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.

What can be monitored and measured 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, you can configure Neo4j to write 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 lesson.

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.

In the previous lesson a about security, 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 lesson 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*.

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:

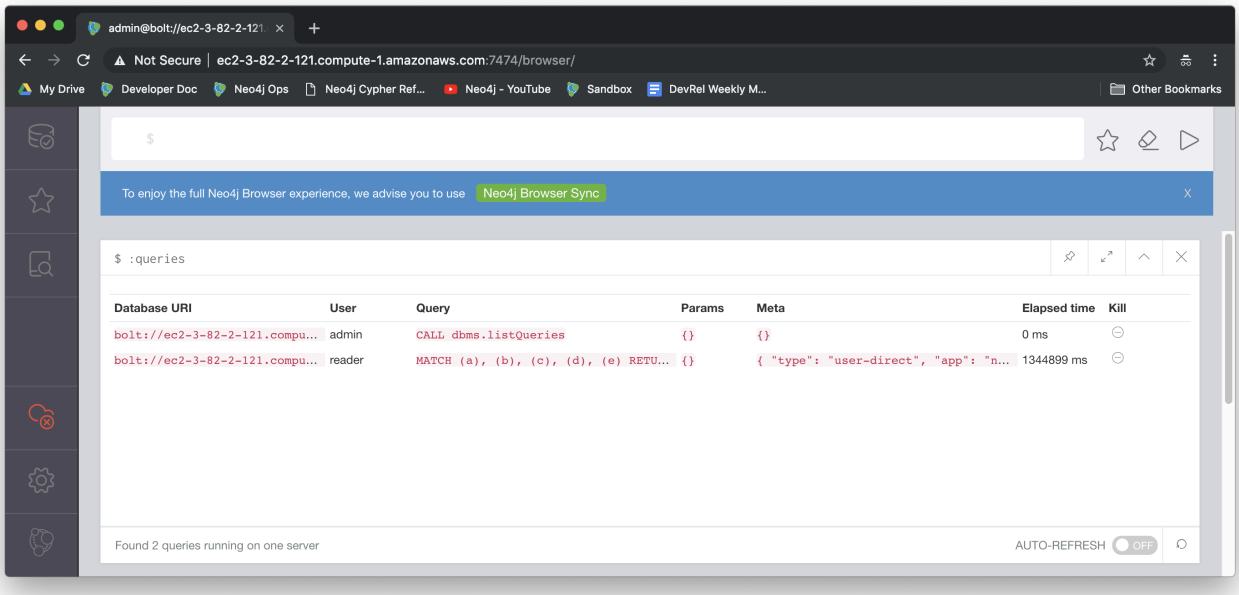
```
dbms.logs.query.enabled=true
dbms.logs.query.threshold=1000ms
dbms.logs.query.parameter_logging_enabled=true
dbms.logs.query.time_logging_enabled=true
dbms.logs.query.allocation_logging_enabled=true
dbms.logs.query.page_logging_enabled=true
dbms.track_query_cpu_time=true
dbms.track_query_allocation=true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

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

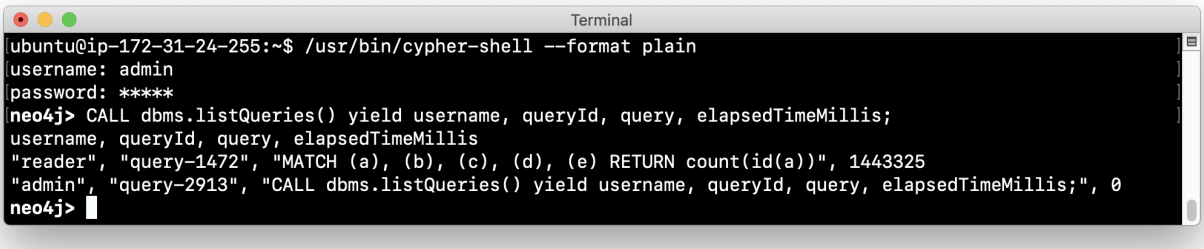
Viewing currently running queries

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?

In the *Introduction to Neo4j* course you learned that long-running Cypher queries can be monitored and killed from a Neo4j browser session. There is a difference between a query that runs in the Neo4j instance for a long time and a query that has run an returns a large result set. You should focus on the queries that run for a long time. In Neo4j Browser you can use the `:queries` command to see all currently running queries:



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



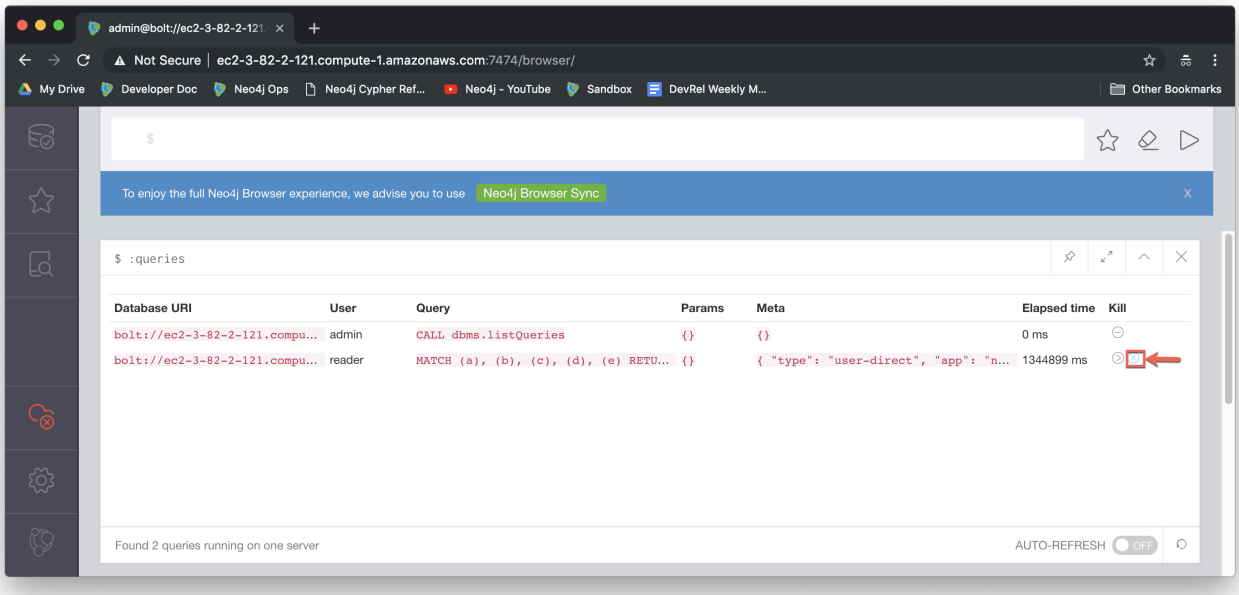
Another useful statement, you can used to view long-running queries and any type of transaction running in the Neo4j instance is by calling `dbms.listTransactions()` which you will use in the next Exercise.

If you have the *admin* role, you can view (and kill) queries from all users.

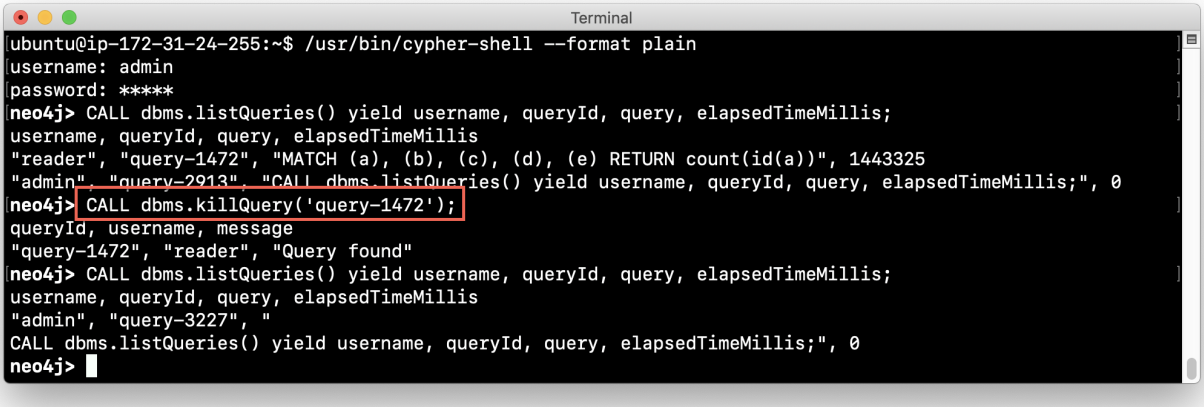
Killing a long-running query

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.



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



Exercise #1: Monitoring queries

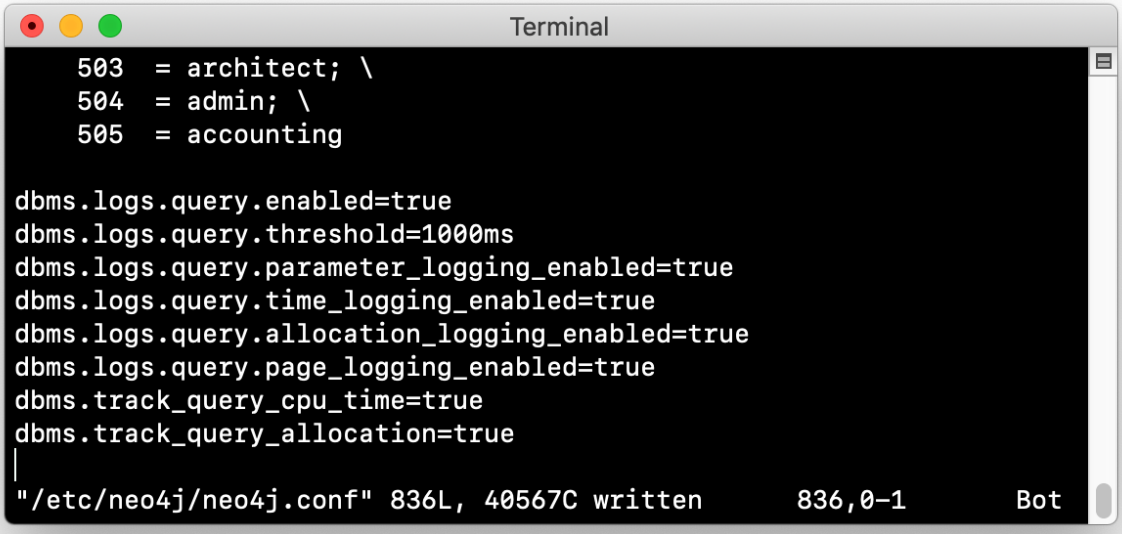
In this Exercise, you enable query logging where an event will be written to the **query.log** file for a query that took more than 1000ms to complete. Then you will monitor and detect a long-running query and kill it.

Before you begin:

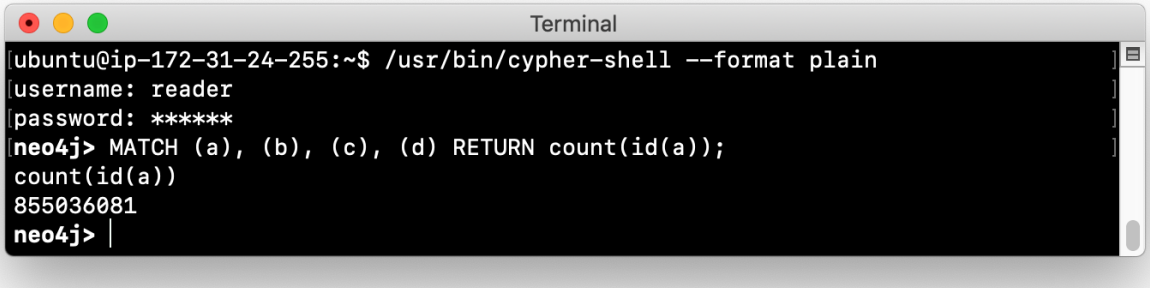
1. For this exercise, you will be using the stand-alone Neo4j instance that you configured for authentication in the previous lesson.
2. In a terminal window, modify the **neo4j.conf** file for the stand-alone Neo4j instance to use the **movie3.db**, rather than the **crimes.db**.

Exercise steps:

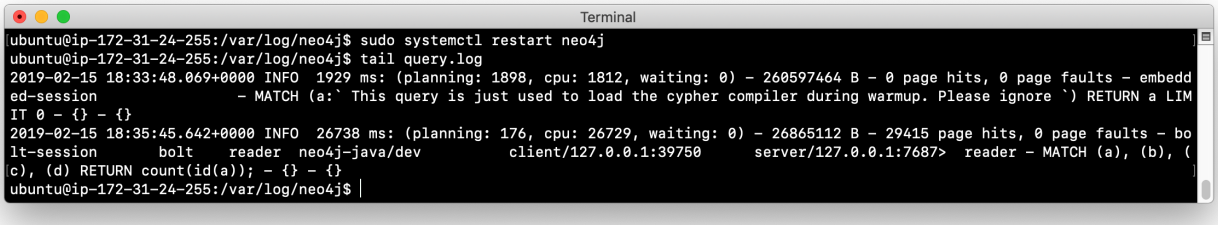
1. Modify the **neo4j.conf** file to create a log record if a query exceeds 1000 ms.



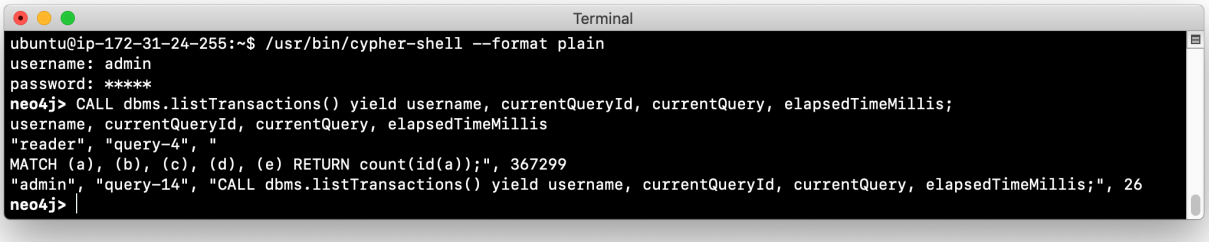
2. Start/restart the Neo4j stand-alone instance.
3. Open a new terminal window and log in to `cypher-shell` with the *reader/reader* credentials. (**Suggestion**: specify --format plain)
4. In this `cypher-shell` session, enter the following statement which will execute a query that runs for longer than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



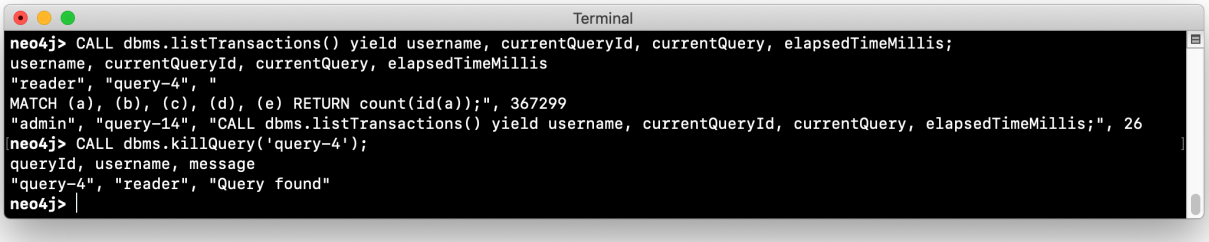
6. In the terminal window where you started the Neo4j instance, view the **query.log**. Is there a record for this query?



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.

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

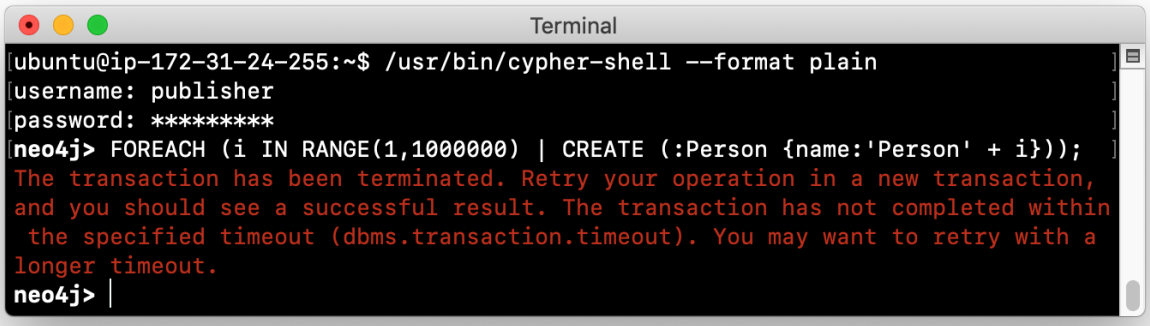
In this Exercise, you configure Neo4j to not allow transactions that take longer than one second to complete.

Before you begin:

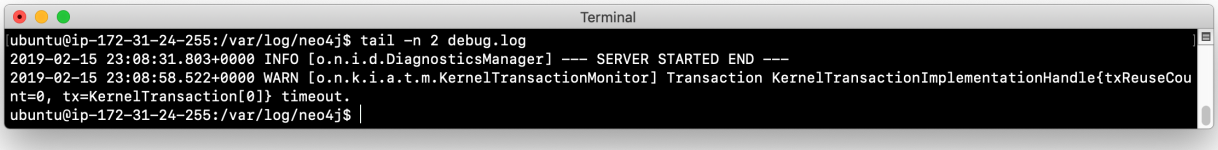
For this exercise, you will be using the stand-alone Neo4j instance that you used in the previous Exercise.

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 lesson.

Monitoring locks

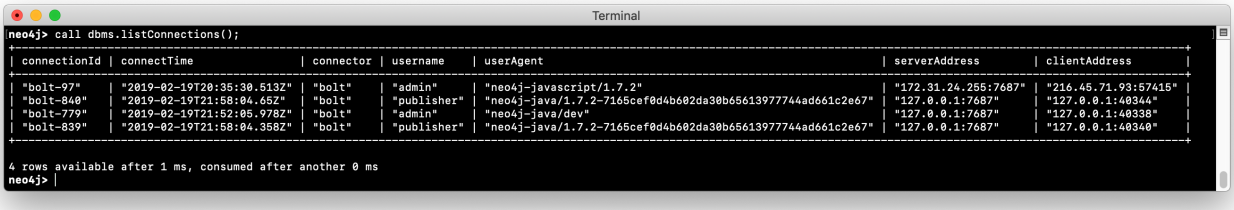
You can query the Neo4j instance's currently running transactions. If you see transactions that are running for a long time, you can further query the Neo4j instance to determine what locks each long-running query is holding. To read more about monitoring locks, see this Neo4j Support Knowledge Base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

Monitoring connections

A Neo4j instance (stand-alone or in a Causal Cluster) uses a set of ports for inter-cluster communication as well as 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. In fact, 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.

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.

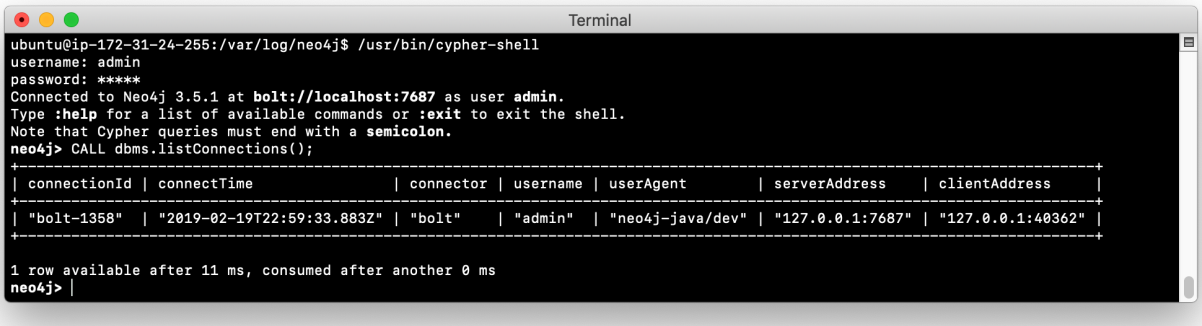
Before you begin:

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**.

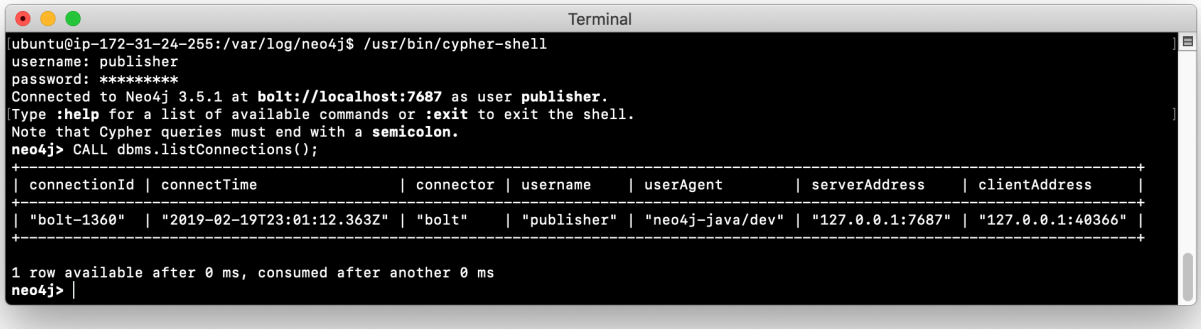
4. Make sure the **write.sh** has execute permissions (`chmod +x write.sh`)

Exercise steps:

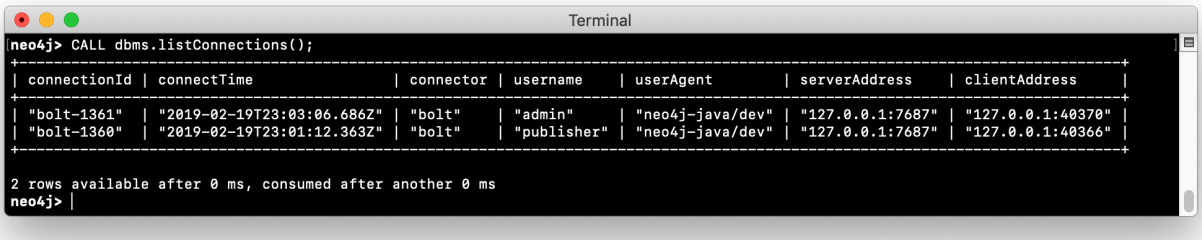
1. In a terminal window, log in to `cypher-shell` with the credentials *admin/admin*.
2. Enter the Cypher statement to list all connections to the Neo4j instance.



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?

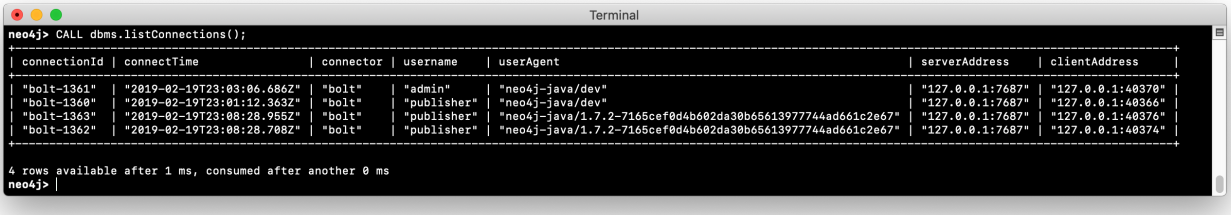


5. In the first *admin* `cypher-shell` session, enter the Cypher statement to list all connections to the Neo4j instance. Do you see all of the connections?

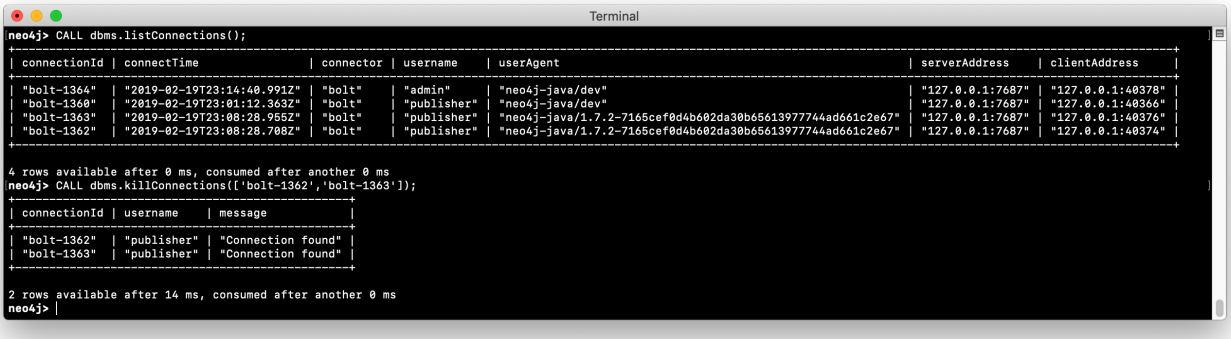


6. In a third terminal window navigate to the **writeApp** folder you created when you unzipped the java application.

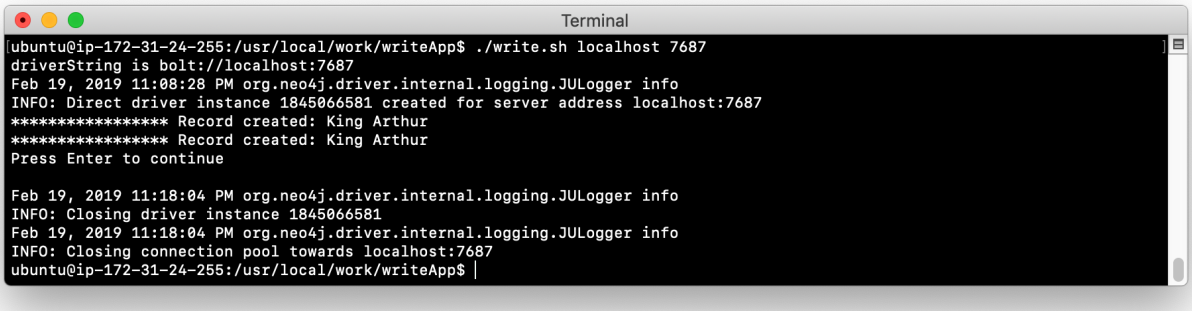
7. Enter `./write.sh localhost 7687`. This java application will open a connection to the Neo4j instance and will ask you to press **Enter** to continue. Do not press **Enter**.
8. In the *admin* `cypher-shell` session, enter the Cypher statement to list all connections.



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

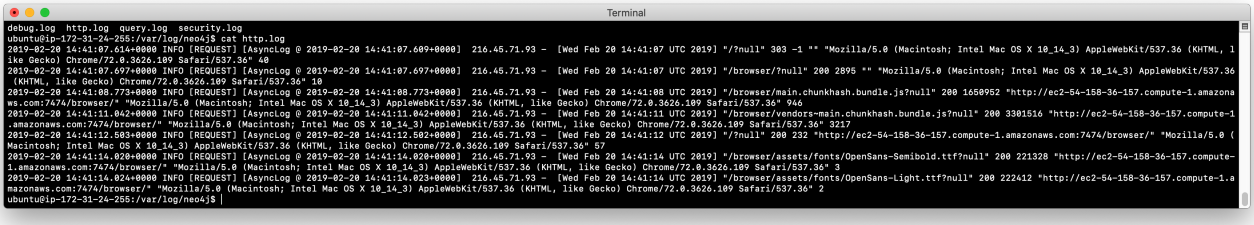
In this Exercise, you enable the Neo4j instance for logging HTTP requests and monitor them.

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

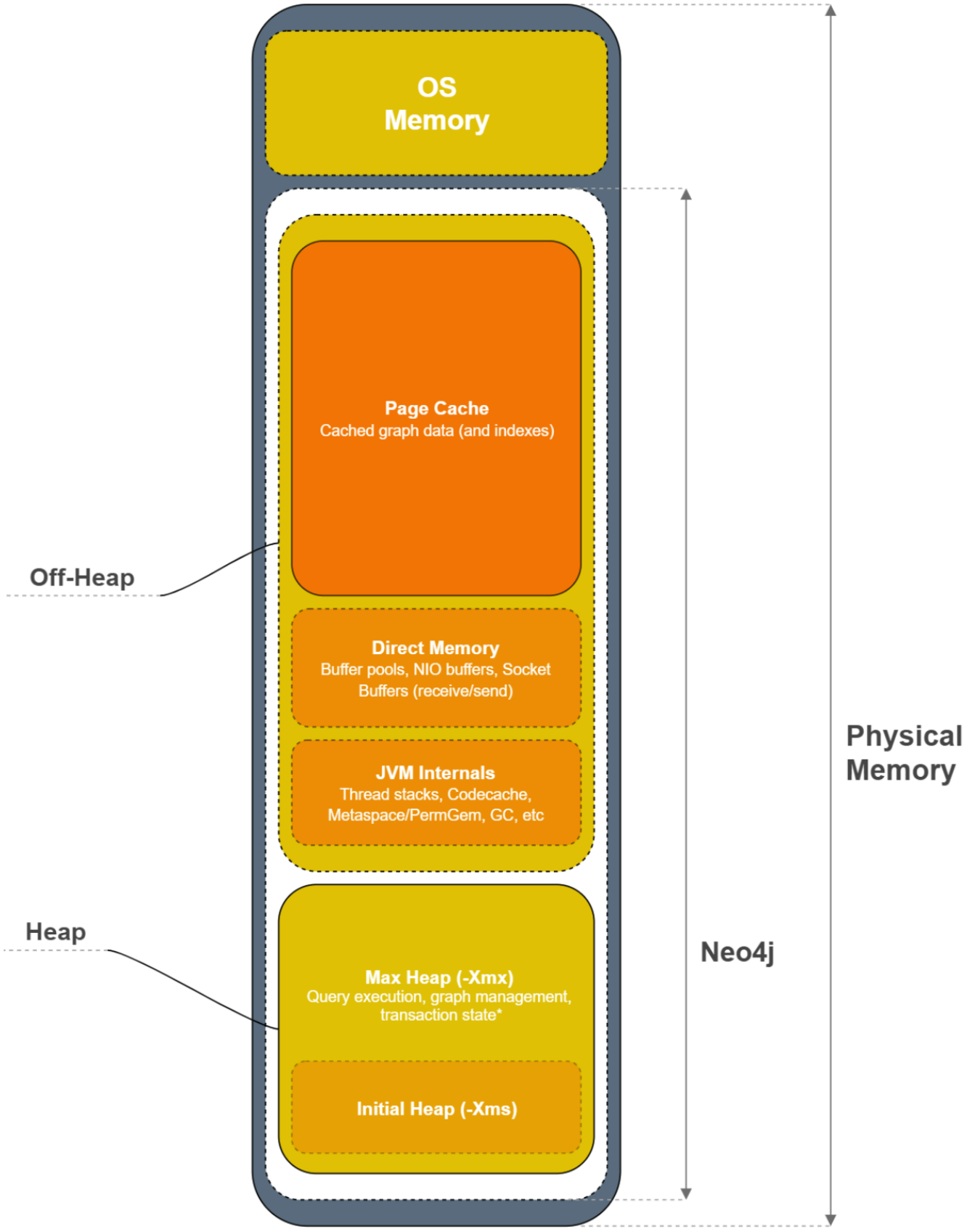
There are many properties that you can set to control how the Neo4j instance executes at runtime. The default values provided in the **neo4j.conf** file are useful for a small database with a small number of connections. In a production environment and in a Causal Cluster environment, you must make sure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

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:

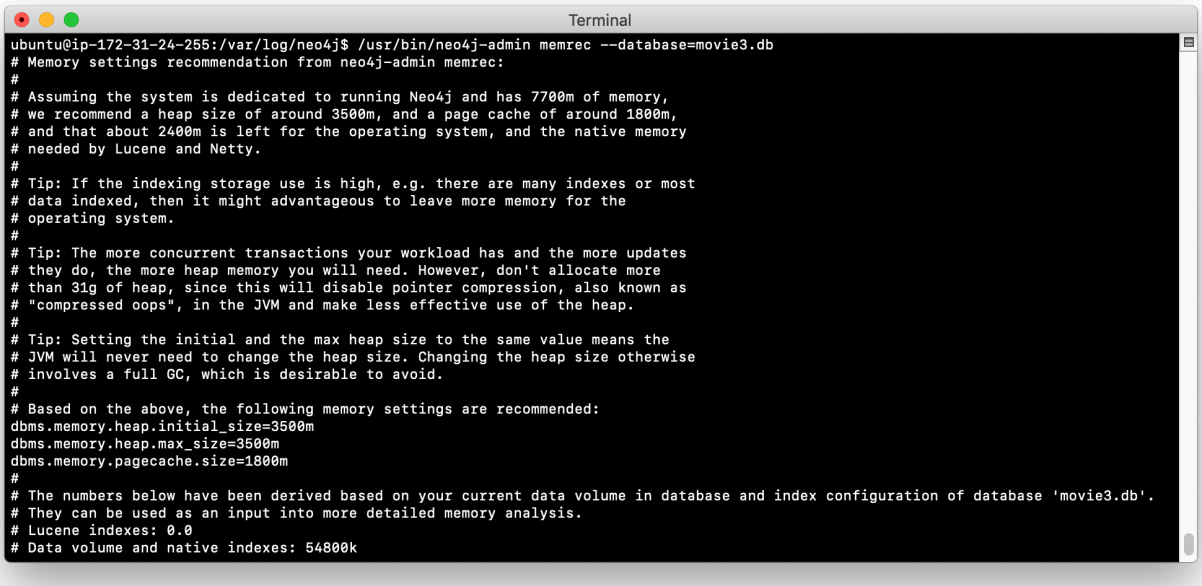
Neo4j Instance Memory Usage	Description
Heap	The JVM has a heap that is the runtime data area from which memory for all class instances and arrays are allocated. Heap storage for objects is reclaimed by an automatic storage management system (known as a garbage collector or GC).
Off-heap	Off-heap refers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

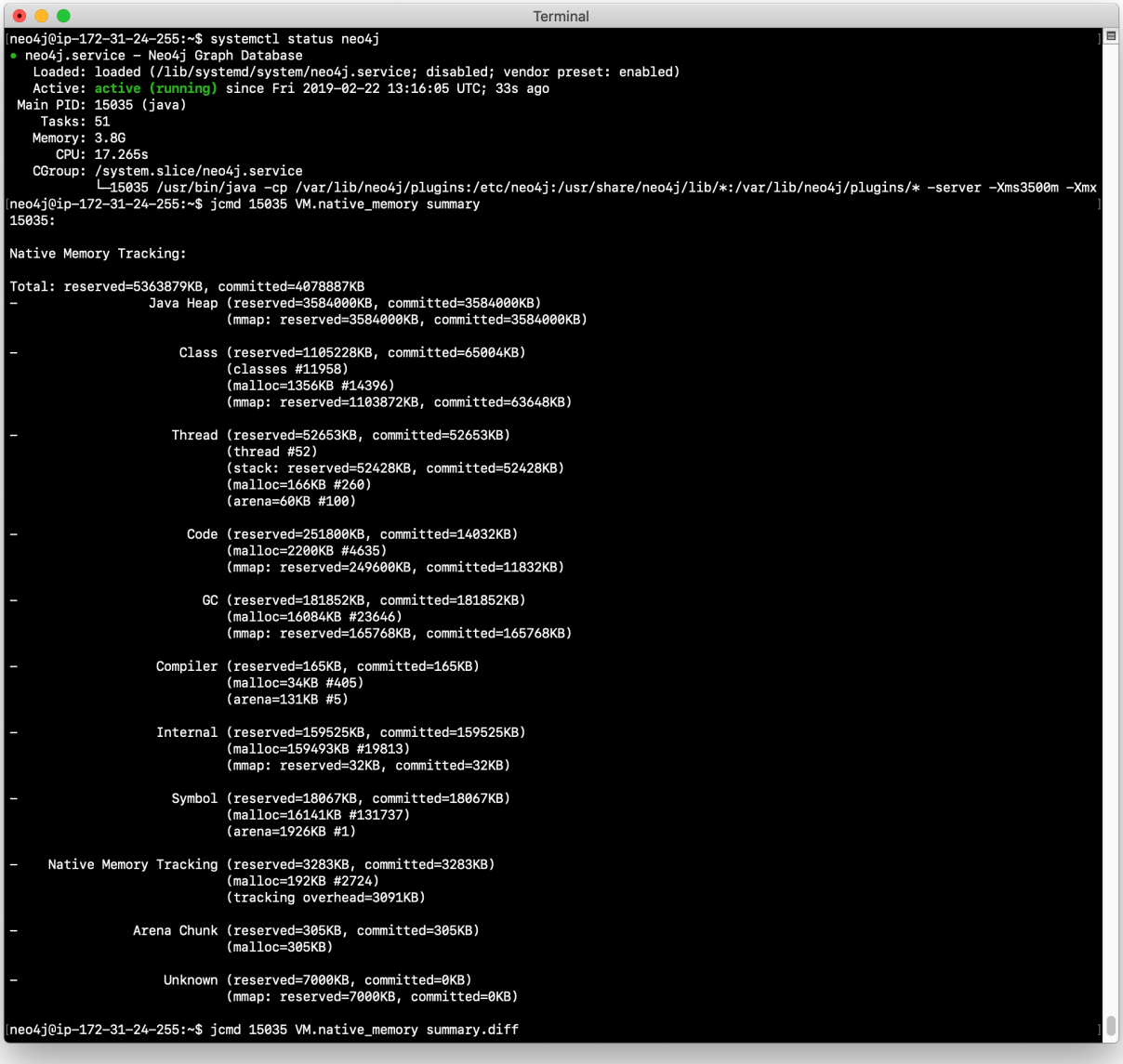
also information about available memory on your system.

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 of the problem.

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:



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

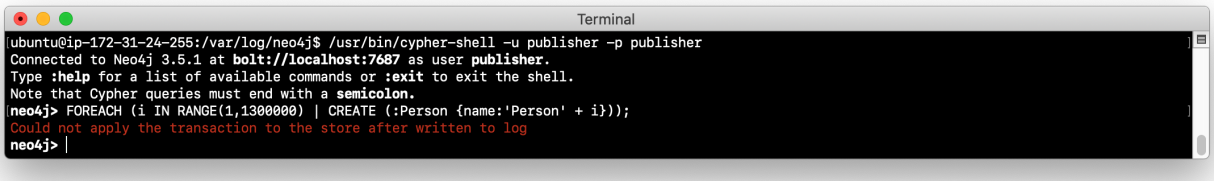
In this Exercise, you will execute a query that exhausts memory, then you will configure memory settings for the Neo4j instance and execute the query again.

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.

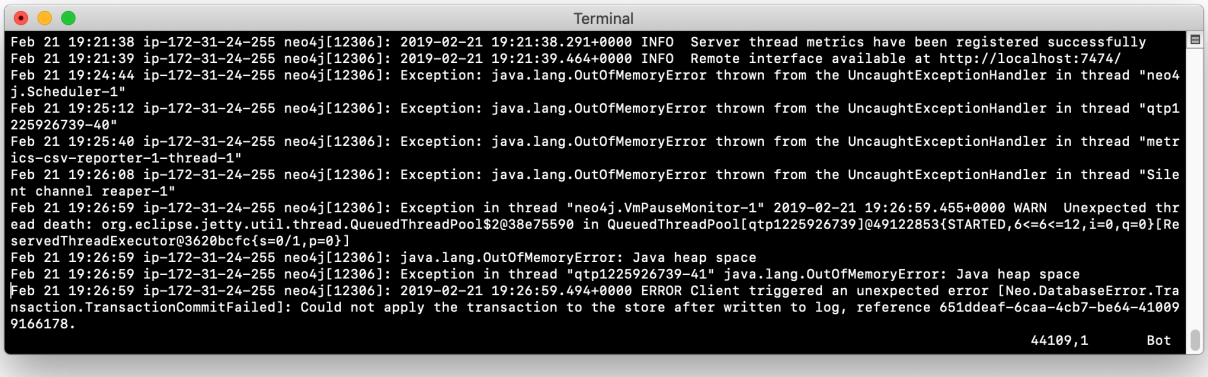
Exercise steps:

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.

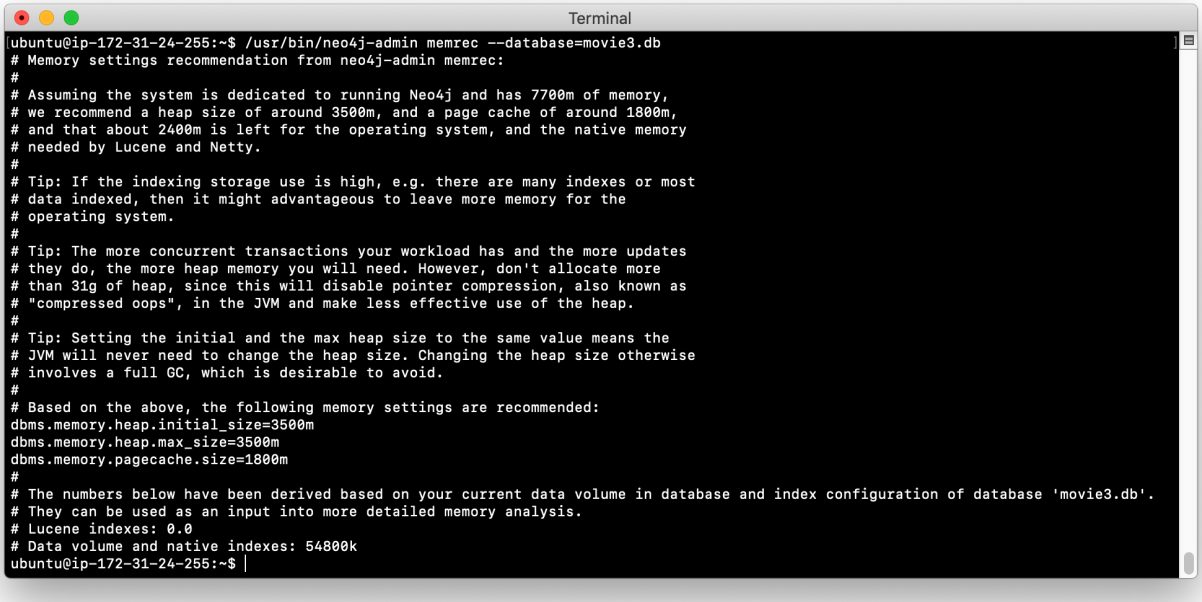


5. View the the Neo4j log **Hint**: journalctl -e -u neo4j on Debian. It should also have an error logged

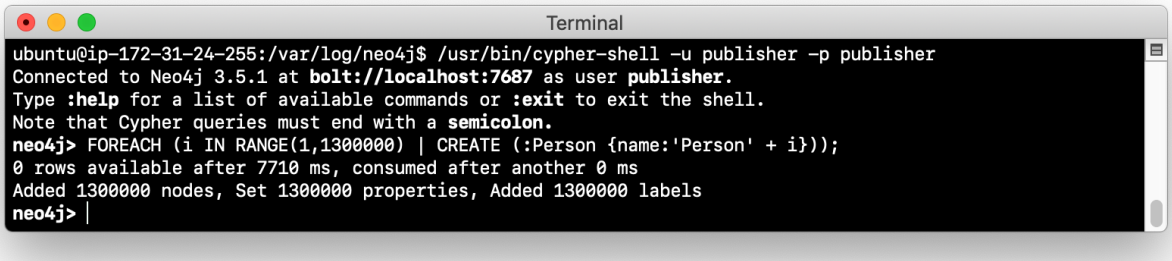
as well as an error in **debug.log**.



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 specifing 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`.
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.



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 with 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.

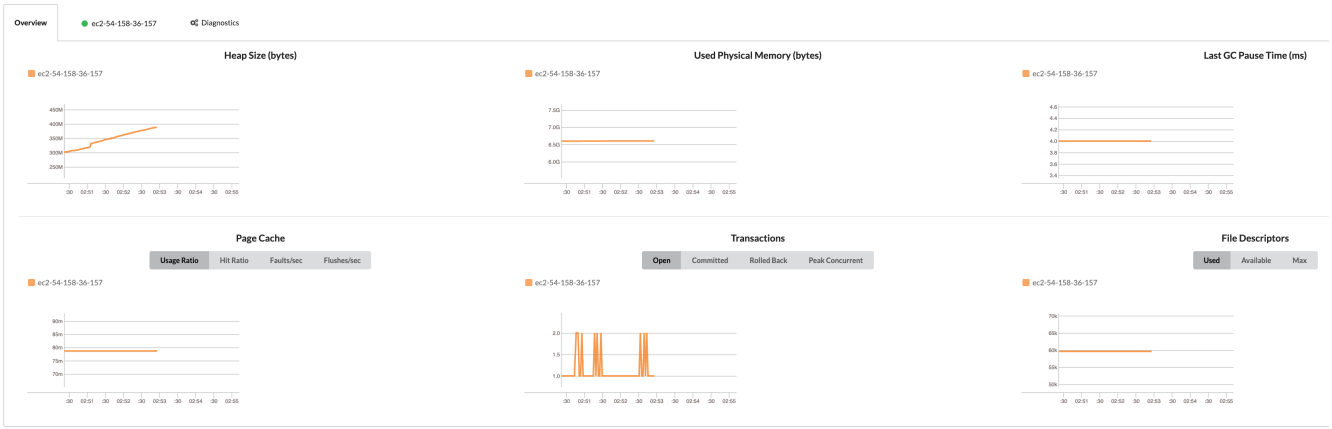
All metrics are written to CSV files in the **metrics** directory. With these files, you can use a visualization tool to view historical or current metrics for the Neo4j instance.

There are other options for collecting and viewing metrics which 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:



Using JMX queries

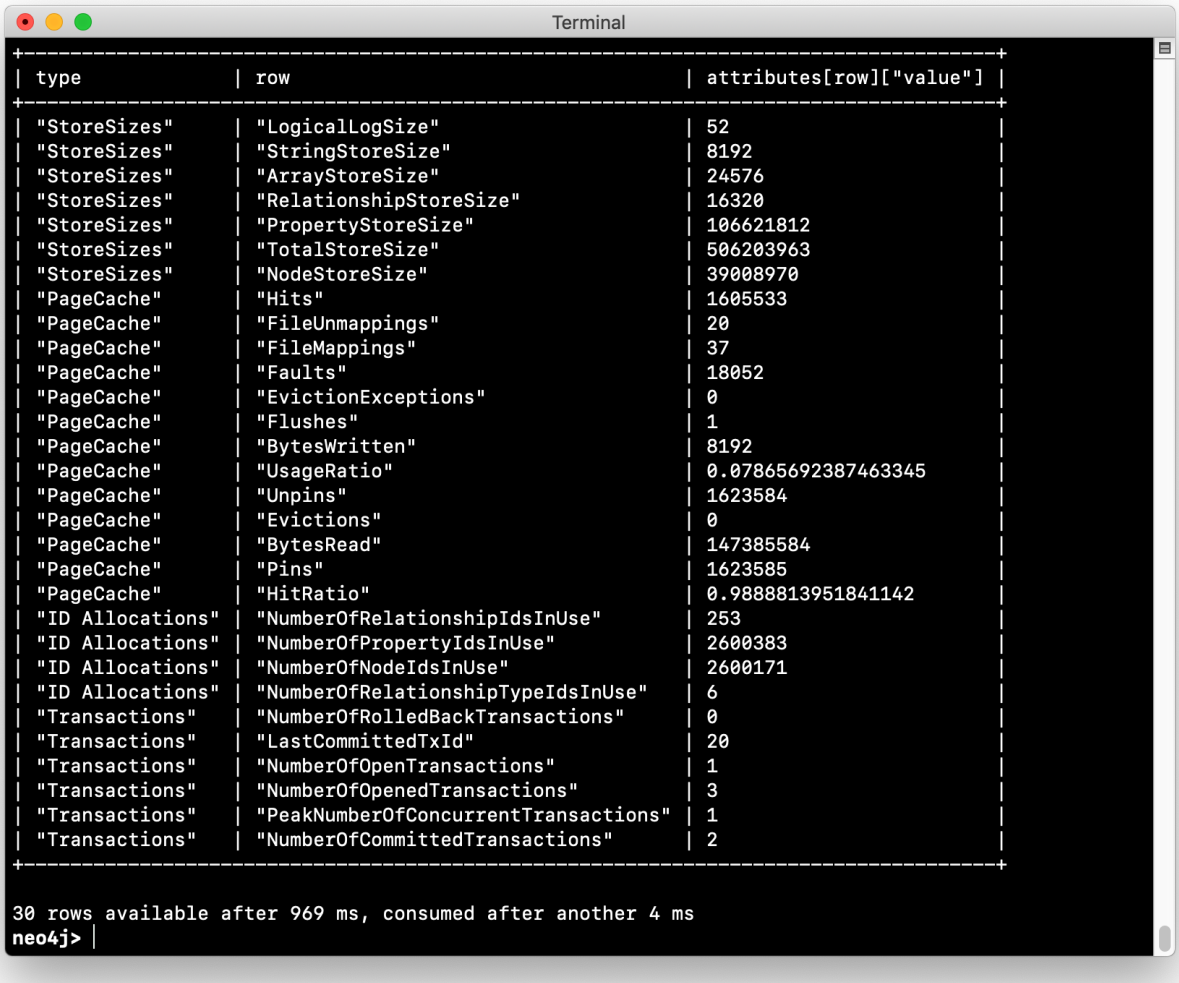
A Neo4j instance can be monitored with Java Management Extensions (JMX). JMX is a low-level mechanism for monitoring the Neo4j instance. However, a best practice is not to use it for remote monitoring as it is a security vulnerability. In addition, running a tool such as `jconsole` that uses JMX can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

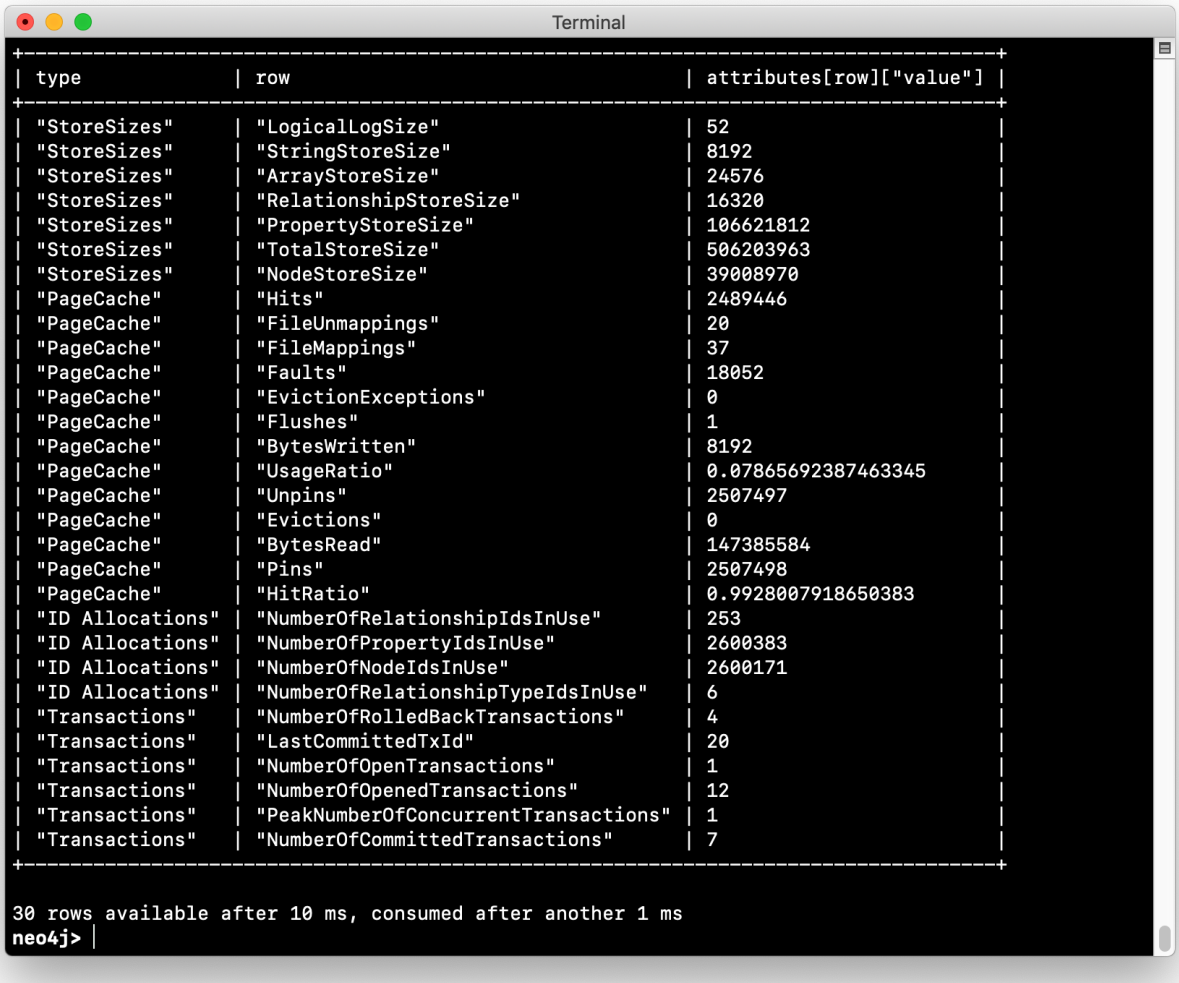
In this Exercise, you will execute a JMX query to view metrics about the Neo4j instance.

Before you begin:

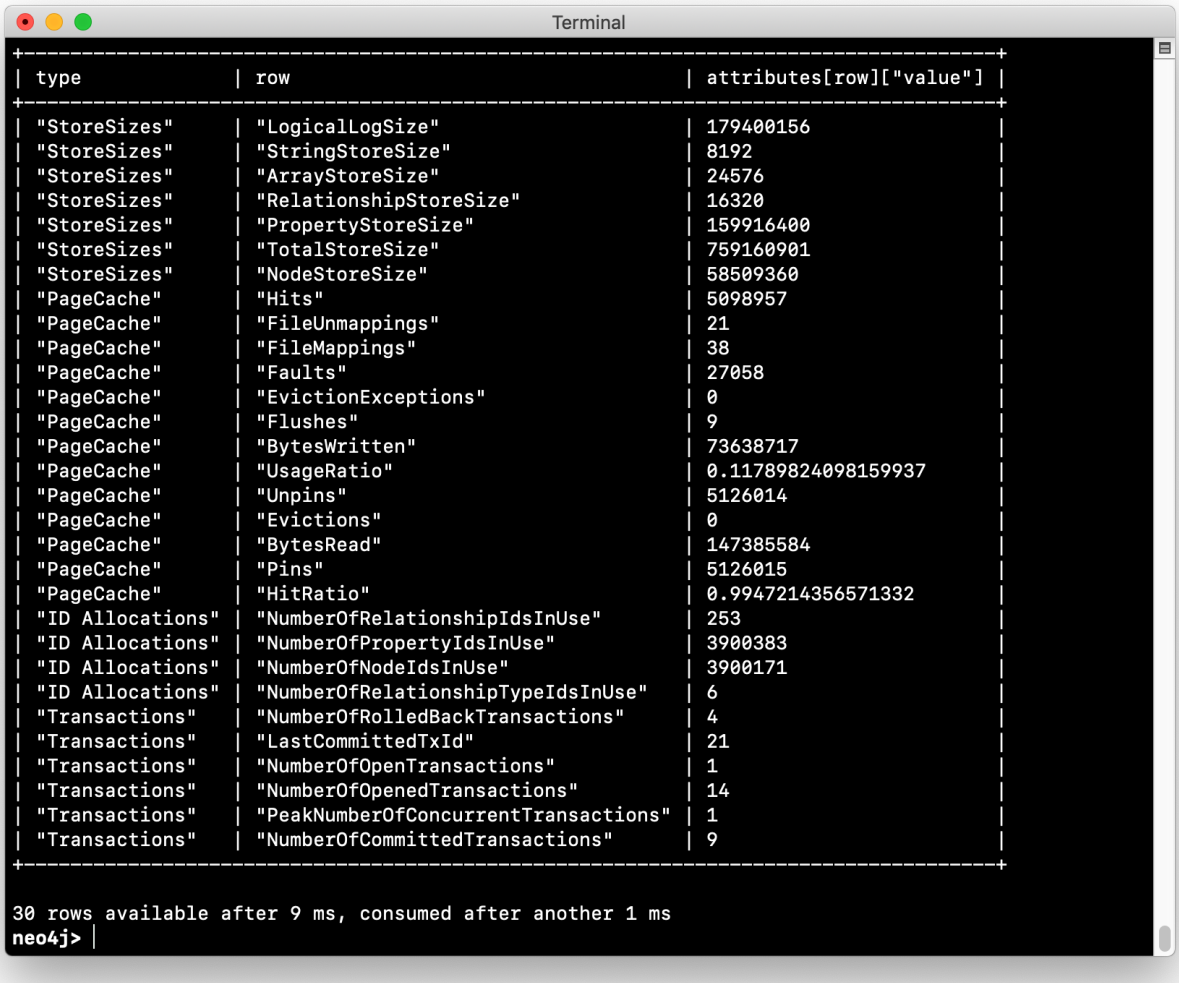
1. Make sure you have a started Neo4j instance.
2. Open a terminal window.

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.


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"	"NumberOfNodeIdsInUse"	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.

- ▢ jcmd
- ▢ vmstat
- ▢ neo4j-admin memrec
- ▢ neo4j-admin analyze

Question 3

How can Neo4j metrics be collected?

Select the correct answers.

- ▢ Placed in CSV files for tools to use.
- ▢ Publish to an endpoint using the Graphite protocol.
- ▢ Publish 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.