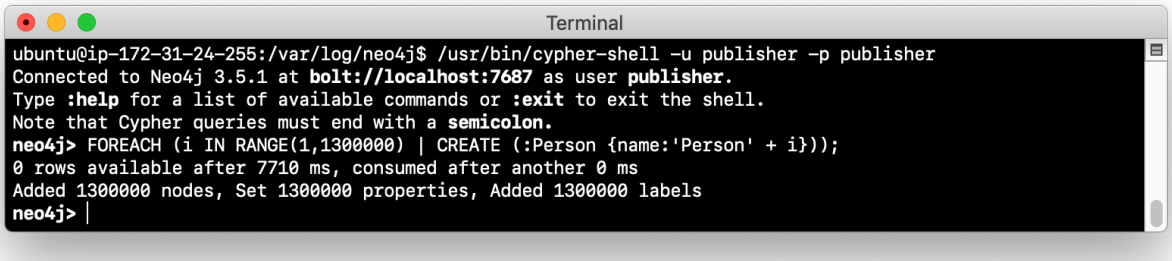
5. View the the Neo4j log **Hint**: `journalctl -e -u neo4j` on Debian. It should also have an error logged as well as an error in **debug.log**.



6. Exit out of `cypher shell`.
7. Stop the Neo4j instance. It may take a few minutes to stop the Neo4j instance as it is cleaning up the transaction log.
8. Execute the command to display the memory requirements for your system specifying the current database which is **movie3.db**.



9. If we want to add 1.3 million nodes to this database, we need to adjust the memory requirements to be at a minimum what we see from `memrec`. In **neo4j.conf**, modify `dbms.memory.heap.initial_size`, `dbms.memory.heap.max_size` , and `dbms.memory.pagecache.size` values to reflect what you see from `memrec`. Make these changes in **neo4j.conf**.
10. Restart the Neo4j instance. This may take a few minutes because the Neo4j instance is cleaning up the transaction log from the previous failed transaction.
11. Log in to `cypher-shell` as *publisher/publisher* and try the Cypher statement again that creates 1.3 million nodes.



Exercise 5: Taking it further:

Perform the above steps while using `jcmd` to monitor memory consumption.

In your production application, you must work with developers and users of the application to understand the size of the transactions. You may need to temporarily set the heap and pagecache sizes higher during a special operation. In most cases, you will set these properties to a value that will be sufficient for all transactions. You must work with Neo4j Technical Support if you run into problems with running out of memory or even with starting the Neo4j instance. If the heap and pagecache sizes are too large, the Neo4j instance will not start.

Managing log files

As an administrator, you will configure the Neo4j instance to log at the appropriate levels. In most production environments, you will archive log files so that they may be viewed at a later time as part of an auditing process or to troubleshoot a problem. Each type of log file (if configured to use) should have its maximum size defined, as well as the number of log files to keep.

```
# Number of HTTP logs to keep.
#dbms.logs.http.rotation.keep_number=5

# Size of each HTTP log that is kept.
#dbms.logs.http.rotation.size=20m

# Number of query logs to keep.
#dbms.logs.query.rotation.keep_number=5

# Size of each query log that is kept.
#dbms.logs.query.rotation.size=20m

# Number of GC logs to keep.
#dbms.logs.gc.rotation.keep_number=5

# Size of each GC log that is kept.
#dbms.logs.gc.rotation.size=20m

# Size threshold for rotation of the debug log. If set to zero then no rotation will
occur. Accepts a binary suffix "k",
# "m" or "g".
#dbms.logs.debug.rotation.size=20m

# Maximum number of history files for the internal log.
#dbms.logs.debug.rotation.keep_number=7

# Threshold for rotation of the security log.
#dbms.logs.security.rotation.size=20m

# Minimum time interval after last rotation of the security log before it may be
rotated again.
#dbms.logs.security.rotation.delay=300s

# Maximum number of history files for the security log.
#dbms.logs.security.rotation.keep_number=7
```

Collecting metrics

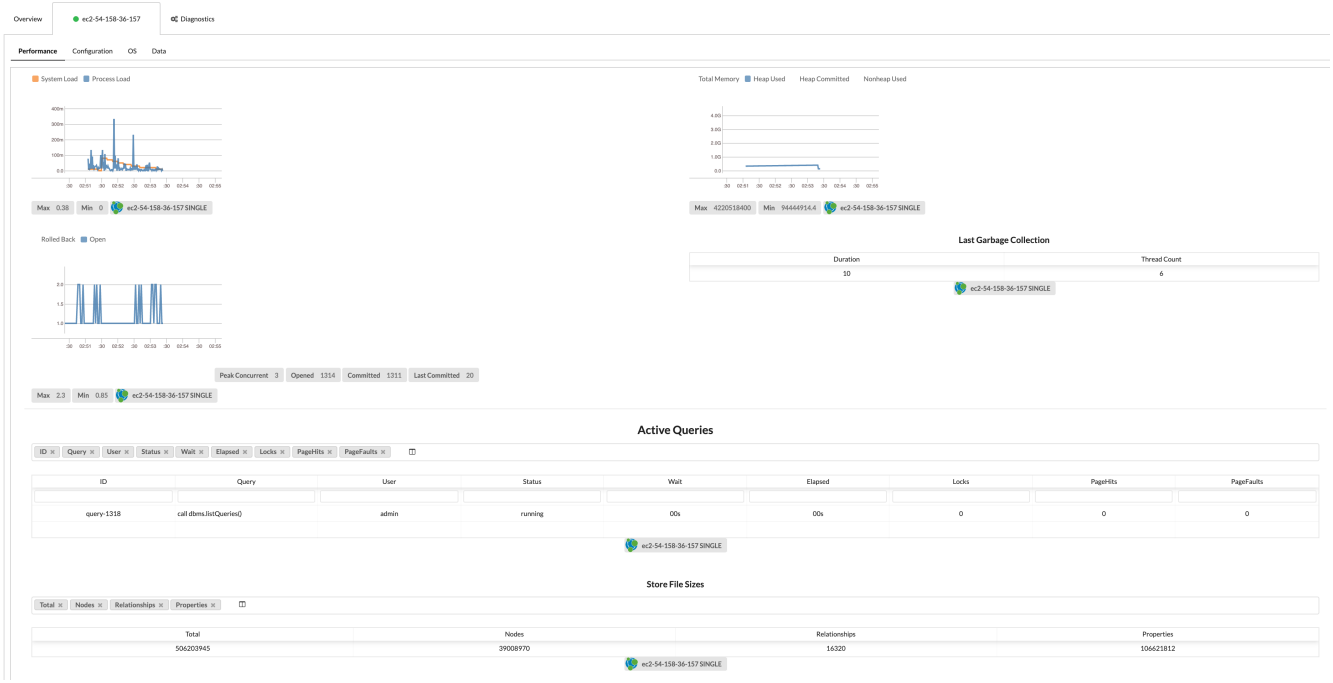
The Neo4j instance automatically collects metrics in the default location for metrics (for example, on Debian, all metrics are placed in **/var/lib/neo4j/metrics**). If for some reason, you do not want metrics collected, you can disable them by setting `metrics.enabled=false` in the Neo4j configuration.

Metrics are collected in CSV format by default on disk, but Neo4j supports Graphite and Prometheus as protocols to publish those metrics to remote services. A typical way that customers set this up is to configure Prometheus and then have some external service pull the metrics as needed from Neo4j. In this way, you don't need to set anything up around CSV metrics, but you can readily integrate Neo4j with any other monitoring application that can utilize Prometheus or Graphite. Examples of those include Stackdriver, Grafana, and DataDog. You typically set up these remote services for 24/7 monitoring and alerting.

Your options for collecting and viewing metrics are described in the Neo4j Operations Manual which include:

- Publishing to an endpoint using the Graphite protocol.
- Publishing to an endpoint using the Prometheus protocol.
- Querying the Neo4j instance using `dbms.queryJMX`.

Halin has been developed for querying the Neo4j instance. Here are a couple of screen shots when using *Halin* for viewing metrics:



In most cases for a production environment, you will set up a remote service for monitoring and alerting, but then you may also execute ad hoc queries with a tool such as *Halin*.

Using JMX queries

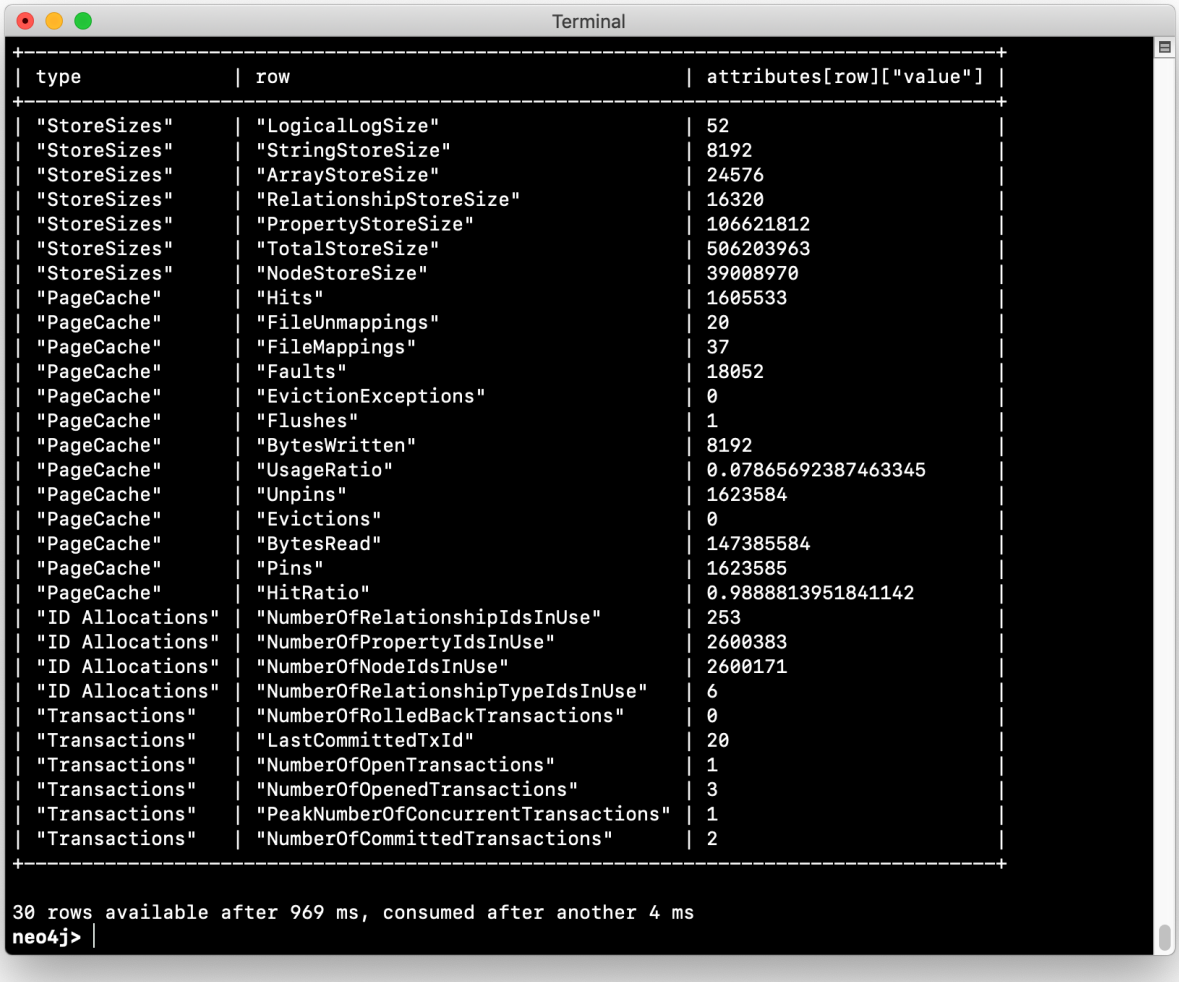
A Neo4j instance can be monitored with Java Management Extensions (JMX). JMX is a low-level mechanism for monitoring the Neo4j instance. However, a best practice is not to use it for remote monitoring as it is a security vulnerability. In addition, running a tool such as `jconsole` that uses JMX can use production system resources which is also not recommended.

Neo4j supports the use of JMX in Cypher queries. This is something that is safe to do remotely and does not consume resources locally.

For example, here is a rather long Cypher statement that retrieves the same information that you would expect to see when you run the `:sysinfo` command in Neo4j Browser:

```
CALL dbms.queryJmx("org.neo4j:instance=kernel#0,name=Store file sizes") YIELD
attributes
       WITH  keys(attributes) AS k , attributes
       UNWIND k AS row
       RETURN "StoreSizes" AS type,row,attributes[row]["value"]

UNION ALL

CALL dbms.queryJmx("org.neo4j:instance=kernel#0,name=Page cache") YIELD attributes
       WITH  keys(attributes) AS k , attributes
       UNWIND k AS row
       RETURN "PageCache" AS type,row,attributes[row]["value"]

UNION ALL

CALL dbms.queryJmx("org.neo4j:instance=kernel#0,name=Primitive count") YIELD
attributes
       WITH  keys(attributes) AS k , attributes
       UNWIND k AS row
       RETURN "ID Allocations" AS type,row,attributes[row]["value"]

UNION ALL

CALL dbms.queryJmx("org.neo4j:instance=kernel#0,name=Transactions") YIELD attributes
       WITH  keys(attributes) AS k , attributes
       UNWIND k AS row
       RETURN "Transactions" AS type,row,attributes[row]["value"]

UNION ALL

CALL dbms.queryJmx("org.neo4j:instance=kernel#0,name=High Availability") YIELD
attributes
       WITH  keys(attributes) AS k , attributes
       UNWIND k AS row
       RETURN "High Availability" AS type,row,attributes[row]["value"]

UNION ALL

CALL dbms.queryJmx("org.neo4j:instance=kernel#0,name=Causal Clustering") YIELD
attributes
       WITH  keys(attributes) AS k , attributes
       UNWIND k AS row
       RETURN "Causal Cluster" AS type,row,attributes[row]["value"];
```

Here is the result of executing this Cypher statement:



Exercise #6: Querying with JMX

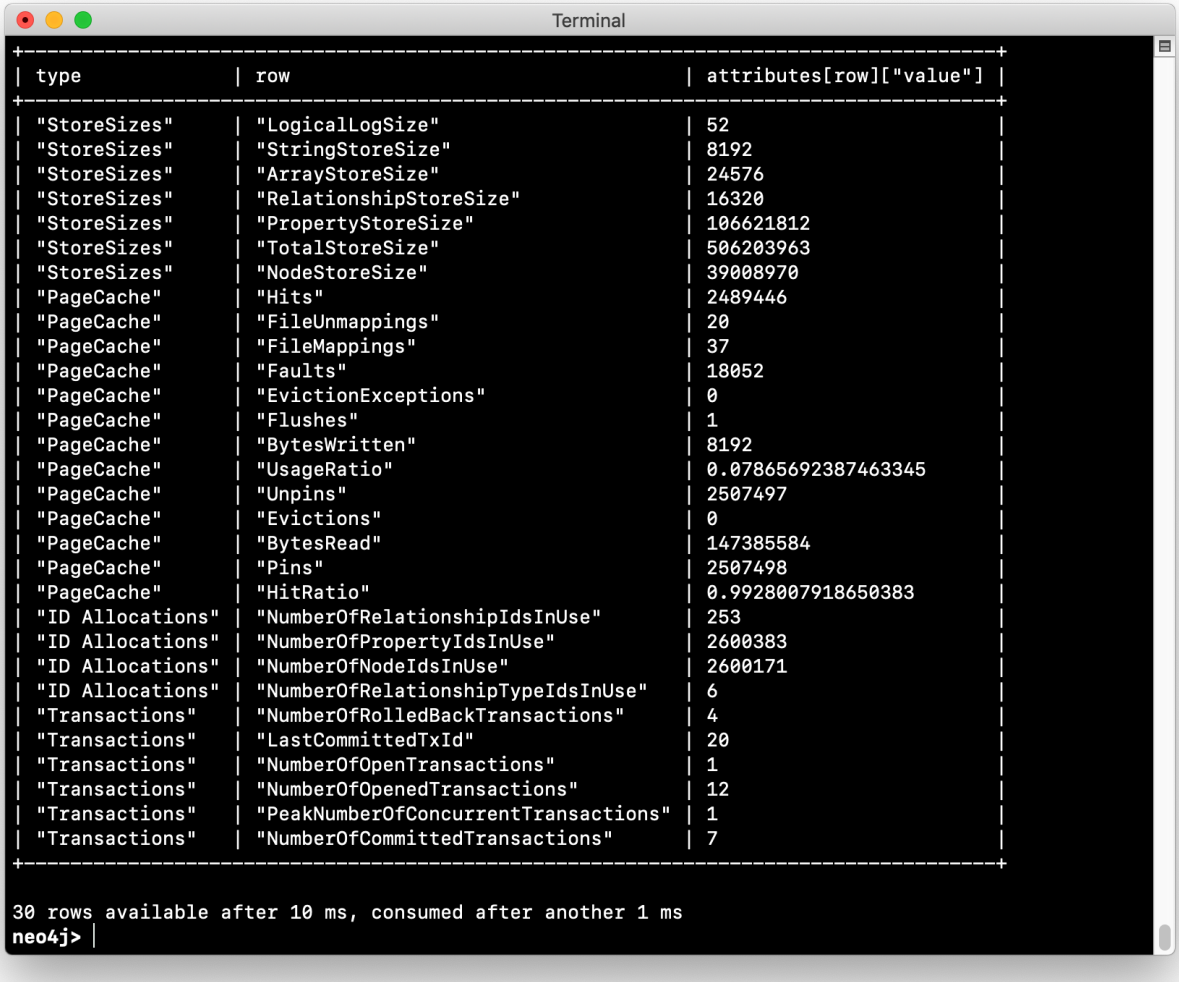
In this Exercise, you will execute a JMX query to view metrics about the Neo4j instance.

Before you begin:

1. Make sure you have a started Neo4j instance.
2. Open a terminal window.

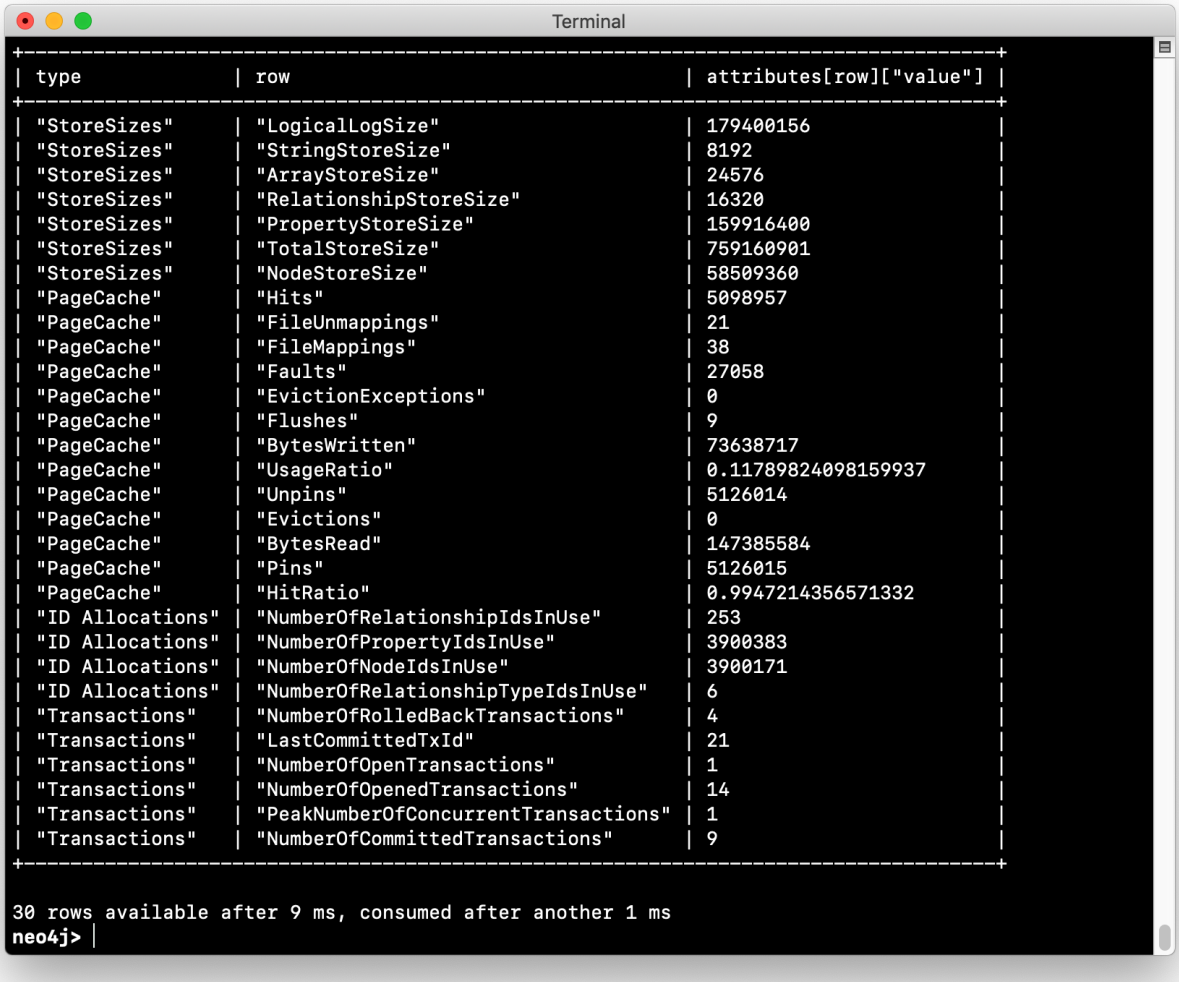
Exercise steps:

1. Log in to the Neo4j instance with `cypher-shell` using the credentials *publisher/publisher*.
2. Execute the Cypher statement shown above for querying for metrics.



3. Execute the Cypher statement for creating 1.3 million nodes: `FOREACH (i IN RANGE(1,1300000) | CREATE (:Person {name:'Person' + i}));`.

4. Execute the Cypher statement shown above for querying for metrics.



Check your understanding

Question 1

What Cypher statements can you run to determine if a query is taking too long to execute?

Select the correct answers.

- ▢ CALL dbms.getStats();
- ▢ CALL dbms.listStats();
- ▢ CALL dbms.listTransactions();
- ▢ CALL dbms.listQueries();

Question 2

What tool can you use to determine how much virtual memory you should configure for the Neo4j instance?

Select the correct answer.

- ▢ jcmd
- ▢ vmstat
- ▢ neo4j-admin memrec
- ▢ neo4j-admin analyze

Question 3

How can Neo4j metrics be used?

Select the correct answers.

- ▢ Placed in CSV files for tools to use.
- ▢ Published to an endpoint using the Graphite protocol.
- ▢ Published to an endpoint using the Prometheus protocol.
- ▢ Queried using dbms.queryJMX().

Summary

You should now be able to:

- Describe the categories of monitoring and measurement you can perform with Neo4j.
- Monitor:
  - queries
  - transactions
  - connections
  - memory usage
- Manage log files.
- Manage the collection of Neo4j metrics.
- Use JMX queries.