

Monitoring Neo4j

Table of Contents

About this module	1
Monitoring in Neo4j	2
Monitoring log files	2
Collecting metrics	2
Monitoring queries	3
Configuring query logging	3
Long-running queries	4
Viewing currently running queries (Neo4j Browser)	4
Viewing currently running queries (cypher-shell)	5
Killing a long-running query	5
Killing a long-running query (cypher-shell)	6
Exercise #1: Monitoring queries	7
Automating monitoring of queries	10
Monitoring transactions	11
Example: Configuring transaction guard	11
Exercise #2: Monitoring transactions	12
Monitoring locks	13
Monitoring connections	14
Monitoring and terminating connections	14
Exercise #3: Monitoring connections	15
Logging HTTP requests	18
Exercise #4: Monitoring HTTP requests	19
Monitoring memory usage	20
Memory consumption of a Neo4j instance	21
Initial memory settings for a database	22
Monitoring memory consumption	23
Using jccmd to obtain summary information	23
Using jccmd in practice	24
Exercise #5: Monitoring a memory issue	25
Managing log files	28
Collecting metrics	29
Using JMX queries	30
Exercise #6: Querying with JMX	33
Check your understanding	35
Question 1	35
Question 2	35
Question 3	35
Summary	36

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 used 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:

The screenshot shows the Neo4j Browser interface with the following details:

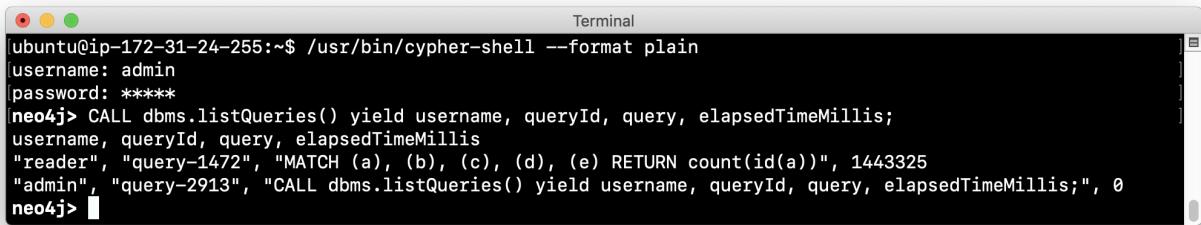
- URL:** admin@bolt://ec2-3-82-2-121.x.
- Toolbar:** Includes icons for My Drive, Developer Doc, Neo4j Ops, Neo4j Cypher Ref..., Neo4j - YouTube, Sandbox, DevRel Weekly M..., and Other Bookmarks.
- Header:** Shows the current session as "admin@bolt://ec2-3-82-2-121.compute-1.amazonaws.com:7474/browser/".
- Left Sidebar:** Contains icons for Home, Favorites, Search, and Settings.
- Central Area:**
 - A message bar says "To enjoy the full Neo4j Browser experience, we advise you to use Neo4j Browser Sync".
 - A search bar with placeholder "\$:queries".
 - A table listing currently running queries:

Database URI	User	Query	Params	Meta	Elapsed time	Kill
bolt://ec2-3-82-2-121.compu...	admin	CALL dbms.listQueries	{}	{}	0 ms	⋮
bolt://ec2-3-82-2-121.compu...	reader	MATCH (a), (b), (c), (d), (e) RETU...	{}	{"type": "user-direct", "app": "n... 1344899 ms	1344899 ms	⋮

- Bottom Status:** Shows "Found 2 queries running on one server" and "AUTO-REFRESH OFF".

Viewing currently running queries (cypher-shell)

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



```
Terminal
ubuntu@ip-172-31-24-255:~$ /usr/bin/cypher-shell --format plain
username: admin
password: *****
neo4j> CALL dbms.listQueries() yield username, queryId, query, elapsedTimeMillis;
username, queryId, query, elapsedTimeMillis
"reader", "query-1472", "MATCH (a), (b), (c), (d), (e) RETURN count(id(a))", 1443325
"admin", "query-2913", "CALL dbms.listQueries() yield username, queryId, query, elapsedTimeMillis;", 0
neo4j> █
```

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()`.

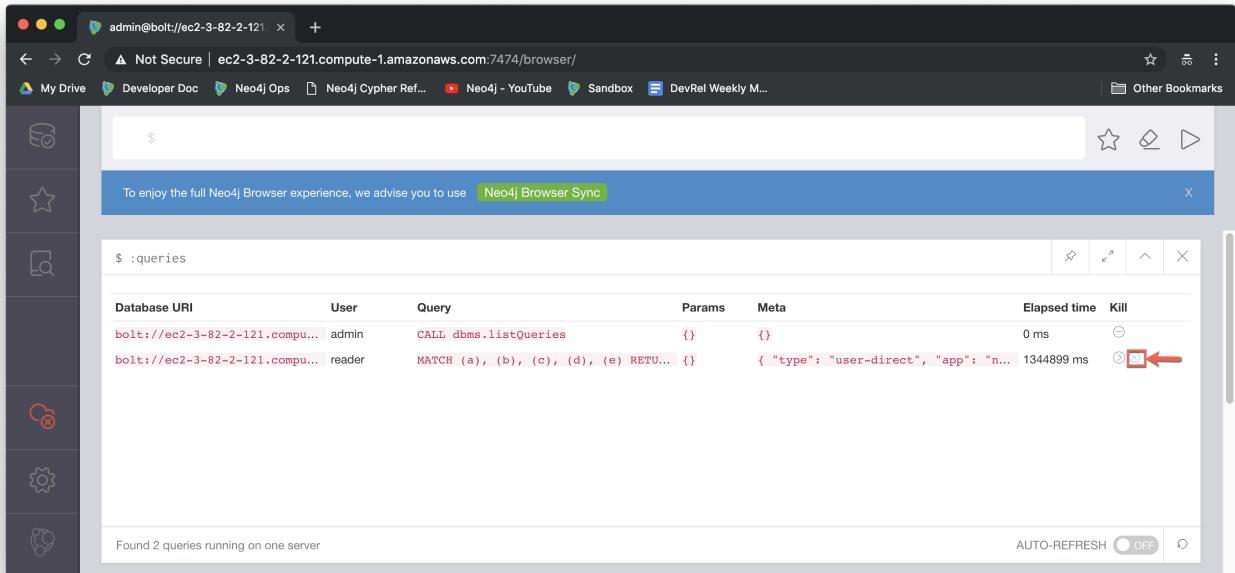
Another statement you can use to view long-running queries and any type of transaction running in the Neo4j instance is `CALL dbms.listTransactions()`; which you will use in the next Exercise.

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.



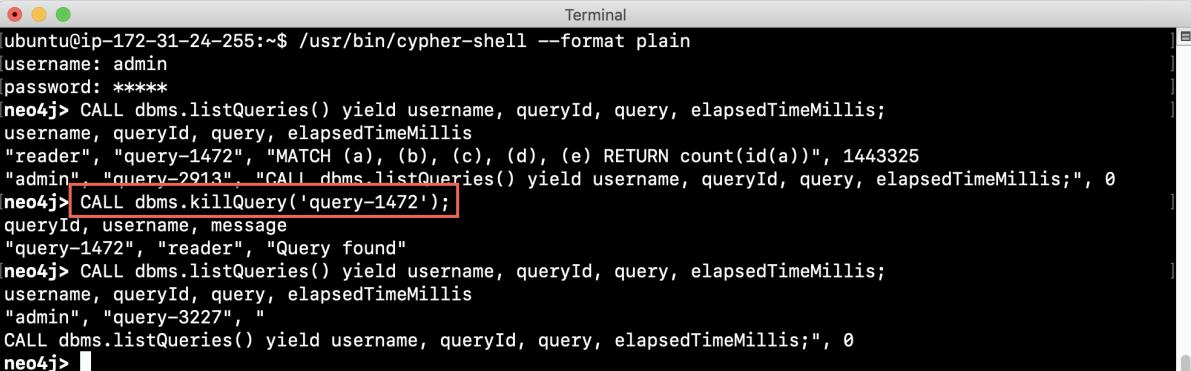
To enjoy the full Neo4j Browser experience, we advise you to use [Neo4j Browser Sync](#)

Database URI	User	Query	Params	Meta	Elapsed time	Kill
bolt://ec2-3-82-2-121.compu...	admin	CALL dbms.listQueries	{}	{}	0 ms	🔗
bolt://ec2-3-82-2-121.compu...	reader	MATCH (a), (b), (c), (d), (e) RETU...	{}	{ "type": "user-direct", "app": "n...	1344899 ms	🔗

Found 2 queries running on one server

Killing a long-running query (cypher-shell)

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



The screenshot shows a terminal window titled "Terminal" with the following session:

```
ubuntu@ip-172-31-24-255:~$ /usr/bin/cypher-shell --format plain
$username: admin
$password: *****
neo4j> CALL dbms.listQueries() yield username, queryId, query, elapsedTimeMillis;
username, queryId, query, elapsedTimeMillis
"reader", "query-1472", "MATCH (a), (b), (c), (d), (e) RETURN count(id(a))", 1443325
"admin", "query-2913", "CALL dbms.listQueries() yield username, queryId, query, elapsedTimeMillis;", 0
neo4j> CALL dbms.killQuery('query-1472');
queryId, username, message
"query-1472", "reader", "Query found"
neo4j> CALL dbms.listQueries() yield username, queryId, query, elapsedTimeMillis;
username, queryId, query, elapsedTimeMillis
"admin", "query-3227", "
CALL dbms.listQueries() yield username, queryId, query, elapsedTimeMillis;", 0
neo4j>
```

The command `CALL dbms.killQuery('query-1472');` is highlighted with a red rectangle.

Exercise #1: Monitoring queries

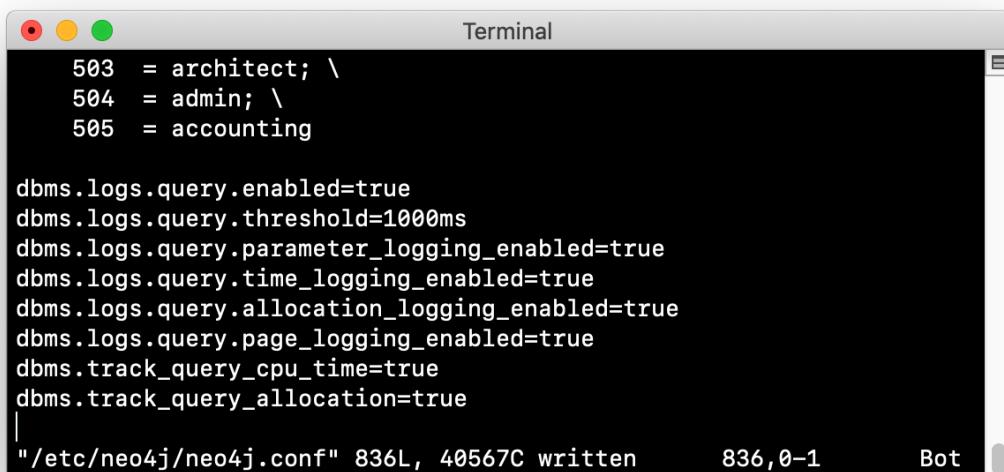
In this Exercise, you enable query logging where an event will be written to the `query.log` file for a query that took more than 1000ms to complete. Then you will monitor and detect a long-running query and kill it.

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.



The screenshot shows a Mac OS X terminal window titled "Terminal". The command entered is:

```
503 = architect; \
504 = admin; \
505 = accounting

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
```

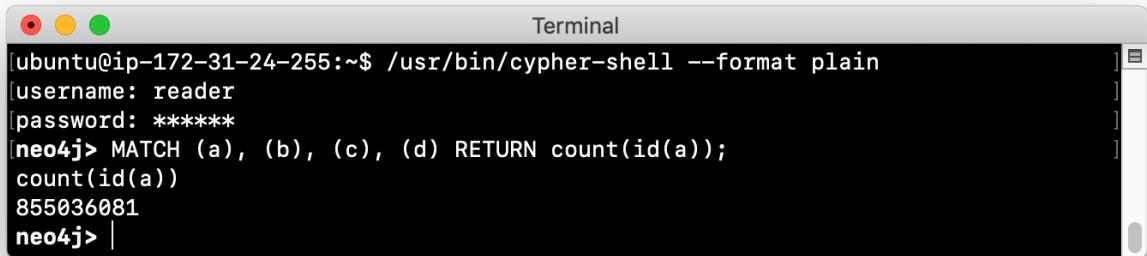
The terminal output shows the file path: "/etc/neo4j/neo4j.conf" and statistics: 836L, 40567C written, 836,0-1 Bot.

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

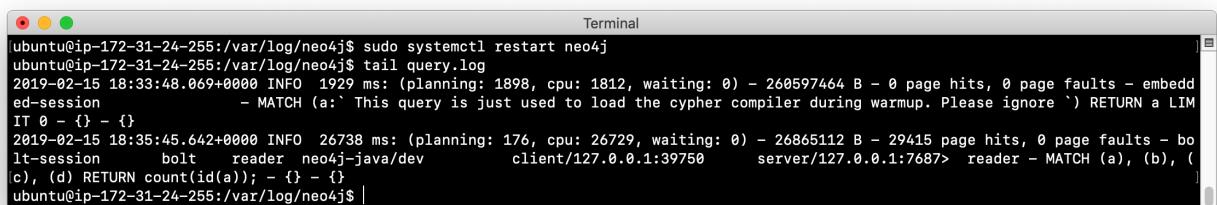
You should never run this type of query against a large database. It is only presented here for illustrative purposes to show a long-running query.

5. Wait about a minute, it should complete.



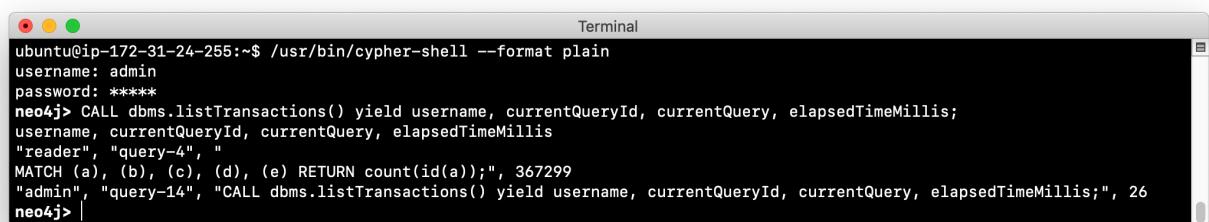
```
Terminal
ubuntu@ip-172-31-24-255:~$ /usr/bin/cypher-shell --format plain
username: reader
password: *****
neo4j> MATCH (a), (b), (c), (d) RETURN count(id(a));
count(id(a))
855036081
neo4j> |
```

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



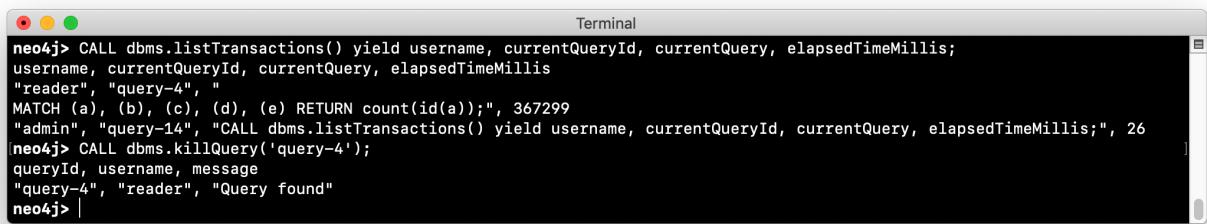
```
Terminal
ubuntu@ip-172-31-24-255:/var/log/neo4j$ sudo systemctl restart neo4j
ubuntu@ip-172-31-24-255:/var/log/neo4j$ tail query.log
2019-02-15 18:33:48.069+0000 INFO 1929 ms: (planning: 1898, cpu: 1812, waiting: 0) - 260597464 B - 0 page hits, 0 page faults - embedd
ed-session          - MATCH (a:` This query is just used to load the cypher compiler during warmup. Please ignore `) RETURN a LIM
IT 0 - {} - {}
2019-02-15 18:35:45.642+0000 INFO 26738 ms: (planning: 176, cpu: 26729, waiting: 0) - 26865112 B - 29415 page hits, 0 page faults - bo
lt-session          bolt    reader  neo4j-java/dev           client/127.0.0.1:39750      server/127.0.0.1:7687> reader - MATCH (a), (b), (
c), (d) RETURN count(id(a)); - {} - {}
ubuntu@ip-172-31-24-255:/var/log/neo4j$ |
```

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.



```
Terminal
ubuntu@ip-172-31-24-255:~$ /usr/bin/cypher-shell --format plain
username: admin
password: *****
neo4j> CALL dbms.listTransactions() yield username, currentQueryId, currentQuery, elapsedTimeMillis;
username, currentQueryId, currentQuery, elapsedTimeMillis
"reader", "query-4", "
MATCH (a), (b), (c), (d), (e) RETURN count(id(a));", 367299
"admin", "query-14", "CALL dbms.listTransactions() yield username, currentQueryId, currentQuery, elapsedTimeMillis;", 26
neo4j> |
```

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



The screenshot shows a macOS Terminal window titled "Terminal". Inside, an "admin" session of the Neo4j Cypher shell is running. The user has run several commands to identify a specific query (query-4) and then killed it using the `CALL dbms.killQuery('query-4');` command. The output shows the query ID, its owner ("reader"), and a confirmation message.

```
neo4j> CALL dbms.listTransactions() yield username, currentQueryId, currentQuery, elapsedTimeMillis;
username, currentQueryId, currentQuery, elapsedTimeMillis
"reader", "query-4", "
MATCH (a), (b), (c), (d), (e) RETURN count(id(a));", 367299
"admin", "query-14", "CALL dbms.listTransactions() yield username, currentQueryId, currentQuery, elapsedTimeMillis;", 26
neo4j> CALL dbms.killQuery('query-4');
queryId, username, message
"query-4", "reader", "Query found"
neo4j> |
```

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?

```
Terminal
ubuntu@ip-172-31-24-255:~$ /usr/bin/cypher-shell --format plain
username: publisher
password: *****
neo4j> FOREACH (i IN RANGE(1,1000000) | CREATE (:Person {name:'Person' + i}));
The transaction has been terminated. Retry your operation in a new transaction,
and you should see a successful result. The transaction has not completed within
the specified timeout (dbms.transaction.timeout). You may want to retry with a
longer timeout.
neo4j> |
```

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

```
Terminal
ubuntu@ip-172-31-24-255:/var/log/neo4j$ tail -n 2 debug.log
2019-02-15 23:08:31.802+0000 INFO [o.n.i.d.DiagnosticsManager] --- SERVER STARTED END ---
2019-02-15 23:08:58.522+0000 WARN [o.n.k.i.a.t.m.KernelTransactionMonitor] Transaction KernelTransactionImplementationHandle{txReuseCount=0, tx=KernelTransaction[0]} timeout.
ubuntu@ip-172-31-24-255:/var/log/neo4j$ |
```

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()`:



```
neo4j> call dbms.listConnections();
+-----+-----+-----+-----+-----+-----+-----+
| connectionId | connectTime | connector | username | userAgent
+-----+-----+-----+-----+-----+-----+-----+
| "bolt-97"   | "2019-02-19T22:35:39.513Z" | "bolt"    | "admin"   | "neo4j-javascript/1.7.2"
| "bolt-840"  | "2019-02-19T22:58:04.652Z" | "bolt"    | "publisher" | "neo4j-java/1.7.2-7165cef0d4b602da30b65613977744ad661c2e67"
| "bolt-779"  | "2019-02-19T22:52:05.978Z" | "bolt"    | "admin"   | "neo4j-java/dev"
| "bolt-839"  | "2019-02-19T22:58:04.358Z" | "bolt"    | "publisher" | "neo4j-java/1.7.2-7165cef0d4b602da30b65613977744ad661c2e67"
+-----+-----+-----+-----+-----+-----+
4 rows available after 1 ms, consumed after another 0 ms
neo4j>
```

The connection with the userAgent value of *neo4j-javascript* 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

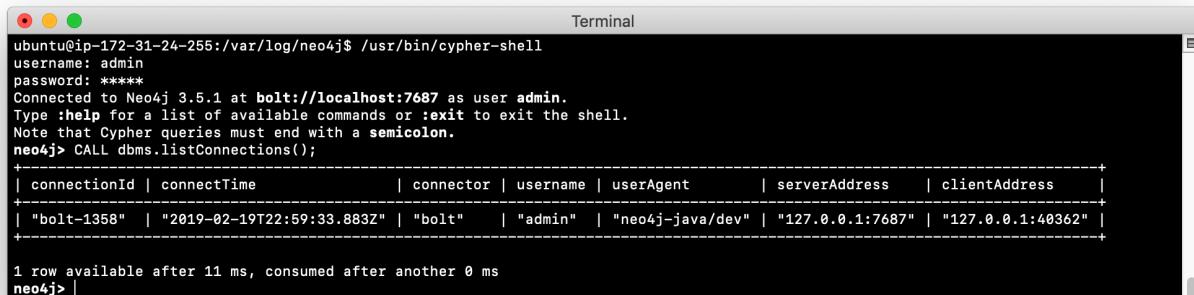
In this Exercise, you access the Neo4j instance from multiple clients and monitor the connections.

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.



```
ubuntu@ip-172-31-24-255:/var/log/neo4j$ /usr/bin/cypher-shell
username: admin
password: *****
Connected to Neo4j 3.5.1 at bolt://localhost:7687 as user admin.
Type :help for a list of available commands or :exit to exit the shell.
Note that Cypher queries must end with a semicolon.
neo4j> CALL dbms.listConnections();
+-----+
| connectionId | connectTime           | connector | username   | userAgent      | serverAddress | clientAddress |
+-----+
| "bolt-1358"  | "2019-02-19T22:59:33.883Z" | "bolt"    | "admin"    | "neo4j-java/dev" | "127.0.0.1:7687" | "127.0.0.1:40362" |
+-----+
1 row available after 11 ms, consumed after another 0 ms
neo4j> |
```

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?



```
ubuntu@ip-172-31-24-255:/var/log/neo4j$ /usr/bin/cypher-shell
username: publisher
password: *****
Connected to Neo4j 3.5.1 at bolt://localhost:7687 as user publisher.
Type :help for a list of available commands or :exit to exit the shell.
Note that Cypher queries must end with a semicolon.
neo4j> CALL dbms.listConnections();
+-----+
| connectionId | connectTime           | connector | username   | userAgent      | serverAddress | clientAddress |
+-----+
| "bolt-1360"  | "2019-02-19T23:01:12.363Z" | "bolt"    | "publisher" | "neo4j-java/dev" | "127.0.0.1:7687" | "127.0.0.1:40366" |
+-----+
1 row available after 0 ms, consumed after another 0 ms
neo4j> |
```

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?

```
neo4j> CALL dbms.listConnections();
+-----+-----+-----+-----+-----+
| connectionId | connectTime | connector | username | userAgent |
+-----+-----+-----+-----+-----+
| "bolt-1361" | "2019-02-19T23:03:06.686Z" | "bolt" | "admin" | "neo4j-java/dev" |
| "bolt-1360" | "2019-02-19T23:01:12.363Z" | "bolt" | "publisher" | "neo4j-java/dev" |
+-----+-----+-----+-----+-----+
2 rows available after 0 ms, consumed after another 0 ms
neo4j>
```

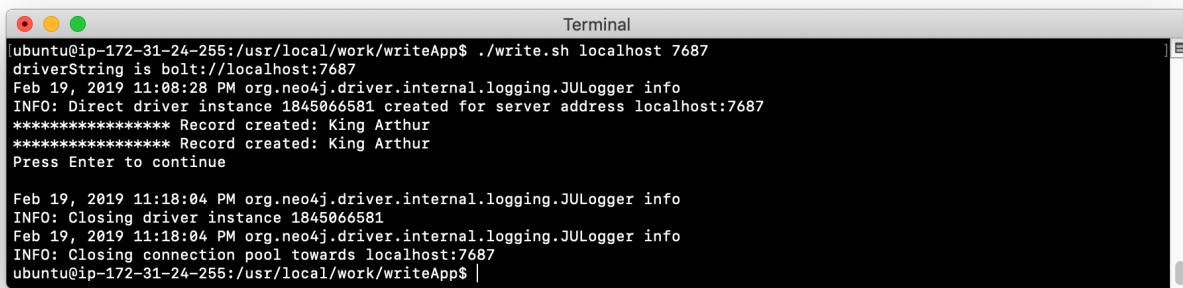
6. In a third terminal window navigate to the **writeApp** folder you created when you unzipped the java application.
7. Enter **./write.sh localhost 7687**. This java application will open a connection to the Neo4j instance and will ask you to press **Enter** to continue. Do not press **Enter**.
8.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

```
neo4j> CALL dbms.listConnections();
+-----+-----+-----+-----+-----+
| connectionId | connectTime | connector | username | userAgent |
+-----+-----+-----+-----+-----+
| "bolt-1361" | "2019-02-19T23:03:06.686Z" | "bolt" | "admin" | "neo4j-java/dev" |
| "bolt-1360" | "2019-02-19T23:01:12.363Z" | "bolt" | "publisher" | "neo4j-java/dev" |
| "bolt-1363" | "2019-02-19T23:08:28.952Z" | "bolt" | "publisher" | "neo4j-java/1.7.2-7165cef0d4b602da30b65613977744ad661c2e67" |
| "bolt-1362" | "2019-02-19T23:08:28.708Z" | "bolt" | "publisher" | "neo4j-java/1.7.2-7165cef0d4b602da30b65613977744ad661c2e67" |
+-----+-----+-----+-----+-----+
4 rows available after 1 ms, consumed after another 0 ms
neo4j>
```

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

```
neo4j> CALL dbms.listConnections();
+-----+-----+-----+-----+-----+
| connectionId | connectTime | connector | username | userAgent |
+-----+-----+-----+-----+-----+
| "bolt-1364" | "2019-02-19T23:14:40.991Z" | "bolt" | "admin" | "neo4j-java/dev" |
| "bolt-1360" | "2019-02-19T23:01:12.363Z" | "bolt" | "publisher" | "neo4j-java/dev" |
| "bolt-1363" | "2019-02-19T23:08:28.952Z" | "bolt" | "publisher" | "neo4j-java/1.7.2-7165cef0d4b602da30b65613977744ad661c2e67" |
| "bolt-1362" | "2019-02-19T23:08:28.708Z" | "bolt" | "publisher" | "neo4j-java/1.7.2-7165cef0d4b602da30b65613977744ad661c2e67" |
+-----+-----+-----+-----+-----+
4 rows available after 0 ms, consumed after another 0 ms
neo4j> CALL dbms.killConnections(['bolt-1362','bolt-1363']);
+-----+-----+
| connectionId | username | message |
+-----+-----+
| "bolt-1362" | "publisher" | "Connection found" |
| "bolt-1363" | "publisher" | "Connection found" |
+-----+-----+
2 rows available after 14 ms, consumed after another 0 ms
neo4j>
```

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.



The screenshot shows a terminal window titled "Terminal" with a black background and white text. The window contains the following log output:

```
[ubuntu@ip-172-31-24-255:/usr/local/work/writeApp$ ./write.sh localhost 7687
driverString is bolt://localhost:7687
Feb 19, 2019 11:08:28 PM org.neo4j.driver.internal.logging.JULogger info
INFO: Direct driver instance 1845066581 created for server address localhost:7687
***** Record created: King Arthur
***** Record created: King Arthur
Press Enter to continue

Feb 19, 2019 11:18:04 PM org.neo4j.driver.internal.logging.JULogger info
INFO: Closing driver instance 1845066581
Feb 19, 2019 11:18:04 PM org.neo4j.driver.internal.logging.JULogger info
INFO: Closing connection pool towards localhost:7687
INFO: Connection pool closed
ubuntu@ip-172-31-24-255:/usr/local/work/writeApp$ |
```

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

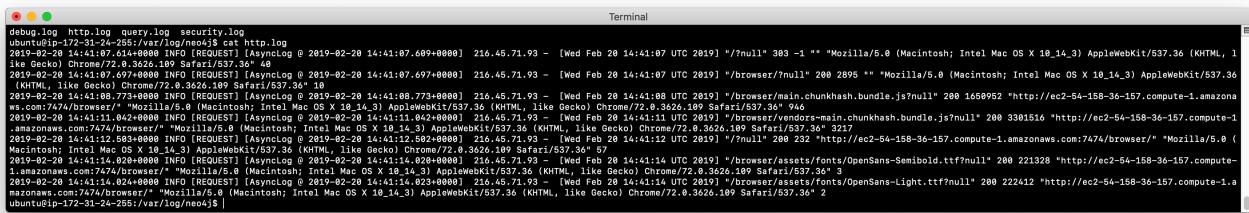
In this Exercise, you enable the Neo4j instance for logging HTTP requests and monitor them.

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.



The terminal window displays the contents of the `http.log` file. The log entries show various HTTP requests from different browsers and operating systems. Key details include the browser type (e.g., Mozilla/5.0, AppleWebKit/537.36), version (e.g., Chrome/72.0.3626.109, Safari/537.36), and the timestamp of the request (e.g., 2019-02-20 14:41:07). The log also includes information about the response status code (e.g., 200, 208) and the URL being requested (e.g., "/").

```
debug.log http.log query.log security.log
ubuntu@ip-172-31-24-255:~/log$ tail -f http.log
2019-02-20 14:41:07.614+0000 INFO [REQUEST] [AsyncLog @ 2019-02-20 14:41:07.609+0000] 216.45.71.93 - [Wed Feb 20 14:41:07 UTC 2019] "/?null" 303 -1 "" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36" 40
2019-02-20 14:41:07.697+0000 INFO [REQUEST] [AsyncLog @ 2019-02-20 14:41:07.697+0000] 216.45.71.93 - [Wed Feb 20 14:41:07 UTC 2019] "/browser/?null" 208 2895 "" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36" 40
2019-02-20 14:41:07.701+0000 INFO [REQUEST] [AsyncLog @ 2019-02-20 14:41:07.701+0000] 216.45.71.93 - [Wed Feb 20 14:41:07 UTC 2019] "/?null" 208 1650952 "http://ec2-54-158-36-157.compute-1.amazonaws.com:7474/browser/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36" 946
2019-02-20 14:41:11.042+0000 INFO [REQUEST] [AsyncLog @ 2019-02-20 14:41:11.042+0000] 216.45.71.93 - [Wed Feb 20 14:41:11 UTC 2019] "/browser/vendors-main.chunkhash.bundle.js?null" 208 3301516 "http://ec2-54-158-36-157.compute-1.amazonaws.com:7474/browser/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36" 3217
2019-02-20 14:41:14.020+0000 INFO [REQUEST] [AsyncLog @ 2019-02-20 14:41:14.020+0000] 216.45.71.93 - [Wed Feb 20 14:41:14 UTC 2019] "/?null" 208 232 "http://ec2-54-158-36-157.compute-1.amazonaws.com:7474/browser/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36" 57
2019-02-20 14:41:14.026+0000 INFO [REQUEST] [AsyncLog @ 2019-02-20 14:41:14.026+0000] 216.45.71.93 - [Wed Feb 20 14:41:14 UTC 2019] "/browser/assets/fonts/OpenSans-Semibold.ttf?null" 208 221328 "http://ec2-54-158-36-157.compute-1.amazonaws.com:7474/browser/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36" 3
2019-02-20 14:41:14.029+0000 INFO [REQUEST] [AsyncLog @ 2019-02-20 14:41:14.029+0000] 216.45.71.93 - [Wed Feb 20 14:41:14 UTC 2019] "/browser/assets/fonts/OpenSans-Light.ttf?null" 208 222412 "http://ec2-54-158-36-157.compute-1.amazonaws.com:7474/browser/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36" 2
ubuntu@ip-172-31-24-255:~/log$ tail -f /var/log/neo4j |
```

Monitoring memory usage

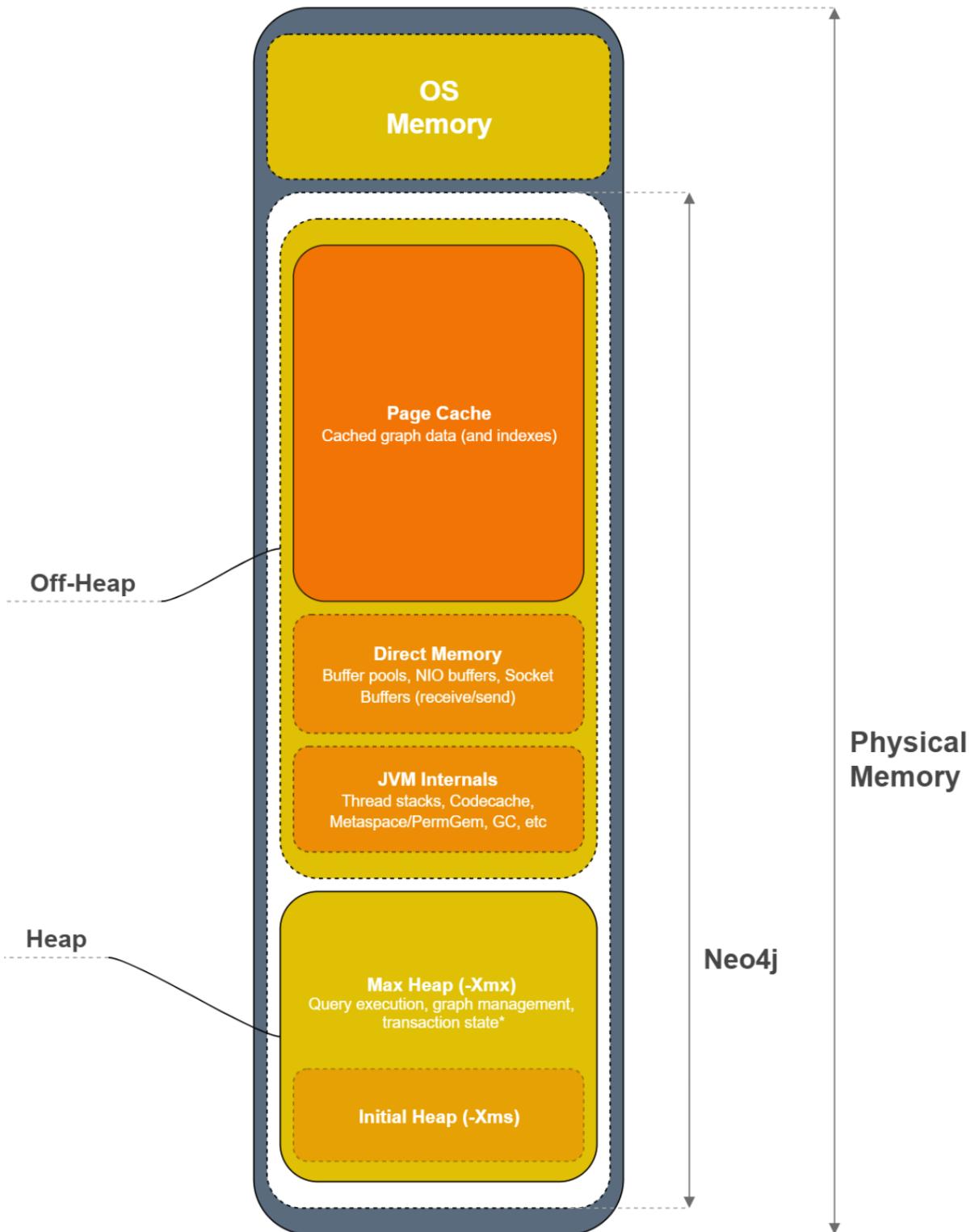
There are many properties that you can set to control how the Neo4j instance executes at runtime. The default values provided in the **neo4j.conf** file are useful for a small database with a small number of connections. In a production environment and in a Causal Cluster environment, you must make sure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

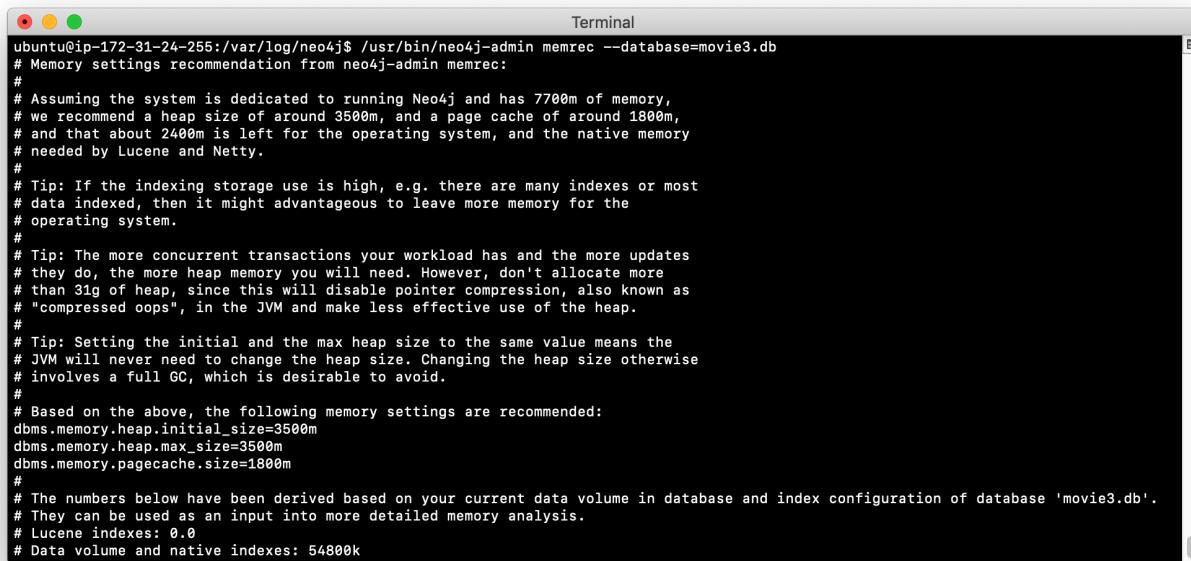
Neo4j Instance Memory Usage	Description
Heap	The JVM has a heap that is the runtime data area from which memory for all class instances and arrays are allocated. Heap storage for objects is reclaimed by an automatic storage management system (known as a garbage collector or GC).
Off-heap	Off-heap refers 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`:



```
Terminal
ubuntu@ip-172-31-24-255:/var/log/neo4j$ /usr/bin/neo4j-admin memrec --database=movie3.db
# Memory settings recommendation from neo4j-admin memrec:
#
# Assuming the system is dedicated to running Neo4j and has 7700m of memory,
# we recommend a heap size of around 3500m, and a page cache of around 1800m,
# and that about 2400m is left for the operating system, and the native memory
# needed by Lucene and Netty.
#
# Tip: If the indexing storage use is high, e.g. there are many indexes or most
# data indexed, then it might advantageous to leave more memory for the
# operating system.
#
# Tip: The more concurrent transactions your workload has and the more updates
# they do, the more heap memory you will need. However, don't allocate more
# than 31g of heap, since this will disable pointer compression, also known as
# "compressed oops", in the JVM and make less effective use of the heap.
#
# Tip: Setting the initial and the max heap size to the same value means the
# JVM will never need to change the heap size. Changing the heap size otherwise
# involves a full GC, which is desirable to avoid.
#
# Based on the above, the following memory settings are recommended:
dbms.memory.heap.initial_size=3500m
dbms.memory.heap.max_size=3500m
dbms.memory.pagecache.size=1800m
#
# The numbers below have been derived based on your current data volume in database and index configuration of database 'movie3.db'.
# They can be used as an input into more detailed memory analysis.
# Lucene indexes: 0.0
# Data volume and native indexes: 54800k
```

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

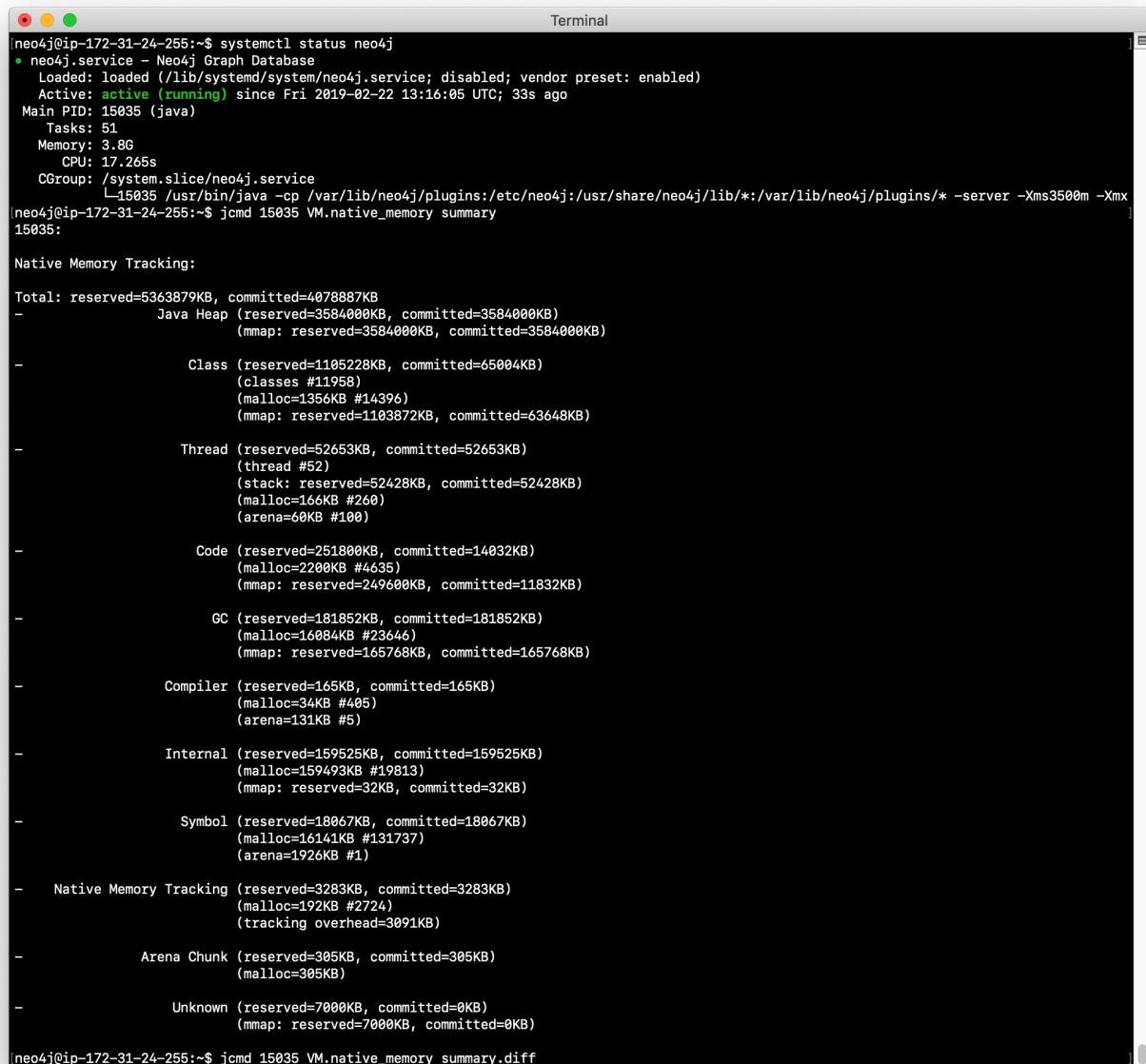
Monitoring memory consumption

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 jcmand to obtain summary information

One way that you can monitor memory usage for a running Neo4j instance is with the `jcmand` 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 `jcmand` execution to get summary information about memory usage on the system:



```
[neo4j@ip-172-31-24-255:~$ systemctl status neo4j
● neo4j.service - Neo4j Graph Database
  Loaded: loaded (/lib/systemd/system/neo4j.service; disabled; vendor preset: enabled)
  Active: active (running) since Fri 2019-02-22 13:16:05 UTC; 33s ago
    Main PID: 15035 (java)
      Tasks: 51
        Memory: 3.8G
          CPU: 17.265s
        CGroup: /system.slice/neo4j.service
            └─15035 /usr/bin/java -cp /var/lib/neo4j/plugins:/etc/neo4j:/usr/share/neo4j/lib/*:/var/lib/neo4j/plugins/* -server -Xms3500m -Xmx
[neo4j@ip-172-31-24-255:~$ jcmand 15035 VM.native_memory summary
15035:

Native Memory Tracking:

Total: reserved=5363879KB, committed=4078887KB
-
  Java Heap (reserved=3584000KB, committed=3584000KB)
    (mmap: reserved=3584000KB, committed=3584000KB)

-
  Class (reserved=1105228KB, committed=65004KB)
    (classes #11958)
    (malloc=1356KB #14396)
    (mmap: reserved=1103872KB, committed=63648KB)

-
  Thread (reserved=52653KB, committed=52653KB)
    (thread #52)
    (stack: reserved=52428KB, committed=52428KB)
    (malloc=166KB #260)
    (arena=60KB #100)

-
  Code (reserved=251800KB, committed=14032KB)
    (malloc=2200KB #4635)
    (mmap: reserved=249600KB, committed=11832KB)

-
  GC (reserved=181852KB, committed=181852KB)
    (malloc=16084KB #23646)
    (mmap: reserved=165768KB, committed=165768KB)

-
  Compiler (reserved=165KB, committed=165KB)
    (malloc=34KB #405)
    (arena=131KB #5)

-
  Internal (reserved=159525KB, committed=159525KB)
    (malloc=159493KB #19813)
    (mmap: reserved=32KB, committed=32KB)

-
  Symbol (reserved=18067KB, committed=18067KB)
    (malloc=16141KB #131737)
    (arena=1926KB #1)

-
  Native Memory Tracking (reserved=3283KB, committed=3283KB)
    (malloc=192KB #2724)
    (tracking overhead=3091KB)

-
  Arena Chunk (reserved=305KB, committed=305KB)
    (malloc=305KB)

-
  Unknown (reserved=7000KB, committed=0KB)
    (mmap: reserved=7000KB, committed=0KB)

[neo4j@ip-172-31-24-255:~$ jcmand 15035 VM.native_memory summary.diff]
```

Using jcmand in practice

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

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

NOTE In order to use `jcmand` 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

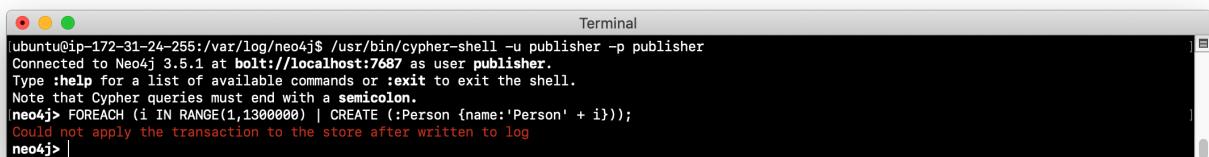
In this Exercise, you will execute a query that exhausts memory, then you will configure memory settings for the Neo4j instance and execute the query again.

Before you begin:

1. Make sure that you have exited out of any `cyphe-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.

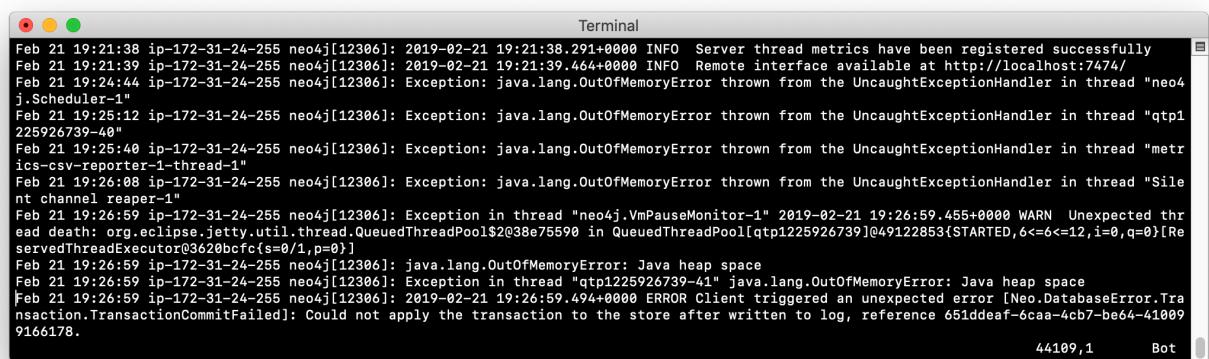
Exercise steps:

1. Start the Neo4j instance.
2. In `cyphe-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.



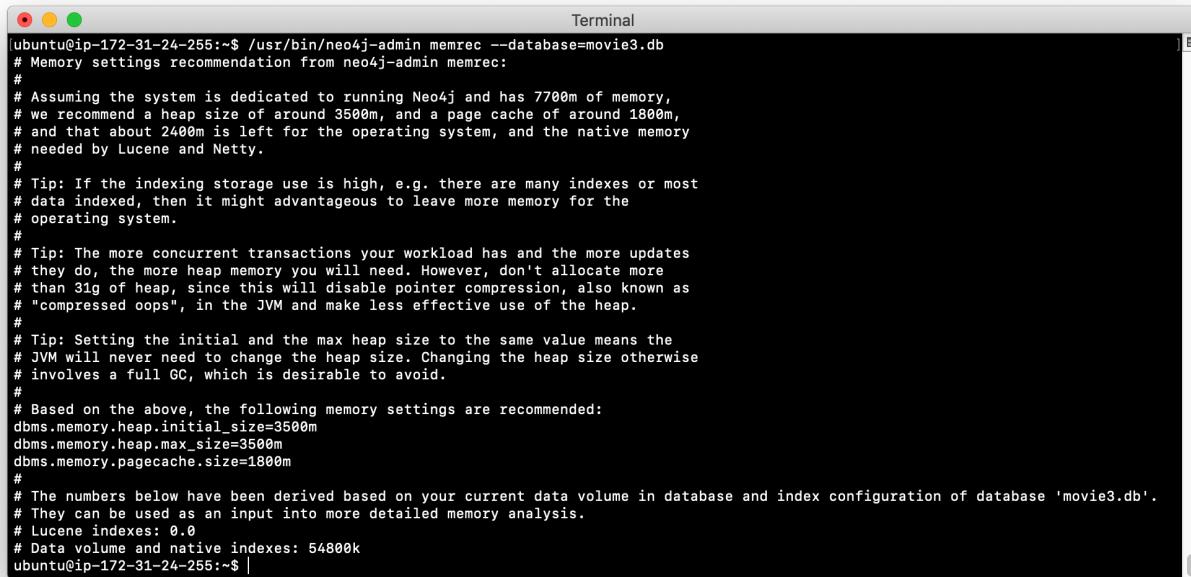
```
ubuntu@ip-172-31-24-255:/var/log/neo4j$ /usr/bin/cypher-shell -u publisher -p publisher
Connected to Neo4j 3.5.1 at bolt://localhost:7687 as user publisher.
Type :help for a list of available commands or :exit to exit the shell.
Note that Cypher queries must end with a semicolon.
|neo4j> FOREACH (i IN RANGE(1,1300000) | CREATE (:Person {name:'Person' + i}));
| Could not apply the transaction to the store after written to log
neo4j> |
```

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`.



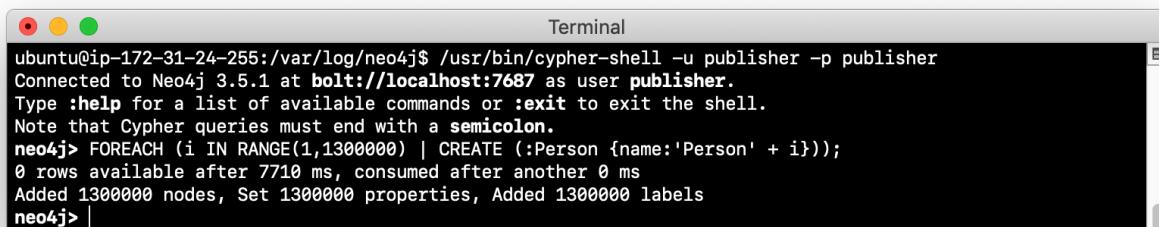
```
Feb 21 19:21:38 ip-172-31-24-255 neo4j[12306]: 2019-02-21 19:21:38.291+0000 INFO  Server thread metrics have been registered successfully
Feb 21 19:21:39 ip-172-31-24-255 neo4j[12306]: 2019-02-21 19:21:39.464+0000 INFO  Remote interface available at http://localhost:7474/
Feb 21 19:24:44 ip-172-31-24-255 neo4j[12306]: Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "neo4j.Scheduler-1"
Feb 21 19:25:12 ip-172-31-24-255 neo4j[12306]: Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "qtp1225926739-40"
Feb 21 19:25:40 ip-172-31-24-255 neo4j[12306]: Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "metrics-csv-reporter-1-thread-1"
Feb 21 19:26:08 ip-172-31-24-255 neo4j[12306]: Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "Sink channel reaper-1"
Feb 21 19:26:59 ip-172-31-24-255 neo4j[12306]: Exception in thread "neo4j.VmPauseMonitor-1" 2019-02-21 19:26:59.455+0000 WARN  Unexpected thread death: org.eclipse.jetty.util.thread.QueuedThreadPool$2@38e75590 in QueuedThreadPool[qtp1225926739]@49122863{STARTED,6<=6<=12,i=0,q=0}[ReceivedThreadExecutor@3620befc{s=0/1,p=0}]
Feb 21 19:26:59 ip-172-31-24-255 neo4j[12306]: java.lang.OutOfMemoryError: Java heap space
Feb 21 19:26:59 ip-172-31-24-255 neo4j[12306]: Exception in thread "qtp1225926739-41" java.lang.OutOfMemoryError: Java heap space
Feb 21 19:26:59 ip-172-31-24-255 neo4j[12306]: 2019-02-21 19:26:59.494+0000 ERROR Client triggered an unexpected error [Neo.DatabaseError.Transaction.TransactionCommitFailed]: Could not apply the transaction to the store after written to log, reference 651ddeadf-6caa-4cb7-be64-410099166178.
```

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**.



```
Terminal
ubuntu@ip-172-31-24-255:~$ /usr/bin/neo4j-admin memrec --database=movie3.db
# Memory settings recommendation from neo4j-admin memrec:
#
# Assuming the system is dedicated to running Neo4j and has 7700m of memory,
# we recommend a heap size of around 3500m, and a page cache of around 1800m,
# and that about 2400m is left for the operating system, and the native memory
# needed by Lucene and Netty.
#
# Tip: If the indexing storage use is high, e.g. there are many indexes or most
# data indexed, then it might advantageous to leave more memory for the
# operating system.
#
# Tip: The more concurrent transactions your workload has and the more updates
# they do, the more heap memory you will need. However, don't allocate more
# than 31g of heap, since this will disable pointer compression, also known as
# "compressed oops", in the JVM and make less effective use of the heap.
#
# Tip: Setting the initial and the max heap size to the same value means the
# JVM will never need to change the heap size. Changing the heap size otherwise
# involves a full GC, which is desirable to avoid.
#
# Based on the above, the following memory settings are recommended:
dbms.memory.heap.initial_size=3500m
dbms.memory.heap.max_size=3500m
dbms.memory.pagecache.size=1800m
#
# The numbers below have been derived based on your current data volume in database and index configuration of database 'movie3.db'.
# They can be used as an input into more detailed memory analysis.
# Lucene indexes: 0.0
# Data volume and native indexes: 54800k
ubuntu@ip-172-31-24-255:~$ |
```

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.



```
Terminal
ubuntu@ip-172-31-24-255:/var/log/neo4j$ /usr/bin/cypher-shell -u publisher -p publisher
Connected to Neo4j 3.5.1 at bolt://localhost:7687 as user publisher.
Type :help for a list of available commands or :exit to exit the shell.
Note that Cypher queries must end with a semicolon.
neo4j> FOREACH (i IN RANGE(1,1300000) | CREATE (:Person {name:'Person' + i}));
0 rows available after 7710 ms, consumed after another 0 ms
Added 1300000 nodes, Set 1300000 properties, Added 1300000 labels
neo4j> |
```

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

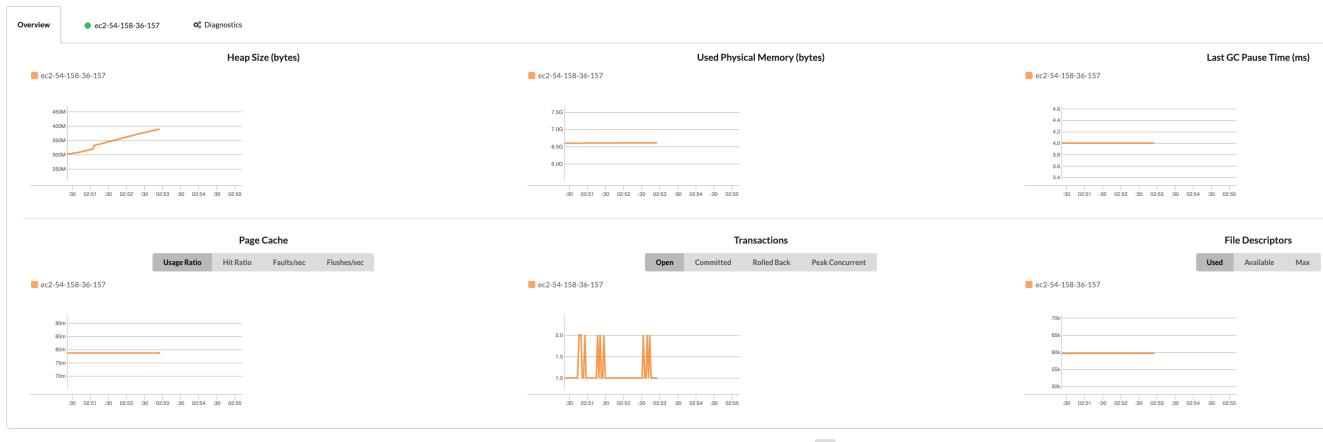
The Neo4j instance automatically collects metrics in the default location for metrics (for example, on Debian, all metrics are placed in `/var/lib/neo4j/metrics`). If for some reason, you do not want metrics collected, you can disable them by setting `metrics.enabled=false` in the Neo4j configuration.

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:

```
Terminal
+-----+-----+
| type      | row          | attributes[row]["value"] |
+-----+-----+
"StoreSizes" | "LogicalLogSize" | 52
"StoreSizes" | "StringStoreSize" | 8192
"StoreSizes" | "ArrayStoreSize" | 24576
"StoreSizes" | "RelationshipStoreSize" | 16320
"StoreSizes" | "PropertyStoreSize" | 106621812
"StoreSizes" | "TotalStoreSize" | 506203963
"StoreSizes" | "NodeStoreSize" | 39008970
"PageCache" | "Hits" | 1605533
"PageCache" | "FileUnmappings" | 20
"PageCache" | "FileMappings" | 37
"PageCache" | "Faults" | 18052
"PageCache" | "EvictionExceptions" | 0
"PageCache" | "Flushes" | 1
"PageCache" | "BytesWritten" | 8192
"PageCache" | "UsageRatio" | 0.07865692387463345
"PageCache" | "Unpins" | 1623584
"PageCache" | "Evictions" | 0
"PageCache" | "BytesRead" | 147385584
"PageCache" | "Pins" | 1623585
"PageCache" | "HitRatio" | 0.9888813951841142
"ID Allocations" | "NumberOfRelationshipIdsInUse" | 253
"ID Allocations" | "NumberOfPropertyIdsInUse" | 2600383
"ID Allocations" | "NumberOfNodeIdInUse" | 2600171
"ID Allocations" | "NumberOfRelationshipTypeIdsInUse" | 6
"Transactions" | "NumberOfRolledBackTransactions" | 0
"Transactions" | "LastCommittedTxId" | 20
"Transactions" | "NumberOfOpenTransactions" | 1
"Transactions" | "NumberOfOpenedTransactions" | 3
"Transactions" | "PeakNumberOfConcurrentTransactions" | 1
"Transactions" | "NumberOfCommittedTransactions" | 2
+-----+-----+
30 rows available after 969 ms, consumed after another 4 ms
neo4j> |
```

Exercise #6: Querying with JMX

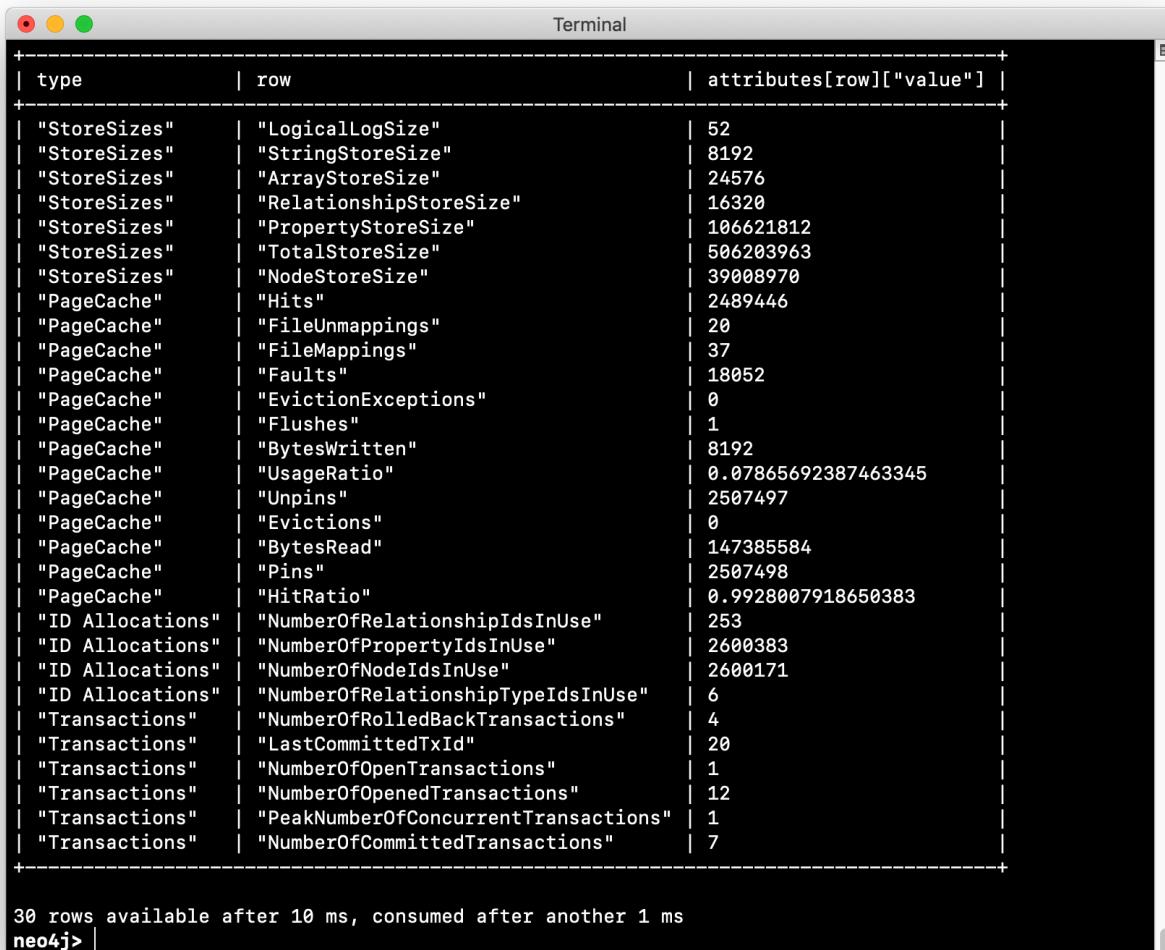
In this Exercise, you will execute a JMX query to view metrics about the Neo4j instance.

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.



The screenshot shows a Mac OS X Terminal window titled "Terminal". The window displays the results of a Cypher query. The output is a table with three columns: "type", "row", and "attributes[row]["value"]". The table contains 30 rows of data, representing various Neo4j metrics. The data includes numerical values for store sizes, page cache statistics, transaction counts, and usage ratios. The last two lines of the output indicate that 30 rows were available after 10 ms and consumed after another 1 ms.

type	row	attributes[row]["value"]
"StoreSizes"	"LogicalLogSize"	52
"StoreSizes"	"StringStoreSize"	8192
"StoreSizes"	"ArrayStoreSize"	24576
"StoreSizes"	"RelationshipStoreSize"	16320
"StoreSizes"	"PropertyStoreSize"	106621812
"StoreSizes"	"TotalStoreSize"	506203963
"StoreSizes"	"NodeStoreSize"	39008970
"PageCache"	"Hits"	2489446
"PageCache"	"FileUnmappings"	20
"PageCache"	"FileMappings"	37
"PageCache"	"Faults"	18052
"PageCache"	"EvictionExceptions"	0
"PageCache"	"Flushes"	1
"PageCache"	"BytesWritten"	8192
"PageCache"	"UsageRatio"	0.07865692387463345
"PageCache"	"Unpins"	2507497
"PageCache"	"Evictions"	0
"PageCache"	"BytesRead"	147385584
"PageCache"	"Pins"	2507498
"PageCache"	"HitRatio"	0.9928007918650383
"ID Allocations"	"NumberOfRelationshipIdsInUse"	253
"ID Allocations"	"NumberOfPropertyIdsInUse"	2600383
"ID Allocations"	"NumberOfNodeIdInUse"	2600171
"ID Allocations"	"NumberOfRelationshipTypeIdsInUse"	6
"Transactions"	"NumberOfRolledBackTransactions"	4
"Transactions"	"LastCommittedTxId"	20
"Transactions"	"NumberOfOpenTransactions"	1
"Transactions"	"NumberOfOpenedTransactions"	12
"Transactions"	"PeakNumberOfConcurrentTransactions"	1
"Transactions"	"NumberOfCommittedTransactions"	7

```
30 rows available after 10 ms, consumed after another 1 ms
neo4j> |
```

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.

```
Terminal
+-----+-----+
| type      | row          | attributes[row]["value"] |
+-----+-----+
"StoreSizes" | "LogicalLogSize" | 179400156
"StoreSizes" | "StringStoreSize" | 8192
"StoreSizes" | "ArrayStoreSize" | 24576
"StoreSizes" | "RelationshipStoreSize" | 16320
"StoreSizes" | "PropertyStoreSize" | 159916400
"StoreSizes" | "TotalStoreSize" | 759160901
"StoreSizes" | "NodeStoreSize" | 58509360
"PageCache" | "Hits" | 5098957
"PageCache" | "FileUnmappings" | 21
"PageCache" | "FileMappings" | 38
"PageCache" | "Faults" | 27058
"PageCache" | "EvictionExceptions" | 0
"PageCache" | "Flushes" | 9
"PageCache" | "BytesWritten" | 73638717
"PageCache" | "UsageRatio" | 0.11789824098159937
"PageCache" | "Unpins" | 5126014
"PageCache" | "Evictions" | 0
"PageCache" | "BytesRead" | 147385584
"PageCache" | "Pins" | 5126015
"PageCache" | "HitRatio" | 0.9947214356571332
"ID Allocations" | "NumberOfRelationshipIdsInUse" | 253
"ID Allocations" | "NumberOfPropertyIdsInUse" | 3900383
"ID Allocations" | "NumberOfNodeIdInUse" | 3900171
"ID Allocations" | "NumberOfRelationshipTypeIdsInUse" | 6
"Transactions" | "NumberOfRolledBackTransactions" | 4
"Transactions" | "LastCommittedTxId" | 21
"Transactions" | "NumberOfOpenTransactions" | 1
"Transactions" | "NumberOfOpenedTransactions" | 14
"Transactions" | "PeakNumberOfConcurrentTransactions" | 1
"Transactions" | "NumberOfCommittedTransactions" | 9
+-----+-----+
30 rows available after 9 ms, consumed after another 1 ms
neo4j> |
```

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.

- jcmsg
- 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.

