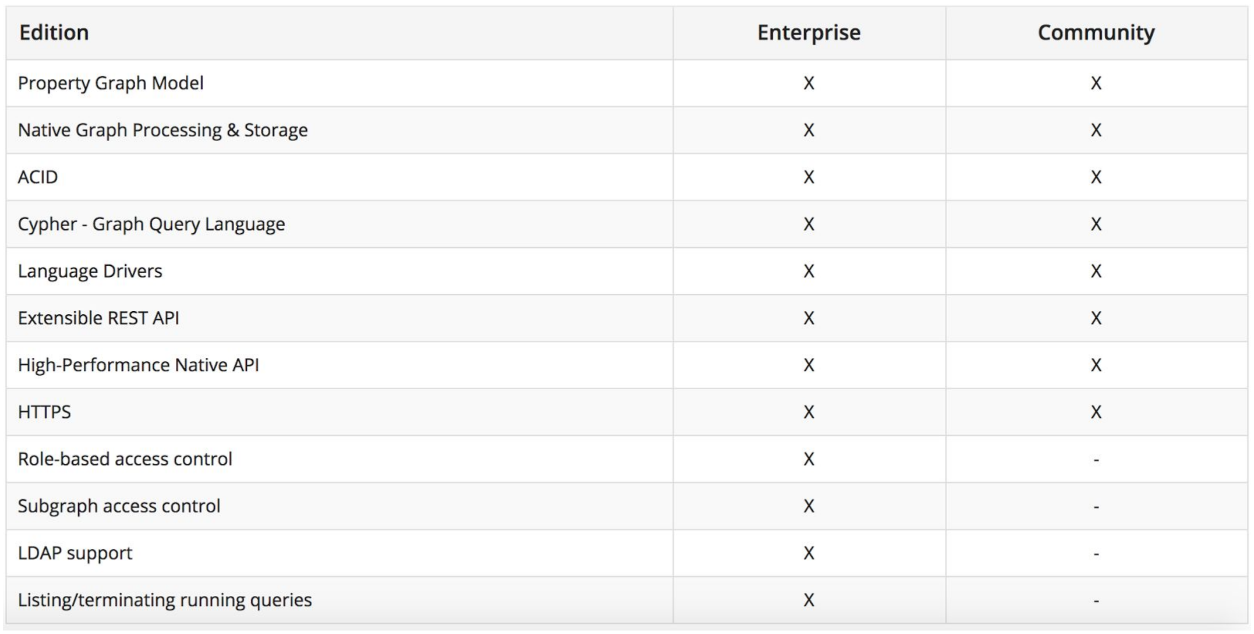
Neo4j Onboarding Handbook

Version 3.3

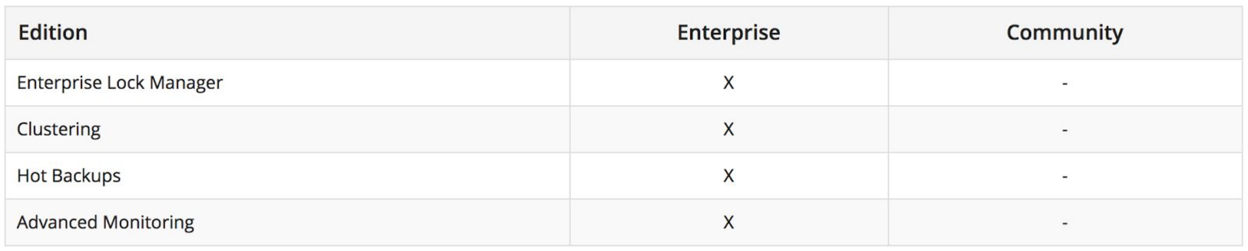
Table of Contents

1. Install and Config

- ## Editions: Enterprise vs. Community



Edition	Enterprise	Community
Property Graph Model	X	X
Native Graph Processing & Storage	X	X
ACID	X	X
Cypher - Graph Query Language	X	X
Language Drivers	X	X
Extensible REST API	X	X
High-Performance Native API	X	X
HTTPS	X	X
Role-based access control	X	-
Subgraph access control	X	-
LDAP support	X	-
Listing/terminating running queries	X	-



Edition	Enterprise	Community
Enterprise Lock Manager	X	-
Clustering	X	-
Hot Backups	X	-
Advanced Monitoring	X	-

- ## Capacity Planning - Hardware Sizing

  A Neo4j implementation should plan for both disk storage as well as performance scaling.

  - Performance
    - Clustering - Given the read-heavy nature of graph workloads, the Neo4j Enterprise Edition offers high availability clustering capabilities, thus

allowing for horizontal(i.e. adding instances) scaling for graph operations while scaling vertically(i.e. memory/cpu) for transaction processing.

        - Cache Sharding - For big data implementations where it is unlikely to fit all data into the memory of a single instance, cache sharding strategy can be employed to target specific branches of graph on different instances, thus allowing for an optimal distribution of load across many instances.

    - Memory - For scaling Neo4j, you need to consider the performance of the database under load and the physical volume of the graph being stored. Given the highly transactional nature of Neo4j database, the more data is fit into the memory the more likely for the transaction to be completed with minimal latency.

    - For highly concurrent or low-latency read requirements:
        - Consider hardware with higher core counts so more threads can be executing on data in parallel.
        - 2-3 bolt worker threads per core in CC is recommended - more threads per core if the transactions are very small or simple writes
        - Try to acquire hardware with more physical memory to increase the ratio of cache hits when data is queried. This will reduce the amount of time the database spends accessing data from disk.

    - For high-volume write transaction requirements:
        - Hardware with better disk speeds (or SSD) will provide higher throughput overall
        - Minimize the amount of system services contention by considering dedicated hardware for Neo4j

    - Other Guidelines
        - I/O is still a significant penalty, faster disks are better (SSD commonly used)
        - The more that fits into memory, the better
        - JVM-based database, balance memory usage between the OS and JVM

- Disk Storage - The database storage files are located in data/databases/graph.db database. These files store such attributes as nodes, relationships, properties and labels, among other things. Given the schema-less nature of the Neo4j database, the storage files uses fixed record lengths for storing data. The following table illustrates the fixed sizes Neo4j uses for the type of Java objects being stored.

Attributes	Rec Length (Bytes)	Comment
nodestore.db	15	Nodes
relationshipstore.db	34	Relationships
propertystore.db	41	Properties for nodes & relationships
propertystore.db.strings	128	Long String Properties
propertystore.db.arrays	128	Long Array Properties
Indexes	N/A	Plan 10-20% additional storage

- ## System Requirements
  - Documentation: http://neo4j.com/docs/operations-manual/current/installation/requirements/
  - Operating System –
    - Ubuntu 14.04, 16.04; Debian 8, 9; CentOS 6, 7; Fedora, Red Hat, Amazon Linux, Windows Server 2012
  - Minimum Requirements:
    - OpenJDK 8, Oracle Java 8, or IBM JDK 8
    - ext4 or similarly formatted file system
    - Minimum 10GB SATA disk (recommendation for higher perf disks)
    - 2GB Memory
    - Intel Core i3 CPU
  - Recommendations:
    - CPU - Intel Core i7 or IBM POWER8
    - Memory - 16-32GB (or more)
    - File System - ext4, ZFS
    - Disks – SSD w/ SATA

- ## Quick Start Overview:
  - Download the latest version for the platform you want.
  - Install Java If you need either Oracle JDK 8, OpenJDK 8 or IBM JDK
    - Windows Enterprise: install your preferred JDK 8

        - Windows Community: the installer has the Java you need
    - Start Neo4j
        - Windows: run the installer. Double-click and enjoy.
        - MacOS & Linux: open a terminal, cd to the extracted folder, start with: **bin/neo4j start** (background) OR **bin/neo4j console** (foreground)
    - In a web browser, open: http://localhost:7474

- ## Installation Details
    - Linux installation can use Debian, RPM packages or install from a tarball
    - For detailed installation instructions, please refer to:
        - http://neo4j.com/docs/operations-manual/current/installation/
    - Debian installation details:
        - https://neo4j.com/docs/operations-manual/3.2/installation/linux/debian/#debian
    - RPM installation details:
        - https://neo4j.com/docs/operations-manual/3.2/installation/linux/rpm/
    - Server commands
        - start - Start Neo4j server
        - stop - Stop Neo4j server
        - restart - Restart Neo4j server
        - status - Display Neo4j status
        - version - Display Neo4j version information
        - console - For Linux/MacOS, run in console, terminate server with <Ctrl-C>

- ## Directory locations & Permissions (i.e. Linux)

Documentation: https://neo4j.com/docs/operations-manual/current/configuration/file-locations/

Directories	Permissions	Description
<neo4jhome>/conf	Read Only	configuration files
<neo4jhome>/data	Read+Write	Data + Indexes + schema
<neo4jhome>/metrics	Read+Write	Metrics data
<neo4jhome>/import	Read Only	Default location for data to be ingested
<neo4jhome>/bin	Read + Execute	Binaries
<neo4jhome>/lib	Read Only	Libraries

<neo4jhome>/plugin	Read Only	Custom Code Extension
<neo4jhome>/logs	Read+Write	Log files

- ## Ports

Documentation: https://neo4j.com/docs/operations-manual/current/configuration/ports/

Name	Default port #	Related Settings	Comments
Backup	6362	dbms.backup.enabled=true dbms.backup.address=0.0.0.0:6362	Backups 6362 Backups are disabled by default. In production environments, external access to this port should be blocked by a firewall.
HTTP	7474		It is recommended to not open up this port for external access in production environments, since traffic is unencrypted. Used by the Neo4j Browser. Also used by REST API.
HTTPS	7473		Also used by REST API.
Bolt	7687		Used by Cypher Shell and by the Neo4j Browser.
Causal Cluster	5000, 6000, 7000	causal_clustering.discovery_listen_address=:5000 causal_clustering.transaction_listen_address=:6000 causal_clustering.raft_listen_address=:7000	The listed ports are the default ports in neo4j.conf. The ports are likely be different in a production installation; therefore the potential opening of ports must be modified accordingly. See also Causal Clustering settings reference
HA Cluster	5001,6001	ha.host.coordination=127.0.0.1:5001 ha.host.data=127.0.0.1:6001	The listed ports are the default ports in neo4j.conf. The ports will most likely be

			different in a production installation; therefore the potential opening of ports must be modified accordingly. See also Highly Available cluster.
Graphite monitoring	2003	metrics.graphite.server=localhost:2003 .	This is an outbound connection in order for the Neo4j database to communicate with the Graphite server. See also Metrics
JMX monitoring	3637	dbms.jvm.additional=-Dcom.sun.management.jmxremote.port=3637	This setting is for exposing the JMX. We are not promoting this way of inspecting the database. It is not enabled by default.
Neo4j-shell	1337	dbms.shell.port=1337	The neo4j-shell tool is being deprecated and it is recommended to discontinue its use. Supported tools that replace the functionality neo4j-shell are described under Tools.

● Configure Neo4j Connectors

- Neo4j supports clients using either the Bolt binary protocol or HTTP/HTTPS. Three different Neo4j connectors are configured by default. These are:

Bolt	port=7687, Connector = dbms.connector.bolt
HTTP	port=7474, Connector = dbms.connector.http
HTTPS	port=7474, Connector = dbms.connector.https

- Additional Options to consider:
  - enabled-->(default=true)
    - Allows the client connector to be enabled or disabled.

        - listen_address→(default=127.0.0.1:<connector-default-port>)
            - The address for incoming connections. Example - To listen for Bolt connections on all network interfaces (0.0.0.0) and on port 7000 -- dbms.connector.bolt.listen_address=0.0.0.0:7000

        - advertised_address→(default= localhost:<connector-default-port>)
            - The address that clients should use for this connector - useful in a **causal cluster** as it allows each server to correctly advertise addresses of the other servers in the cluster

        - tls_level→(default=OPTIONAL|disabled|required)
            - BOLT only - Allows the connector to accept encrypted and/or unencrypted connections.

    - SSL Security Certificates — By default the official drivers always encrypt all client-server communication. This applies to both Bolt and HTTP communications. The database can either use a system provided certification which will be generated upon the initial first ever startup of the database or alternatively, using a custom third party provided one. Please note the use of the system provided certification is not recommended for production use.

      Neo4j supports chained SSL certificates. All certificates need to be in the PEM format, and they must be combined into one file. The private key is also required to be in the PEM format. Multi-host and wildcard certificates are supported. Such certificates are required if Neo4j has been configured with multiple connectors that bind to different interfaces.

- ## Transaction Logs

    - Transaction logs are maintained by Neo4j and used by backup as well as cluster service. By default, log switches happen when log sizes surpass 250 MB(configured via **dbms.tx_log.rotation.size**).

    - When designing your backup strategy it is important to configure **dbms.tx_log.rotation.retention_policy** such that transaction logs are kept between incremental backups.

    - There are several different means of controlling the amount of transaction logs that is kept, using the **parameter dbms.tx_log.rotation.retention_policy**. Example:

        - # Keep transaction logs indefinitely
        - dbms.tx_log.rotation.retention_policy=true
        - # Keep the most recent non-empty log
        - dbms.tx_log.rotation.retention_policy=false

        - # keep transaction logs for last 20 days
        - dbms.tx_log.rotation.retention_policy=20 days

        - # keep most recent 20 transaction logs
        - dbms.tx_log.rotation.retention_policy=20 files

        - # keep only 1GB in transaction logs
        - dbms.tx_log.rotation.retention_policy=1GB size

        - # Keep transaction logs containing most recent 100,000 transactions
        - dbms.tx_log.rotation.retention_policy=100k txs

- ## Open File Descriptors

    - The usual default of 1024 for the open file limit is often not enough, especially when many indexes are used or a server installation sees too many connections (network sockets also count against that limit). Users are therefore encouraged to increase that limit to a realistic value of 40000 or more, depending on usage patterns.

- ## Memory Configuration

    - As noted earlier, memory based reads are always many orders of magnitude faster than reading from disks, which is why any database and/or application immensely benefits from more memory, specially in cases where the read patterns are more random than sequential. As such, the more of the graph data is fitted into the memory, the more efficient and faster the application will run. Therefore, it is recommended that the memory settings listed below are appropriately consider early on and to be planned for.

    - OS memory sizing - To avoid paging, OS memory size should not exceed available RAM. While this memory is typically utilized for running most non-Neo4j related work, it is possible that the index and schema data might also need to access it. More often not, if the database is the only application running on the system, between 1-2GB seems to be sufficient. For planning purposes, use the following formula for sizing:

- OS Memory = 1GB + (size of graph.db/index) +
  (size of graph.db/schema)

- Page cache sizing
    - This cache is used for storing the graph data in memory in order to speed up both read as well as write operations. It keeps the data in the same format as it is represented on disk. As such the goal here is to fit as much of the memory into the RAM as possible to minimize any disk access if possible.
    - The page cache also improves write performance by writing to the cache and deferring durable writes. This is safe since all transactions are always durably written to the transaction log, so Neo4j can recover the store files in the event of a crash. The write performance is also improved by batching up many small writes into fewer page-sized writes.
    - This cache size is typically set at %110 of the current size of the database files - calculated as:
        - $du -hc $<neo4jHome>/data/graph.db/*store.db*
    - Configured by setting the **dbms.memory.pagecache.size** parameter. If not set, the database will automatically use half the available RAM minus max heap size.

- Heap cache sizing
    - Heap memory sizing is highly workload dependent and for maximum concurrency as well as performance consistency, it should be adequately tested and sized for the specific application workload attributes such as number of concurrent operations, size of queries.
    - For many setups, a heap size between 8G and 16G is large enough to run Neo4j reliably, however, best to start of 8-12GB and do not size it greater than 26GB.
    - The heap memory size is determined by the configuration parameters **dbms.memory.heap.initial_size** and **dbms.memory.heap.max_size**.
    - It is recommended to set these two parameters to the same value to avoid unwanted full garbage collection pauses.

- Sanity check
    - Double check the memory settings to ensure no paging is likely to occur:
        - Actual OS allocation = available RAM - (page cache + heap size)

- Usage Data Collector
    - The Neo4j Usage Data Collector is a sub-system that gathers usage data, reporting it to the UDCserver at udc.neo4j.org.
    - Easy to disable if needed.
    - Nothing confidential is gathered - as listed below:
        - Kernel version: The build#, and if there are any modifications to kernel.
        - Store id: A randomized globally unique id created at the same time a database is created.
        - Ping count: UDC holds an internal counter which is incremented for every ping, and reset for every restart of the kernel.
        - Source: Set to “neo4j” if downloaded from the Neo4j website, otherwise, it is set to “maven”)
        - Java version
        - Registration id: For registered server instances.
        - Tags about the execution context (e.g. test, language, web-container, app-container, spring, ejb).
        - Neo4j Edition (community, enterprise).
        - A hash of the current cluster name (if any).
        - Distribution information for Linux (rpm, dpkg, unknown).
        - User-Agent header for tracking usage of REST client drivers
        - MAC address to uniquely identify instances behind firewalls.
        - The number of processors on the server.
        - The amount of memory on the server.
        - The JVM heap size.
        - The # of nodes, relationships, labels and properties in the database.

- **Network Recommendations**

- **Neo4j on Cloud**
    - https://neo4j.com/developer/guide-cloud-deployment/
    - AWS
        - https://d0.awsstatic.com/whitepapers/Database/neo4j-graph-databases-aws.pdf
        - https://vimeo.com/54142111
        - https://neo4j.com/developer/neo4j-cloud-aws-ec2-ami/
        - https://www.youtube.com/watch?v=hh0pAuQMvNQ
    - Azure
        - https://neo4j.com/blog/deploy-neo4j-microsoft-azure-part-2/
        - https://neo4j.com/users/microsoft-azure/

2. Upgrade

Upgrade Standalone - Major Release

http://neo4j.com/docs/operations-manual/current/upgrade/#upgrade-planning

- Stop Neo4j
- Always start by making a copy of the existing store (graph.db) and conf directories, after stopping the Neo4j server
- Lay down the new version
- Copy the previous instance's store (data/databases/graph.db or similar) into the new version's **data/databases** directory
- Set the following in neo4j.conf:
  dbms.allow_upgrade=true (when upgrading to 3.2 use **dbms.allow_format_migration=true**)
- Start Neo4j
- ***Note: For upgrades from pre-3.0 to 3.x, use the config-migrator tool to migrate config files***
- ***Note: For a cluster upgrade, refer to a subsequent slide!***

Upgrade Standalone - Patch Upgrade

- Read this:
  http://neo4j.com/docs/operations-manual/current/upgrade/deployment-upgrading/ - upgrade-instructions-3x
- Stop Neo4j
- Always start by making a copy of the existing store (graph.db) and conf directories, after stopping the Neo4j server
- Lay down the new version
- Copy contents of /conf and /plugins directories over to new version
- Copy the previous instance's store (data/databases/graph.db or similar) into the new version's /data/databases directory
- Start Neo4j

Upgrade HA Cluster - Major upgrade

- https://support.neo4j.com/hc/en-us/articles/218142268-Upgrading-a-Neo4j-Cluster-with-Minimal-Downtime
- http://neo4j.com/docs/operations-manual/current/upgrade/high-availability/
- Step 0: Backup the master data store!
- Stop the database on each instance, one at a time
- Lay down the new version on the previous master
- Update conf/neo4j.conf, setting dbms.allow_upgrade=true (when upgrading to 3.2 use dbms.allow_format_migration=true)
- Update conf/neo4j.conf, and set it to non-HA mode:
  dbms.mode=SINGLE
- Start the database, which will automatically upgrade the store
- Stop the database, set it back to HA mode
- For each slave, remove the data store directory, and install the new version
- NOTE: If your data store is large, you can copy the upgraded store from the master to each slave
- Now start the master, then start each slave

Upgrade HA Cluster - Patch Upgrade

- Step 0: Backup the master data store!
- Stop the database on each node, one at a time
- Install the new version on the previous master and slave instances, leaving the data store in place
- Start the master database, then each slave
- Rolling Upgrade:
- Test this in pre-prod cluster first to make sure there are no unforeseen issues with transaction log compatibility!
- Upgrade the slaves first
- Then upgrade the master
- We do testing to make sure MOST patch releases will allow cluster communication to continue, and so you can achieve a rolling upgrade for patch releases only.

Upgrading your Causal Cluster from 3.1.x to 3.2.x

This article outlines possible steps to upgrade your Neo4j 3.1.2+ Causal Cluster to 3.2.2. For this upgrade path, Neo4j does not support rolling upgrades, so downtime is required to complete the process. However, the following procedure tries to minimize the downtime for both, reads and writes while retaining data safety/high availability where possible. This trade-off may not be the best for all setups.

Assumptions

- You have a working Neo4j Causal Cluster in Version 3.1.x where x>=2.
- You have three core servers and no followers.
- Your operating system is Linux.
- The installation of Neo4j has been done by extracting the .tar.gz file.
- You have licensed multi-data center operations, see http://neo4j.com/docs/operations-manual/current/clustering/causal-clustering/multi-data-center/#multi-dc-licensing.
- Your configuration, plugins and files in data/dbms/ are not changed while following the upgrade steps of this article (except for the adjustments described in the steps itself, of course).
- Your free disk space is at least double the size of your current installation including your database files.

Upgrade Steps

Before following the below steps, read the complete article thoroughly and test the procedure in a pre-production environment.

Step 0 - Preparation

- Make sure you have access to the installation source of Neo4j 3.2.2.
- Make sure your client applications have a good error handling in place, since they will receive execeptions during a short period of time. E.g., when using the Java driver, you may want to configure it with `Config.build().withMaxTransactionRetryTime( .. ).toConfig()`.

● Make sure your client applications differentiate between read and write transactions. I.e., whenever you only read, use `session.readTransaction()` or set the access mode of your session to `AccessMode.READ`.

● Make sure that all your plugins are compatible with the new version.

● Make a backup of your database.

Step 1 - Install Neo4j 3.2.2

On each server:

○ Extract the neo4j-enterprise-3.2.2-unix.tar.gz file

○ Copy all files from the following directories from your old (and running) Neo4j insallation to the newly extracted (and still shutdown) installation (retain the directory structure):

○ files in conf/

○ files in data/dbms/

○ files in plugins/

In the following, we are referring to `old` and `new` files or instances. The old files/instances are those of the Neo4j 3.1 installation, the new ones are those of the Neo4j 3.2 installation.

Step 2 - Set Followers as "Follower Only"

○ Identify Leader and Follower instances of your old cluster installation. You may use one of the following commands:

○ `:sysinfo` in the Neo4j Browser or

○ CALL `dbms.cluster.overview()` via Cypher

○ Pick a Follower instance that is not already in "Follower Only" mode:

○ In the old neo4j.conf file of that instance

○ set/add `causal_clustering.refuse_to_be_leader=true` and

○ set/add `causal_clustering.multi_dc_license=true`.

○ Restart the instance, which will not be able to become a Leader any longer.

○ Wait until this instance is up and running again.

○ Repeat Step 2 until all Followers (in our szenario two) are in "Follower Only" mode.

During this step you may encounter a short (a few ms) delay in processing read requests. However, no downtime is expected if you have configured your application to retry transactions (see Step 0). After this step, we have a three instances cluster where only one instance is allowed to be the Leader. This reduces the high-availability for your writes. E.g., if the Leader now fails, reads are still processed, but no writes are possible.

Step 3 - Shutdown Leader

○ Stop the old Leader. This step starts the downtime for writes! Reads are still processed through the two Follower instances.

Step 4 - Copy database to the New Leader

○ On the old Leader instance:

○ Copy your (old) active database folder (by default: data/databases/graph.db) to the new 3.2.x installation (which has been setup in step 1). By copying the database folder, we still have the latest writes available for fallback.

Step 5 - Configure 3.2.x for the Format Migration

○ On the Leader box in the new 3.2.x neo4j.conf file, make sure to have the following parameters set:

○ dbms.allow_upgrade=true (when upgrading to 3.2 use dbms.allow_format_migration=true)

○ dbms.mode=SINGLE

Step 6 - Upgrade the database files (Format Migration)

○ On the Leader box in the new installation:

○ Start up Neo4j 3.2.x. Depending on the size of your database, this step may take a few moments to finish.

○ Check your logs/neo4j.log file for success. It should have the following line:

```
2017-07-06 18:08:59.933+0000 INFO Successfully finished upgrade of database
```

○ Stop Neo4j 3.2.x. We now have a upgraded database which we use to seed the new cluster.

Step 7 - Revert Configuration from Step 5

○ On the Leader box in the new conf/neo4j.conf file, make sure to have the following parameters set:

○ dbms.allow_upgrade=false (when upgrading to 3.2 use dbms.allow_format_migration=false)

○ dbms.mode=CORE

Step 8 - Seed the Cluster

○ Copy the newly upgraded database folder (by default: data/databases/graph.db) to the new installation of the Follower instances. I.e.:

○ Copy your new active database folder from the Leader box to one of the new Follower installations.

○ Copy your new active database folder from the Leader box to the other new Follower installation.

Step 9 - Switch Versions

Starting with one of the old Follower instances:

○ Stop the instance.

Repeat this process for the second Follower instance. This is the beginning of the downtime for reads!

○ Now, start up the new cluster:

○ Start the new Leader instance.

○ Start the two new Follower instances.

This procedure stops the downtime for reads and writes.

Step 10 - Validate

Validate that your cluster is healthy. CALL `dbms.cluster.overview()` may be used as a starting point. If you still have the Neo4j Browser open in a web browser, you may need to refresh the site.

Step 11 - Backup

● Take a full backup of your running database including the consistency check to validate the upgraded database.

Step 12 - Remove 3.1.x

● Remove your old installation files on all of your three servers.

Upgrading your Causal Cluster from 3.2.1 to 3.x.x

It turns out that there is nothing preventing a rolling upgrade of a Causal Cluster from any version 3.2.1 or newer to a newer version. For example, 3.2.2 to 3.3.1 is doable in a rolling fashion.

- Upgrade each Read Replica first.
- Next each Follower
- Lastly, the Leader
- Note: This is the recommended order to limit failovers to the least possible, 1.

Fallback Scenarios

The following sections describe the steps to revert your setup to its original state. Depending on when you decide to do the fallback, you will jump into one of the following sections and execute the steps described in there.

Coming from Step 1 above

● Remove the new installation files from all servers.

Coming from Step 2 above

● Follow the steps from section 'Coming from Step 1 above'.

Coming from Step 3 above

Starting with one of the old Follower instances:

● In the old neo4j.conf file of that instance

○ set causal_clustering.refuse_to_be_leader=false or remove that parameter completely and

○ revert causal_clustering.multi_dc_license to its originally configured value.

● Restart the (old) Follower instance.

- Wait until this instance is up and running again. > Repeat this process for the second Follower instance.
- Follow the steps from section 'Coming from Step 1 above'.

Coming from Step 4, 5, 6, 7 or 8 above

- Start the old Leader

This step forms the end of the downtime for writes!

- Follow the steps from section 'Coming from Step 3 above'.

Coming from Step 9 or 10 above

Be aware that you will lose writes, that have been committed to the new cluster when doing the following!

- Stop the new Followers.
- Stop the new Leader.
- Start the old Leader.
- Follow the steps from section 'Coming from Step 3 above'.

3. Causal Clustering

Why Causal Cluster:

1. **Safety**: Core Servers provide a fault tolerant platform for transaction processing which will remain available while a simple majority of those Core Servers are functioning.
2. **Scale**: Read Replicas provide a massively scalable platform for graph queries that enables very large graph workloads to be executed in a widely distributed topology.
3. **Causal consistency**: when invoked, a client application is guaranteed to read at least its own writes.

Detailed Causal Clustering Information is provided in the Documentation:
https://neo4j.com/docs/operations-manual/current/clustering/causal-clustering/introduction/

Spin up Causal Cluster on AWS:

- Spin up 3 AWS EC2 instances and Install Neo4j on them.
- Following Ports needs to be opened on the EC2 instances for cluster and outside communication:

7474	http
7473	https
7687	bolt
5000	listen discovery port
6000	transaction listen port
7000	raft listen port
6362	backup
3637	Remote JMX

- Neo4j.conf file configuration on AWS EC2 instance:

  Set the default_listen_address as shown below so one can remotely access the neo4j browser.

  dbms.connectors.default_listen_address=0.0.0.0

  Set the default_advertised_address to Public IP address of the AWS EC2 instance

  dbms.connectors.default_advertised_address=34.205.172.119

  Set the dbms.mode to CORE or READ_REPLICA based on the instance

  dbms.mode=CORE

  Set the expected number of Cores

  causal_clustering.expected_core_cluster_size=3

  Address and port that this machine advertises that it's RAFT server is listening at. Make sure to set this to Internal IP address of the AWS EC2 instance.

  causal_clustering.raft_advertised_address=10.0.0.6:7000

Address and port that this machine advertises that its transaction shipping server is listening at. Make sure to set this to Internal IP address of the AWS EC2 instance.

causal_clustering.transaction_advertised_address=10.0.0.6:6000

FAQs (Work in Progress)

Q1. I am seeing frequent leader elections in my causal cluster. What should I look for and how can I prevent this?

A1. Look for GC and if its higher than 7 seconds perhaps change the Leader_election_timeout to more than 7 seconds.

Q2. One of the Core Follower was offline for sometime. Now it joins but fails to catchup and times out?

A2. increase the join timeout

Causal Cluster VS High Availability:

Causal Cluster	High Availability
Raft Protocol	Paxos Protocol
Minimum of 3 instances	Minimum of 2 instances + Arbiter
Set of Cores (3 or 5 or 7) + Read Replicas	Master(1) + Slaves (1 or more)
Transactions are committed, once a majority of Core Servers in a cluster (N/2+1) have	Transactions are committed on Master instance and optimistically pushed to Slave.

accepted the transaction	(does not guarantee commits on Slaves).
No Branching	This can lead to branching. (https://neo4j.com/docs/operations-manual/current/clustering/high-availability/architecture/#_branching)
Bolt driver has load balancer built it that works with Causal Cluster. Driver maintains Routing table that has information of which is Core Leader and who are Core Followers and Read Replicas	Need a Load Balancer to work with HA.

4. Drivers

- Neo4j offers official support for Java, C#, Python and JavaScript drivers, as well as community drivers for Ruby, PHP, R, Go and others. The Neo4j community also has support for popular frameworks like Spring Data, Django ORM, Laravel, JDBC and more. There are also integrations for other databases and analytics tools, like MongoDB, Cassandra, ElasticSearch and Spark/GraphX.

- The Driver Project
  - Hosted on Github
  - Apache Project
  - Versioned and Released independent of Neo4j
  - One Driver can support multiple server versions
  - The driver API is topologically agnostic
  - Published to Maven Central, PyPI, etc
  - Quarterly Releases (on Average)

- Driver Release Versions

    - The 1.x.x (i.e. 1.4.3) series drivers have been built for Neo4j 3.x.
        - The driver major version#(i.e. **1**.x.x) correlates with the Bolt protocol version.
        - The driver minor version#(i.e. 1.**4**.x) describes the driver feature set.
        - The driver patch#(i.e 1.4.**3**) defines the patch level of the driver.

    - The 1.x series drivers are designed to work with 3.x series servers
        - 1.0 – Bolt Direct
        - 1.1 – Bolt Routing for Causal Clustering
        - 1.2 – Transaction Functions
        - 1.3 – Routing Context
        - 1.4 – Byte Arrays and Multiple Bookmarks
        - 1.5 – Async(.Net and Java) and connection management

    - The 2.x series will be aligned with 4.x series servers

    - Always use the latest driver release within a major series. To find out the latest version of the driver, visit
        - https://www.nuget.org/packages/Neo4j.Driver/
- Official Drivers

    - Native API drivers are available for Java, Python,.Net as well as Javascript
    - The driver API is topologically agnostic
        - That is the underlying database topology(i.e. single instance, causal cluster, etc) can be altered without requiring a application code change.
        - Only the connection URI needs to be modified when changing topolog
    - The official drivers do not support HTTP communication, however, there are a number of community drivers that also allow for HTTP access.

- Driver Configuration Options
        - Security(Encryption, Trust)
        - Connection Management(Pool size, socket options, timeout)
        - Routing (load balancing, retries)

- Java
    - To enable Async in Java, Java driver 1.5+ will only be supported in Java 8 only
    - The 1.4 Java driver will only be supported for Java 7 – will patch and backport as needed

Language	**Versions supported**	**Notes**	Async

Java	* Works with any JVM language Oracle JDK 7/8 and OpenJDK 7/8- Async also supported on version 8 and higher on 3.2.3+	<**dependency**> <**groupId**>org.neo4j.driver</**groupId**> <**artifactId**>neo4j-java-driver</**artifactId**> <**version**>x.y.x</**version**> **</dependency>**	yes
Python	CPython 2.7, 3.4, 3.5 and 3.6	pip install neo4j-driver	No
Javascript	built to work with all LTS versions of Node.JS, specifically the 4.x and 6.x series runtimes (see https://github.com/nodejs/LTS)	npm install neo4j-driver	Yes
.NET	* Works with any .NET language * Uses the .NET standard 1.3 (https://github.com/dotnet/standard/blob/master/docs/versions.md)	PM> Install-Package neo4j-driver	Yes

- Driver Core API Components:
  - Driver Object - Top Level Object for all Neo4j interaction
  - Sessions - Logical context for sequence of transactions
  - Transactions - Basic unit of work
  - Results - Stream of data and meta-data

- Driver Object
  - A Neo4j client application requires a driver object instance in order to provide access to the database.
  - Thread-safe
  - Constructed on startup and destroyed on exit. Destroying a driver instance will immediately shut down any connections(including connection pools) previously opened via that driver.
  - To construct a driver instance, a connection URI and authentication data must be supplied.

  - The driver object lifecycle

```
public class DriverLifecycleExample : Idisposable
{
        public IDriver Driver { get; }
        public DriverLifecycleExample(string uri, string user, string
password)
        {
        Driver = GraphDatabase.Driver(uri, AuthTokens.Basic(user,
password));
        }
        public void Dispose()
        {
                Driver.Dispose();
        }
}
```

- The driver API is topologically agnostic – Run the same code against cluster or single instance
- Only the connection URI needs to be modified when changing topology.
    - Driver URI example
    - Direct Driver(Bolt)
        - bolt://localhost:7687
    - Routing Driver
        - bolt+routing://graph.example.com:7687
        - The initial connection to the cluster needs to be to a Core instance. In the example above, **graph.example.com** should be a DNS entry that resolves to *all of the initial core members* of the cluster. Unless it knows about all cores which might contain a routing table, then you run the risk of the driver not knowing how to reach the rest of the cluster in the event that the core instance it is connected to initially fails.
    - Routing Driver with Routing Context (Drivers 1.3+/Servers 3.2+)
        - bolt+routing://graph.example.com:7687/?region=europe&country=sw
        - The driver communicates the routing context to the cluster. It then obtains refined routing information back from the cluster, based on information suggested in the routing context. In the example above, the driver communicates that it prefers to route queries to the servers located in the country Sweden in the region Europe.
    - A prerequisite for using a Routing driver with routing context is that the Neo4j database is operated on a Causal Cluster with the Multi-data center licensing option enabled. Additionally, the routing contexts must be defined within the cluster as *routing policies*.

Best Practice: Bolt+Routing

Example: bolt+routing://graph.example.com:7687

The initial driver connection to the cluster needs to be to a Core instance. In the example above, **graph.example.com** should be a DNS entry that resolves to *all of the initial core members* of the cluster. This can be handled in DNS, a proxy, load balancer, or anything else that will allow a single hostname or IP address to resolve to all hostnames or IP addresses of the initial core members of the cluster.

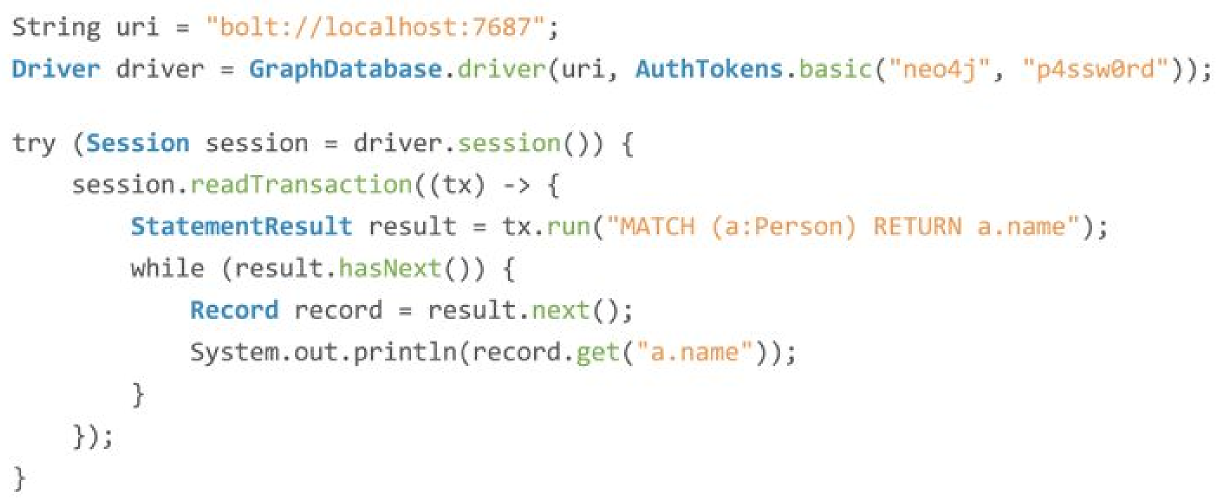
- Sessions
    - A session is a short lived thread-unsafe container for a sequence of transactions.
    - Sessions borrow connections from a pool as required and so should be considered lightweight and disposable.
    - Single-threaded and maybe backed by one or more tcp connections(or none if idle)
    - Sessions <> connections
        - Connections
            - Managed internally and not directly exposed to the user
            - Acquired/released by sessions as required
            - Belong to a connection pool owned by a driver object
            - Can only be explicitly closed via driver.close()

- Transactions
    - Auto-commit Transactions
        - Consists of only one Cypher statement, cannot be automatically replayed on failure, and cannot take part in a causal chain.
        - Not recommended to use auto-commit transactions in production or when performance or resilience are a primary concern
            - Ex: session.Run("CREATE (a:Person {name: $name})", new {name});

    - Transaction Functions:
        - Recommended form for containing transactional unit of work
        - Handle connection problems &transient errors via auto retry mechanism(configured on Driver construct)
        - Query results obtained within a function will need to be consumed within the function
        - Returned values need to be derived values rather than raw results
            - Example: session.WriteTransaction(tx => tx.Run("CREATE (a:Person {name: $name})", new {name}));

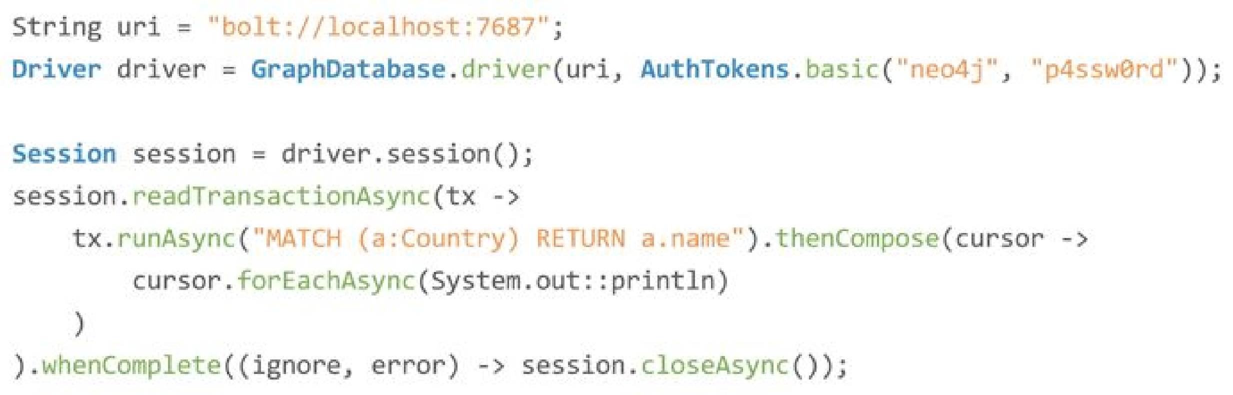
- Explicit Transactions:
  - Explicit transactions are the longhand form of transaction functions, providing access to explicit BEGIN,COMMIT and ROLLBACK operations.
  - While this form is useful for a handful of use cases, it is recommended to use transaction functions wherever possible.
    - Example: Try =(Transaction tx = session.beginTransaction()) {tx.run(("CREATE (a:Person {name: ‘Alice’})”);
- Results/records/values

- Examples:
  - Java(Blocking)



```
String uri = "bolt://localhost:7687";
Driver driver = GraphDatabase.driver(uri, AuthTokens.basic("neo4j", "p4ssw0rd"));

try (Session session = driver.session()) {
    session.readTransaction((tx) -> {
        StatementResult result = tx.run("MATCH (a:Person) RETURN a.name");
        while (result.hasNext()) {
            Record record = result.next();
            System.out.println(record.get("a.name"));
        }
    });
}
```

  - Java(Aysnc)



```
String uri = "bolt://localhost:7687";
Driver driver = GraphDatabase.driver(uri, AuthTokens.basic("neo4j", "p4ssw0rd"));

Session session = driver.session();
session.readTransactionAsync(tx ->
    tx.runAsync("MATCH (a:Country) RETURN a.name").thenCompose(cursor ->
        cursor.forEachAsync(System.out::println)
    )
).whenComplete((ignore, error) -> session.closeAsync());
```

  - Python(Blocking)

```
uri = "bolt://localhost:7687"
driver = GraphDatabase.driver(uri, auth=("neo4j", "p4ssw0rd"))

def print_names(tx):
    result = tx.run("MATCH (a:Person) RETURN a.name")
    for record in result:
        print(record["a.name"])

with driver.session() as session:
    session.read_transaction(print_names)
```

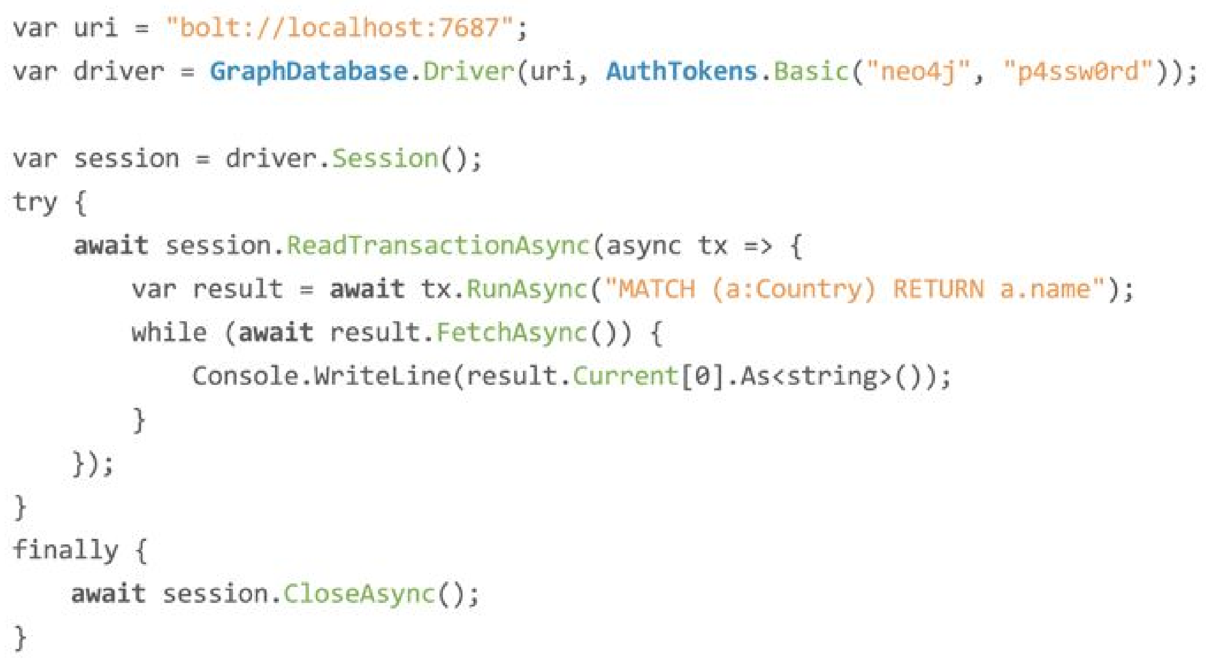
- Javascript(Async)

```
const neo4j = require('neo4j-driver').v1;
const uri = 'bolt://localhost:7687';
const driver = neo4j.driver(uri, neo4j.auth.basic('neo4j', 'p4ssw0rd'));

const session = driver.session();
const resultPromise = session.readTransaction(tx =>
    tx.run('MATCH (a:Person) RETURN a.name')
);
resultPromise.then(result => {
    result.records.forEach(record => console.log(record));
    session.close();
});
```

  - C#(Blocking)

```
var uri = "bolt://localhost:7687";
var driver = GraphDatabase.Driver(uri, AuthTokens.Basic("neo4j", "p4ssw0rd"));

using (var session = driver.Session()) {
    session.ReadTransaction(tx => {
        var result = tx.Run("MATCH (a:Person) RETURN a.name");
        foreach (var record in result) {
            Console.WriteLine(record["a.name"]);
        }
    });
}
```

  - C#(Async)



```
var uri = "bolt://localhost:7687";
var driver = GraphDatabase.Driver(uri, AuthTokens.Basic("neo4j", "p4ssw0rd"));

var session = driver.Session();
try {
    await session.ReadTransactionAsync(async tx => {
        var result = await tx.RunAsync("MATCH (a:Country) RETURN a.name");
        while (await result.FetchAsync()) {
            Console.WriteLine(result.Current[0].As<string>());
        }
    });
}
finally {
    await session.CloseAsync();
}
```

- **Passing Bookmarks** - Using a Bolt driver with a Causal Cluster provides causal consistency for the application, meaning the application is guaranteed to always have a consistent view of related data.

  This is exposed to the application through bookmarking.   After carrying out work in a session, a bookmark is created, which is essentially a pointer to the work. As a result, this bookmark should be passed to other new sessions which are later created for other work.   This ensures that the dependency of this later work on the previous is respected.

  Whether or not the data has been replicated to every corner of the cluster, the bookmark guarantees that whichever instance services the next session will be up to date with previous work.

  It is possible both to construct a session based on one or more existing bookmarks, and to extract a bookmark from a session. This functionality allows multiple sessions to be chained together.

  When running multiple independent transactions, it is recommended to use separate sessions. This avoids the small latency overhead of the causal chain.

  Example: Create a bookmark

  ```
  Driver driver = GraphDatabase.driver(seedUri, Authtokens.basic("neo4j", "neo4j"));
  String bookmark;
  try (Session session = driver.session(AccessMode.WRITE))
  ```

```
{
        try (Transaction tx = session.beginTransaction())
        {
                tx.run("CREATE (person:Person {name: {name}, title: {title}})",
                parameters("name", "Arthur", "title", "king")); tx.success(); tx.close();
        }
                finally
        {
                bookmark = session.lastBookmark();
        }
}
return bookmark;
```

- Example: Use a Bookmark

```
try ( Session session = driver.session( AccessMode.READ ) )
{
        StatementResult result;
        try ( Transaction tx = session.beginTransaction( bookmark ) )
        {
                result = tx.run( "MATCH (person:Person {name: {name}})
                        RETURN    person.title",   parameters( "name", "Arthur" ) );
                tx.success();
                tx.close();
                while ( result.hasNext() )
                {
                        Record record = result.next();   System.out.println( "Arthur's title is
                        " + record.get( "person.title" ) );
                }
        }
}
```

```
● Driver driver = GraphDatabase.driver(seedUri, Authtokens.basic("neo4j",
  "neo4j"));
● String bookmark;
● try (Session session = driver.session(AccessMode.WRITE)) {
●     try (Transaction tx = session.beginTransaction())  { tx.run("CREATE
  (person:Person {name: {name}, title: {title}})",  parameters("name", "Arthur",
  "title", "king")); tx.success(); tx.close();
●     }
●     finally  { bookmark = session.lastBookmark();
●     }
● }
● return bookmark;
```

- Bolt vs. Bolt+routing
    - Bolt is mainly used for Single instance environments, legacy HA environments, and situations where a client needs to connect to a very specific cluster instance, or for dev and testing.
    - Bolt+routing is used in Causal Clustering and allows the client all the benefits of the Causal Cluster.
        - Example: bolt+routing://graph.example.com:7687
        - Graph.example.com should be a DNS entry, shared hostname, etc. which resolves to the hostname or IP address of all of the cores in the cluster, or at least the initial core cluster members (recommended minimum of 3).
- HTTP API Examples
    - The only HTTP API currently recommended is the transactional endpoint. All others have been deprecated.
    - See docs: https://neo4j.com/docs/developer-manual/current/http-api/#http-api-transactional
    - It is most common to use implicit transactions, or, one set of query statements per transaction, invoked by a single HTTP request.
        - Endpoint: **http://localhost:7474/db/data/transaction/commit**
    - If you need to send multiple requests/queries as part of the same logical piece of work (i.e. transaction), then you may choose to use the explicit endpoint. Please note the following gotchas and recommendations:
        - Always commit EVERY transaction, even reads. Otherwise, the transaction will wait for the idle transaction timeout, and then be rolled back. This impacts the length of time that transaction holds onto any acquired locks, memory resources, and occupies a worker thread.
        - If you don't have enough worker threads to handle the workload, then the commit messages may not get through in a timely fashion, and open transactions will accumulate until they start timing out and rolling back.
        - You should configure any load balancer or proxy between Neo4j and the client to use sticky sessions, since not all cluster members will know about the transaction.
        - To use this, you will first make a request to the transaction endpoint:
            - http://localhost:7474/db/data/transaction
        - This returns a transaction ID. Your next request would then use that ID to send a request to that same transaction. For example, if you got ID 1234 back from the first request, the second would go to:
            - http://localhost:7474/db/data/transaction/1234
        - Finally, when the transaction can be committed, send a final request, and add the commit URI:
            - http://localhost:7474/db/data/transaction/1234/commit
    - The number of concurrent HTTP requests the server can handle is controlled by the **dbms.threads.worker_count** setting, which defaults to one thread per logical CPU core.

        - You may increase this incrementally to find out whether you can use more threads without sacrificing performance per query.
        - This setting is the total number of jetty threads the server should keep for HTTP requests. Thus, set it to (threads per core) x (logical cores).
            - Example: If you want to use 3 threads per core, and have 4 cores, set it to 12.

5. Backup/Restore/Consistency Check

- Backup
  - http://neo4j.com/docs/operations-manual/current/backup/perform-backup/
  - Online Backup is only available in Neo4j Enterprise (server and embedded)
  - Online backup (enabled by default: dbms.backup.enabled)
  - Full backup:
    - Specify an empty target directory, or one that does not already contain a valid store file.
    - By default, automatically runs consistency check against the backed-up store.
  - Incremental backup:
    - If the target directory contains a valid store, it will automatically capture an incremental backup.
    - Relies on historical logical logs, so if those are already purged, it will take a new full backup. If you have problems with logs not being available, you can increase the dbms.tx_log.rotation.retention_policy setting in neo4j.conf.
  - Backup Command:

```
$neo4j-admin backup
        --backup-dir=<backup-path>
        --name=<graph.db-backup>
        [--from=<address>]
        [--fallback-to-full[=<true|false>]]
        [--timeout=<timeout>]
        [--check-consistency[=<true|false>]]
        [--cc-report-dir=<directory>]
        [--additional-config=<config-file-path>]
        [--cc-graph[=<true|false>]]
        [--cc-indexes[=<true|false>]]
        [--cc-label-scan-store[=<true|false>]]
        [--cc-property-owners[=<true|false>]]
```

  - Make sure the memory settings are configured properly!
    - **neo4j-admin backup** uses the conf/neo4j.conf settings for heap and page cache by default. This is a potential problem on a Neo4j server that has explicit heap and page cache settings. These should be set per the following KB article:
    - https://support.neo4j.com/hc/en-us/articles/115013375988-Understanding-memory-configurations-for-neo4j-admin-backup

    - Environment variables:
        - NEO4J_CONF Path to directory which contains neo4j.conf.
        - NEO4J_DEBUG Set to anything to enable debug output.
        - NEO4J_HOME Neo4j home directory.
        - HEAP_SIZE Set size of JVM heap during command execution.
            - Takes a number and a unit, for example 512m.

- ## Restore
    - http://neo4j.com/docs/operations-manual/current/backup/restore-backup/
    - **Note:** From **Docker**, neo4j-admin restore won't work, since you can't stop Neo4j. Instead, simply stop the container, replace the graph.db with the backup you want to restore, and then start the container again.
    - neo4j-admin restore

      --from=<backup-directory>
      [--database=<name>]
      [--force[=<true|false>]]
        - options:

          --from=<backup-directory> Path to backup to restore from.
          --database=<name> Name of database. [default:graph.db]
          --force=<true|false> If an existing database should be replaced. [default:false]

- ## Consistency Check
    - The Consistency Check is run automatically during any backup.
    - For very large stores, or stores with very large indexes, this can be quite slow. If this is an issue for you, you are welcome to skip it on backup, and run it later on manually
        - You may want to try just skipping the index checking to start with.
        - --check-indexes=false
    - To skip during a backup, use --check-consistency=false
    - If you experience an error or suspect an issue with your data or indexes, it is a good idea to run a manual consistency check against your data store:
    - This must be done OFFLINE. So, if this is production, simply take a full backup
    - If this is pre-production or test, you can use the following to run the manual check after stopping the database:
        - http://neo4j.com/docs/operations-manual/current/tools/consistency-checker/
    - ### Usage
    - $neo4j-admin check-consistency
        - [--database=<name>]

        - [--backup=</path/to/backup>]
        - [--verbose[=<true|false>]]
        - [--report-dir=<directory>]
        - [--additional-config=<config-file-path>]
        - [--check-graph[=<true|false>]]
        - [--check-indexes[=<true|false>]]
        - [--check-label-scan-store[=<true|false>]]
        - [--check-property-owners[=<true|false>]]

    - Optional Manual Checks:
        - --check-graph (default=True)
            - Perform checks between nodes, relationships, properties, types and tokens.
        - --check-indexes(default=True)
            - Perform checks on indexes.
        - --check-label-scan-store (default=True)
            - Perform checks on the label scan store.
        - --check-property-owners (default=True)
            - Perform additional checks on property ownership. This check is very expensive in time and memory.

    - Rare Consistency Errors:
        - Error: Inconsistent Counts in the metadata
            - Indicative of the count metadata at either the node or relationship corresponding to the actual number of nodes or relationships
            - Resolution: Rebuild graph.db/neostore.counts.db.*
        - Error: Inconsistent Counts among indexes
            - Indicative of the schema indexes not being representative of the actual da ta
            - Resolution: Rebuild graph.db/schema/index
        - Error: Inconsistent Labels Scan Store
            - Indicative of the schema label store not being representative of actual data
            - Resolution: Rebuild graph.db/schema/label
        - Error: Node not found in the expected index
            - Indicative of a node that does not exist in an index
            - Resolution: Rebuild graph.db/schema/index
        - Error: another node in the unique index with the same property value

            - indicative of an index, as a result of a UNIQUE CONSTRAINT, recording a value not the same as the node it originates from
            - Resolution: Rebuild graph.db/schema/index
        - Error: LabelScanStore mismatches
            - Indicative of the labelscantore referencing nodes which are no longer in use
            - Resolution: Rebuild graph.db/neostore.labelscanstore.db and restarting.

- ## Best Practices
    - Typical backup strategy always depends on how much data you can tolerate losing in case of catastrophic failure.
        - Most observed pattern consists of daily full backups coupled taking incremental backups several times a day while ensuring the Neo4j log files are kept for longer than this period.
            - Example:Daily full backups + hourly incrementals, while retaining 7 days of logs ((i.e. dbms.tx_log.rotation.retention_policy=7 days)
    - Ideally backup should be done from remote machine so not to compete for machine resources.
        - Make sure to set dbms.backup.address=0.0.0.0:port to enable remote backup connection
    - On AWS, recommended to perform backups with an I/O and RAM-optimized EC2 instance and followed by moving the backups to Amazon S3 (or to Amazon EBS).
    - Note, if neo4j-admin backup is run on the same server as the Neo4j instance, by default it will use the pagecache settings from the neo4j.conf file. This has the potential to cause out of memory issues if the pagecache for the production instance is quite large. The recommended approach is:
        - To make a copy of the production neo4j.conf file, and comment out, **#dbms.memory.pagecache.size**
        - This will then use heuristics to decide on pagecache for the backup instance of neo4j and not cause memory issues.
        - Once the neo4j.conf is created, copy it to a location and make sure to set the NEO4J_CONF environment variable to point to the location of this file.
        - You can also set the HEAP_SIZE environment variable for the backup process to use:

```
export NEO4J_CONF=/tmp/conf
export HEAP_SIZE=512M
```

6. Security

- ## Authentication

  - By default, Neo4j requires clients to supply authentication credentials when accessing the REST API. Without valid credentials, access to the database will be forbidden.
  - The authentication and authorization data is stored under data/dbms/auth. If necessary, this file can be copied over to other neo4j instances to ensure they share the same username/password
  - Authentication is enabled by default and must be explicitly disabled if desired. Doing so will allow any client to access the database without supplying authentication credentials.
  - Authentication can be disabled by uncommenting the below parameter in the neo4j.conf file:
    - # To disable authentication, uncomment this line:
      - dbms.security.auth_enabled=false
    - Authentication for HA REST status endpoints only can be disabled by adding the below parameter in the neo4j.conf file:
      - dbms.security.ha_status_auth_enabled=false

- ## Locking down Neo4j
  - For production deployments we recommend disabling any unused entry points to minimize risk for unauthorized access.
- ## Native Roles
  - 5 built in roles: reader, publisher, architect, editor, admin
  - https://neo4j.com/docs/operations-manual/current/security/authentication-authorization/native-user-role-management/native-roles/

- ## Custom Roles
  - Roles can be created by admin. Used for the sole purpose of executing custom stored procedures
- ## Managing Native Users/Roles
  - Done via built in procedures:

    - https://neo4j.com/docs/operations-manual/current/security/authentication-authorization/native-user-role-management/procedures/

- LDAP Integration
    - https://neo4j.com/docs/operations-manual/current/security/authentication-authorization/ldap-integration/
    - Works with both openLDAP and Active Directory
    - **How do I allow for authentication using Active Directory attribute samAccountName:** https://support.neo4j.com/hc/en-us/articles/115013527168
    - Key Configuration Parameters:

        - Dbms.security.ldap.authentication.user_dn_template
        - Dbms.security.ldap.authorization.user_search_base
        - Dbms.security.ldap.authorization.user_search_filter
        - Dbms.security.ldap.authorization.group_membership_attributes
        - Dbms.security.ldap.authorization.group_to_role_mapping
        - dbms.security.ldap.authentication.use_samaccountname=true (3.2.2+)
        - Dbms.security.ldap.authorization.system_username
        - dbms.security.ldap.authorization.system_password

    - Leverage ldapsearch to verify settings:

        - ms.security.ldap.host> -x -D
        - <dbms.security.ldap.authentication.user_dn_template : replace {0}> -W -b
        - <dbms.security.ldap.authorization.user_search_base>
        - "<dbms.security.ldap.authorization.user_search_filter : replace {0}>"
        - <dbms.security.ldap.authorization.group_membership_attributes>

        - ldapsearch -v -H ldap://myactivedirectory.example.com:389 -x -D cn=john,cn=Users,dc=example,dc=com -W -b cn=Users,dc=example,dc=com "(&(objectClass=*)(cn=john))" memberOf

    - Security.log helpful for diagnosing issues.

- Unified SSL Framework

As of 3.2.2, Neo4j added a Unified SSL framework to enable and configure encryption both for client/server communication and to encrypt communication between servers in a Causal Cluster (Intra Cluster Encryption).

This Unified SSL Framework is well documented at:

https://neo4j.com/docs/operations-manual/current/security/ssl-framework/

- Key SSL Framework settings/concepts
  - You can have multiple SSL Policies (one for client/server communication and one for Intra-Cluster Encryption)
  - The Policy name is derived from the configuration setting:
    - `dbms.ssl.policy.<policy-name>.base_directory`
  - Intra-Cluster Encryption is for Causal Cluster only and not for HA
  - Intra-Cluster Encryption supports Mutual Authentication and is configured with the following settings:
    - `dbms.ssl.policy.<policy-name>.trusted_dir`
    - `dbms.ssl.policy.<policy-name>.revoked_dir`
    - `dbms.ssl.policy.<policy-name>.client_auth=REQUIRE`
  - To specify that the protocol uses the defined policy, use the following settings:
    - `causal_clustering.ssl_policy=<policy-name>`
    - `bolt.ssl_policy=<policy-name>`
    - `https.ssl_policy=<policy-name>`
  - The folder specified with the base_directory setting must be created on the server under the certificates folder (or custom name defined by dbms.directories.certificates)
  - The trusted and revoked folders **MUST** be created under the above folder even if mutual authentication is not enabled
  - To enforce bolt driver connections use SSL, set the following:
    - `dbms.connector.bolt.tls_level=REQUIRED`

Best Practice: Turning off Mutual Authentication

Mutual Authentication is currently only supported for Intra-Cluster encryption and not for driver client/server communication.

When a SSL Policy is configured, by default the

```
dbms.ssl.policy.<policy-name>.client_auth
```

setting is set to **REQUIRED**. This will cause driver (including Neo4j Browser and cypher-shell) connections to fail. You must set it as follows to allow proper connection:

```
dbms.ssl.policy.<policy-name>.client_auth=none
```

7. Cypher

Cypher is a rich language with many useful clauses, functions, and constructs.

Here are some useful resources for learning, referencing, and understanding Cypher.

- http://neo4j.com/docs/developer-manual/current/cypher/
- http://neo4j.com/blog/common-confusions-cypher/
- https://neo4j.com/docs/cypher-refcard/current/
- https://neo4j.com/developer/kb/?tag=cypher

The following will only address some Cypher topics and constructs, with emphasis on best practice, performance, troubleshooting, and illuminating some of the more common stumbling blocks.

View query plans with PROFILE and EXPLAIN

By prefixing a cypher query with **PROFILE** or **EXPLAIN**, you can generate a visual query plan and additional information.

PROFILE - To run the query and produce a query plan, execution time, total db hits, and the number of rows (records) flowing to and from each operation.

EXPLAIN - To only generate the plan (query is NOT executed) with estimated rows. Does not include execution time or total db hits.

Cardinality, Cartesian Products, and Aggregations

These three topics are interrelated, and must he understood together.

- Unmanaged cardinality issues can significantly impact performance and heap usage.
- Cartesian products from MATCHes or OPTIONAL MATCHes greatly increase cardinality.
- Aggregations in the right places at the right times can help reduce cardinality and streamline query performance.
- Aggregations over cartesian products can be tricky, DISTINCT might be needed for correct results.

Manage cardinality - Operations execute per record

Cypher queries build up records as execution progresses. These records are streamed between cypher operators according to the query plan. You can also think of these as the “rows” that flow as inputs and outputs to and from cypher operators (these are even called “rows” within the query plans produced by an **EXPLAIN** or **PROFILE** of a query).

Nearly all Cypher clauses and operations execute per-record in the stream, so managing the cardinality of the built-up records of a query is essential to constructing streamlined queries.

The term “cardinality” refers to the number of times a certain value (which could be a node, relationship, path, or any other specific value) occurs across all the records in the stream. It can also refer to the total records in the stream. “Minding cardinality” is all about keeping this in mind; since operations execute per record in the stream, if the operation executes upon a variable with a value common amongst some or all of the built-up records (such as a MATCH out from a variable that refers to the same node in every row), then the operation is being performed redundantly, multiplicatively, on the same node for as many rows upon which it occurs. This can result in unnecessary db hits, and can further increase cardinality, which may slow down subsequent operations, as well as the entire query.

You may need to use aggregations or usage of DISTINCT at various places in your query to adjust cardinality to avoid performing redundant and costly operations.

Please ensure you PROFILE your queries, not just for timing and db hit information, but also to see if the db hits and number of rows streamed between the operations makes sense.

Avoid Cartesian Products

Variables in a MATCH with no relation to each other may generate cartesian products, which greatly increases cardinality.
MATCH (a:Person), (b:Company) RETURN a,b
The above returns the cartesian product, all possible combinations, of every person in the database with every company in the database.

MATCH (a:Person)-[:WORKS_FOR]->(b:Company) RETURN a,b
This one is better, since the records generated will be every person in the database with every company they worked for. A ***collect()*** on one variable or another would further reduce cardinality and provide more context to the query (all distinct people with the collection of companies they worked for, or all the distinct companies with the collection of people working for them).

MATCH (a:Person)-[:WORKS_FOR]->(b:Company)-[:HAS_PRESENCE_IN]->(c:Country) RETURN a,b,c
Even though each of these nodes has a relationship to each other, there is no direct relationship between **a** and **c**, so there will be cross product between each person at a company, and each country the company has a presence in. So far example, if there was only 1 company, with 5 persons working for it, with a presence in 3 countries, 15 records would result.

Aggregations

Aggregation functions (such as collect(), count(), sum(), avg(), min(), and max()) aggregate data across multiple records and usually reduce cardinality according to the implicit grouping key.

- All non-aggregation variables in the WITH or RETURN are implicitly used as the grouping key, and become the context for the aggregation variables. There is no explicit GROUP BY.
- If not familiar with this behavior, it may help to verbalize the WITH or RETURN variables to make sure you understand the records that will result:
- "For the given <non-aggregation var1> and <non-aggregation var2> and …, give me the <aggregation function> of <aggregation var>, ..."

Take the following query:
MATCH (movie:Movie)<-[:ACTED_IN]-(actor:Person)
WITH movie, actor, count(actor) as actorCount
...

The WITH clause might translate to:

"For the given movie and actor, give me the count of actor"

Upon reading this out, it may become clearer that there is a mistake in this query...since there is only one `actor` per record, the count of that single actor is always 1. If we removed `actor`, it will be removed from the implicit grouping key, meaning that we are now getting the count of all the actors for the given movie.

We are also reducing the cardinality. Before the WITH we had one record per movie/actor combination, so the same movie was occurring on multiple records (for each actor in that movie), and the same actor was occurring on multiple records (for each movie they acted in).

If we had wanted to perform a subsequent MATCH at this point, from `movie` to `director`, that match would have been performed multiple redundant times for the same movie present in multiple records.

After executing **WITH movie, collect(actor) as actors** we now have one distinct movie per record, and a match to find directors for the movie would only be performed once for each distinct movie node.

Notable Cypher Clauses

Cypher is not SQL. While some of the syntax seems the same or similar, it pays to learn and appreciate the differences.

The following covers some particular clauses having nuances that can trip up novices and journeymen alike.

MERGE

MERGE performs upsert operations on nodes, relationships, and patterns. It can be tricky to use if you don't understand how it works, which may lead to incorrect and unexpected results.

- If the entire pattern does not exist, the entire pattern will be created (excepting variables already bound to graph elements).
- A common symptom of not fully grasping the above may be the unexpected creation of duplicate graph elements.
- https://neo4j.com/developer/kb/understanding-how-merge-works/

WITH

Can be used to change which variables are in scope, and allows function calls to perform aggregations and calculations.

- Used often to aggregate results to pass into the next Cypher clause.
- WITH can be used multiple times in succession, which is often the only way to perform a series of subsequent operations that can't otherwise happen in a single clause, such as nested aggregations.
- Make sure to pass ALL identifiers that need to be accessed further along in the query!

WHERE

WHERE is not a clause in its own right — rather, it's part of MATCH, OPTIONAL MATCH, or WITH clause.

- In the case of WITH, WHERE simply filters the results.
- For MATCH and OPTIONAL MATCH on the other hand, WHERE adds constraints to the patterns described. It should not be seen as a filter after the matching is finished.
- OPTIONAL MATCH needs particular emphasis here. The behavior of OPTIONAL MATCH is to never filter out records where the OPTIONAL MATCH fails (either because of the lack of the pattern itself, or because of its associated WHERE clause). Instead, any newly-introduced variables from the OPTIONAL MATCH will be null. Existing variables will not be changed.
    - A WHERE after an OPTIONAL MATCH can be a common point of confusion for developers new to Cypher, as most SQL developers are used to WHERE always filtering out rows. If needing to filter out rows after an OPTIONAL MATCH, add a WITH between the OPTIONAL MATCH and the WHERE.
    - https://neo4j.com/developer/kb/why-where-clause-does-not-filter/

UNWIND

UNWIND takes a collection of elements and changes them into rows, binding them to a variable. The reverse of a COLLECT().

- For each record in the stream, generates a cross product of that record with each element of the collection.
- UNWIND only adds to the variables in scope. If the collection that was unwound was bound to a variable, that variable is still in scope.
- For a given record, if the collection being unwound is empty, UNWIND will filter out the record (so **UNWIND [] as foo** will wipe out all records in the stream)

    - To prevent wiping out the record, if needed, use **CASE** to use **[null]** in place of an empty collection.
- You can also UNWIND segments of a collection according to collection range syntax.
- https://neo4j.com/docs/developer-manual/current/cypher/clauses/unwind/

LIMIT

Limit limits the number of records in the entire result stream.

Note that LIMIT (as well as ORDER BY and SKIP) can be used after WITH clauses earlier in your query, not just after the final RETURN.

8. Procedures/Java API

Procedures make up one of the most powerful and flexible components of Neo4j. They are written in Java, and callable from Cypher.

Procedures use the following syntax:

CALL <procedureName>(<arguments>) [YIELD <variables yielded>]

Procedure names are typically formatted like java package names (such as ‘apoc.coll.intersection()’). They return a stream of some kind of result object (whose instance variables become the yielded variables) or void. If not a void return, you must YIELD at least one variable.

Class files with annotated procedure methods can be turned into a jar and placed into Neo4j’s plugins folder, where they become available for use on the next Neo4j restart.

Procedures use the Neo4j core API, and can often execute operations much faster than the Cypher equivalent. You have full access to Java, which allows flexibility of coding that isn’t possible with Cypher alone.

Procedures can be written to implement custom batching in multiple transactions, graph algorithms, custom traversals, imports, exports, connections to other databases and systems, and more.

APOC Procedures

APOC Procedures is the most popular procedure package for Neo4j at this time, and provides many powerful and flexible procedures and functions across a wide variety of use cases. We include the option of installing APOC Procedures into your db instances when using Neo4j Desktop, and recommend including APOC Procedures with your production deployments.

Keep in mind that APOC is closely tied with Neo4j implementation, so it is important to use the correct APOC version given your Neo4j version, and to upgrade your APOC version accordingly when you upgrade Neo4j (see the documentation for the version matrix).

GitHub - https://neo4j-contrib.github.io/neo4j-apoc-procedures/
Documentation - https://neo4j-contrib.github.io/neo4j-apoc-procedures/

Algo Procedures

Your own custom procedures

As previously mentioned, procedures are powerful and flexible, though they are more complex to implement, and the work required to write a custom procedure tends to be more technical, and of course requires Java proficiency.

9. Ingestion/Import/Load CSV

There are few options to ingest data into Neo4j:

1. Neo4j-admin import utility
2. LOAD CSV

neo4j-admin import:

This tool command line utility used to perform fast batch inserts into Neo4j database from CSV files. This can only be used on an empty database.

Syntax:

```
neo4j-admin import [--mode=csv] [--database=<name>]
                   [--additional-config=<config-file-path>]
                   [--report-file=<filename>]
                   [--nodes[:Label1:Label2]=<"file1,file2,...">]
                   [--relationships[:RELATIONSHIP_TYPE]=<"file1,file2,...">]
                   [--id-type=<STRING|INTEGER|ACTUAL>]
                   [--input-encoding=<character-set>]
                   [--ignore-extra-columns[=<true|false>]]
                   [--ignore-duplicate-nodes[=<true|false>]]
                   [--ignore-missing-nodes[=<true|false>]]
                   [--multiline-fields[=<true|false>]]
                   [--delimiter=<delimiter-character>]
                   [--array-delimiter=<array-delimiter-character>]
                   [--quote=<quotation-character>]
                   [--max-memory=<max-memory-that-importer-can-use>]
```

The documentation provided in manual best describes the format of the CSV files that nodes and relationship files should be.
Please refer to following documentation for further information:

https://neo4j.com/docs/operations-manual/current/tools/import/

LOAD CSV

LOAD CSV is very versatile cypher to load data from CSV files. LOAD CSV is great for loading small to medium datasets (approx 10 million rows in csv). The CSV files does not have to be in specific format. Even if it includes multiple columns, load csv can only load the necessary columns.

Features of LOAD CSV

- LOAD CSV can be used to just get samples, counts and distributions from the CSV files without loading data into CSV files. This can be very useful for validating data in the CSV files.
  For example:
  ```
  // assert correct line count
  LOAD CSV FROM "file-url" AS line
  RETURN count(*);

  // check first few raw lines
  LOAD CSV FROM "file-url" AS line WITH line
  RETURN line
  LIMIT 5;
  ```
- LOAD CSV supports resources compressed with gzip, Deflate, as well as ZIP archives.
- CSV files can be stored on the database server and are then accessible using a file:/// URL. Alternatively, LOAD CSV also supports accessing CSV files via HTTPS, HTTP, and FTP.

  For example:
  ```
  file:///data.csv

  https://host/path/data.csv
  ```
- All data from the CSV file is read as a string, you have to use toInt, toFloat, split or similar functions to convert
- Empty fields have to be skipped or replaced with default values during LOAD CSV

Best Practice: LOAD CSV

- Make sure to have indexes and constraints declared and ONLINE for entities you want to **MATCH** or **MERGE** on
- Always **MATCH** and **MERGE** on a single label and the indexed primary-key property
- Prefix your load statements with **USING PERIODIC COMMIT 10000**

- If possible, separate node creation from relationship creation into different statements.
- If your import is slow or runs into memory issues, see Mark's blog post on Eager loading
  http://www.markhneedham.com/blog/2014/10/23/neo4j-cypher-avoiding-the-eager/

- Make sure to set the heap size to a sufficient value. You do this by setting dbms.memory.heap.initial_size and dbms.memory.heap.max_size to at least 4G in **neo4j.conf**.
- Ensure that the page cache is sufficient. You do this by setting **dbms.memory.pagecache.size** in **neo4j.conf**. Ideally, the page cache should be large enough to keep the whole database in memory.
- Use **bin/cypher-shell** instead of the browser for better control. By default it connects to the database running on localhost but you can point it to a database anywhere over the network.

More documentation can be found:

http://neo4j.com/docs/developer-manual/current/cypher/clauses/load-csv/#query-load-csv

10. Monitoring

JMX Monitoring:

- Neo4j Enterprise Edition provides a set of JMX MBeans for understanding many details of the state of your instance or cluster. You can use Java tools such as JConsole or other licensed Application Monitoring solutions to view this data and proactively respond to issues.
- JMX
  - Neo4j-shell (Deprecated in 3.x versions of Neo4j)
    - bin/neo4j-shell –c "dbinfo –l"
    - bin/neo4j-shell –c "dbinfo –g Transactions"
  - Stored Procedures using Cypher-shell or Neo4j Browser
    - Query JMX management data by domain and name. For instance, "call dbms.queryJmx('org.neo4j:*')".
  - 3rd Party integration
    - Via JMX etc.
    - Usage of JVM monitoring tools like Jconsole, VisualJM, AppDynamics etc…

Enabling remote access

- Per default, the Neo4j Enterprise Server edition do not allow remote JMX connections, since the relevant options in the conf/neo4j.conf configuration file are commented out. To enable this feature, you have to uncomment by removing the # characters from the various dbms.jvm.additional=-Dcom.sun.management.jmxremote config parameters.

```
dbms.jvm.additional=-Dcom.sun.management.jmxremote.port=3637
dbms.jvm.additional=-Dcom.sun.management.jmxremote.authenticate=true
dbms.jvm.additional=-Dcom.sun.management.jmxremote.ssl=false
dbms.jvm.additional=-Dcom.sun.management.jmxremote.password.file=/etc/neo4j/jmx.password
dbms.jvm.additional=-Dcom.sun.management.jmxremote.access.file=/etc/neo4j/jmx.access
dbms.jvm.additional=-Djava.rmi.server.hostname=ec2-34-205-172-119.compute-1.amazonaws.com
```

- When commented in, the default values are setup to allow remote JMX connections with certain roles, refer to the conf/jmx.password, conf/jmx.access and conf/neo4j.conf files for details.
- Make sure that conf/jmx.password has the correct file permissions. The owner of the file has to be the user that will run the service, and the permissions should be read only for that user. On Unix systems, this is 0600.
- Also make sure to specify the absolute path for files jmx.password and jmx.access in the corresponding parameters *"dbms.jvm.additional=-Dcom.sun.management.jmxremote.password.file"* and *"dbms.jvm.additional=-Dcom.sun.management.jmxremote.access.file"*
    - On Windows, follow the tutorial at http://docs.oracle.com/javase/7/docs/technotes/guides/management/security-windows.html to set the correct permissions. If you are running the service under the Local System Account, the user that owns the file and has access to it should be SYSTEM.
- Make sure if connecting to AWS EC2 instance for Remote JMX to uncomment the hostname as below:

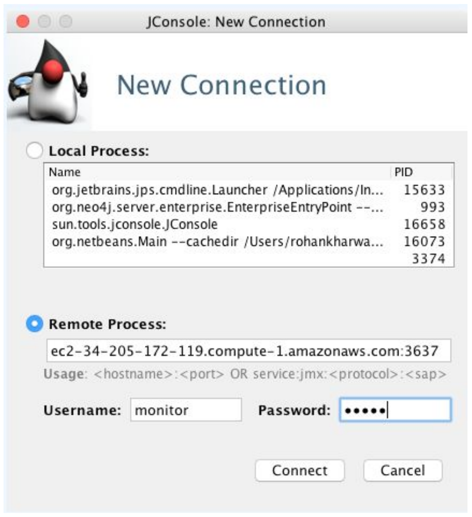
```
dbms.jvm.additional=-Djava.rmi.server.hostname=ec2-34-205-172-119.compute-1.amazonaws.com
```

- With this setup, you should be able to connect to JMX monitoring of the Neo4j server using :3637, with the username monitor and the password Neo4j.
- Note that it is possible that you have to update the permissions and/or ownership of the conf/jmx.password and conf/jmx.access files — refer to the relevant section in conf/neo4j.conf for details.

Connecting to Neo4j

- With the Neo4j database running, you'll connect your monitoring tool to the process running Neo4j.

- For local connection you want to look for a process such as "org.neo4j.server.enterprise.EnterpriseEntryPoint" to attach to.
- For remote connection, specify the public hostname to connect to JMX monitoring of the Neo4j server, the port :3637, along with the username “monitor” and the password “Neo4j”.



Local Process:

Name	PID
org.jetbrains.jps.cmdline.Launcher /Applications/In...	15633
org.neo4j.server.enterprise.EnterpriseEntryPoint --...	993
sun.tools.jconsole.JConsole	16658
org.netbeans.Main --cachedir /Users/rohankharwa...	16073
	3374

Remote Process: ec2-34-205-172-119.compute-1.amazonaws.com:3637

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

Username: monitor Password: •••••

What to Monitor

- We recommend referring to the documentation section describing the JMX MBeans to understand all metrics available.
- Query to see all JMX Metrics
- call dbms.queryJmx("org.neo4j:*")
- One example is to monitor the state of data replicated between two instances in a cluster. The difference between the LastCommittedTxId attribute on each instance will give you an idea of how much lag there is between your Master instance and any one of your Slave instances. Of course, if you are using a Enterprise Application Monitoring tool, you can define a threshold of lag as an alarm condition to notify the appropriate people.
- Use Stored procedures to extract the LastCommittedTxId (or other metrics).

  Example - call dbms.queryJmx("org.neo4j:instance=kernel#0,name=Transactions") yield attributesreturn attributes.LastCommittedTxId["value"]

Causal Cluster Monitoring with REST Endpoints:

There are multiple ways to query the Causal Cluster instance to find out the status of the instance.

1. Stored Procedure query
2. HTTP Endpoints

Stored Procedure

There are couple of Stored procedures that are useful. One such is **dbms.cluster.role()** that helps to determine the role of the instance if it is a Leader or Follower.
Another stored procedure that provides overview of the cluster configuration is **dbms.cluster.overview()**.

HTTP Endpoints

The following table provides CORE HTTP endpoint responses:

Endpoint	Instance State	Returned Code	Body text
`/db/manage/server/core/writable`	Leader	`200 OK`	`true`
	Follower	`404 Not Found`	`false`
`/db/manage/server/core/read-only`	Leader	`404 Not Found`	`false`
	Follower	`200 OK`	`true`
`/db/manage/server/core/available`	Leader	`200 OK`	`true`
	Follower	`200 OK`	`true`

Neo4j Metrics Monitoring:

Metrics reporting provides continuous analysis through the output of metrics as well as the inspection and management of currently-executing queries.

- Can report in two different ways:
  - Export Metrics to CSV
  - Send metrics to Graphite or any monitoring tool based on the Graphite protocol.
  - Monitoring Neo4j using Grafana (via Graphite)
    - https://s3.amazonaws.com/support.neo4j.com/Monitoring+Neo4j+using+Grafana.pdf
- Enabling metrics logging:

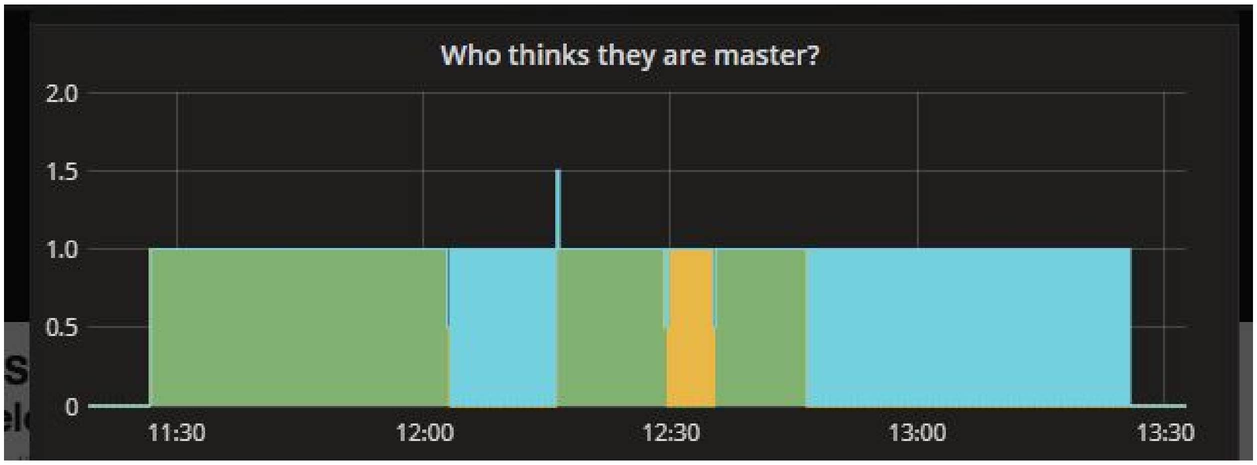
  https://neo4j.com/docs/operations-manual/3.2/monitoring/metrics/#metrics-enable

- Metrics can be used for monitoring
  - Transactions - number of transactions started, committed, active etc…
  - Page Cache metrics - reports page faults, evictions, hits etc…
  - Database entity metrics - number of nodes, rels, properties etc…
  - Core Metrics - Causal cluster metrics
  - Http threads
  - Legacy HA Monitoring
  - Complete list can be found here:
  - https://neo4j.com/docs/operations-manual/3.2/monitoring/metrics/reference/#metrics-reference

Best Practice: Monitoring Neo4j Using Grafana
https://s3.amazonaws.com/support.neo4j.com/Monitoring+Neo4j+using+Grafana.pdf   The above link will download a document that describes how to configure useful metrics to monitor in Grafana. The same idea can be applied to other monitoring tools. It also describes how to configure Alerts in Grafana for critical metrics.

Below are some ways of monitoring Causal Cluster, Legacy HA, Transactional and HTTP endpoint metrics.

Causal Cluster Monitoring:

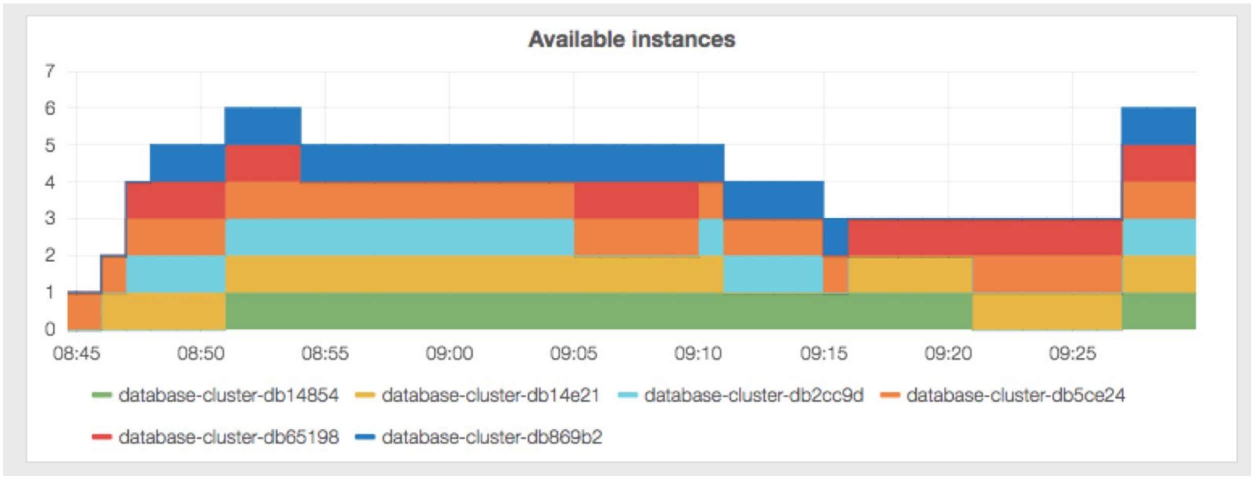


There are quite a few metrics that are enabled with Metrics monitoring to monitor the Causal Cluster. The complete list of those is provided in the documentation with link above. However the above picture shows when any of the 3 instances was a Leader. This is shown with help from config parameter **neo4j.causal_clustering.core.is_leader.** This helps us figure out if there were many re-elections and during what period.

Legacy HA Monitoring:

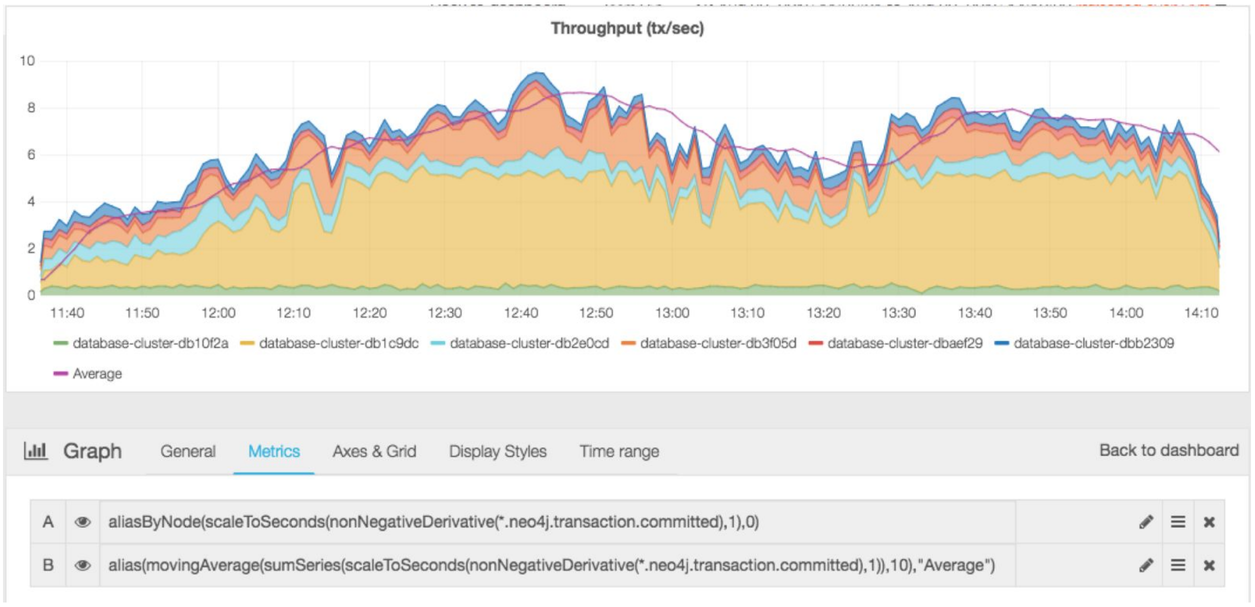
Two examples of valuable metrics for monitoring the cluster stability is **neo4j.cluster.is_available** and **neo4j.cluster.is_master**. Tracking the availability gives a good oversight of the general health of the instances in the cluster and combining both metrics also provides a representation of the roles in the cluster.

The following shows cluster availability while performing destructive operations (killing and restarting instances).



Transaction Monitoring:

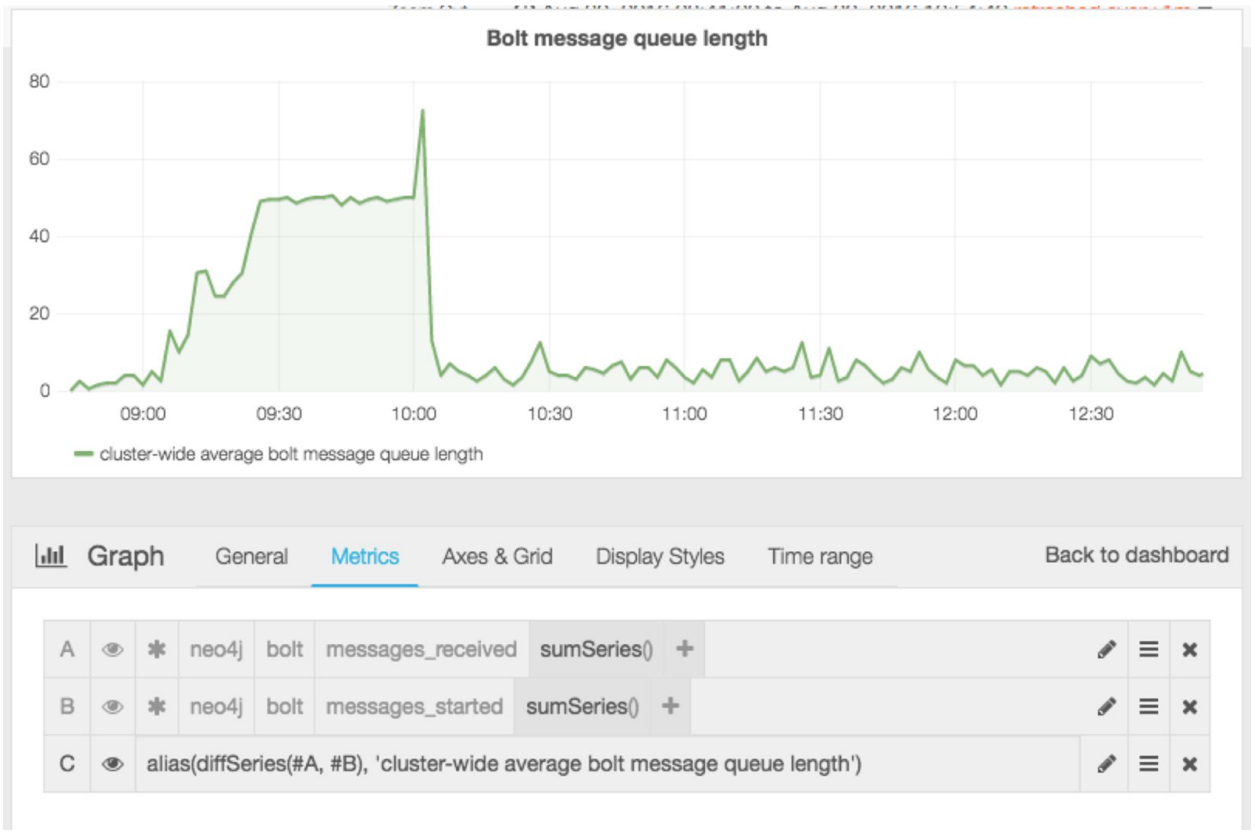
There are multiple metrics to help monitor transactions. A good place to start is **neo4j.transaction.committed** and **neo4j.transaction.active**, there are metrics to separate these in reads and writes as well. Committed transaction is an accumulating metric, using graphites API to plot the derivative is useful to get a more readable throughput plot. Combining these outputs with the cluster metrics discussed above a simple expected output would be that writes are being sent to the master only. Throughput can also be deceiving since transactions have different processing time. It is therefore good to combine with the active transactions metrics which will give a view of the current load on each instance.



```
aliasByNode(scaleToSeconds(nonNegativeDerivative(*.neo4j.transaction.committed),1),0)
alias(movingAverage(sumSeries(scaleToSeconds(nonNegativeDerivative(*.neo4j.transaction.committed),1)),10),"Average")
```

Above is the derivative of the committed transactions. The result is stacked to easily differentiate between instances. A moving average has also been added which is useful when throughputs are spiky.

The drivers also provide metrics that can help giving an even better overview of the transactions. These are also accumulated metrics but by using Graphites API to get the derivative and dividing metrics series with one another it is possible to customise these into much more useful outputs. Following are three examples of how to plot the average queue length, average queue time and average processing time. If the queue length is increasing it suggest that more transactions are being sent then what the system can handle.



```
A  neo4j  bolt  messages_received  sumSeries()
B  neo4j  bolt  messages_started  sumSeries()
C  alias(diffSeries(#A, #B), 'cluster-wide average bolt message queue length')
```

REST / HTTP Monitoring:

If still on the REST API there are the **neo4j.server.threads.jetty.all** and **neo4j.server.threads.jetty.idle** metrics to monitor transactional endpoints. If the idle threads reaches zero it suggest that the system is working at maximum capacity. At this point, you might consider throttling the number of requests from the client side, or increasing the number of dbms.threads.worker_count. The default is one thread per CPU core.

Neo4j also allows for monitoring jvm metrics such as memory and thread usage and gc information.

Monitoring system information of instances can be a great help. Running on ubuntu it is possible to install collectd which will send system metrics to graphite to plot alongside with your neo4j monitoring.

Setting up collectd

Grafana supports integration with collectd (https://collectd.org) to collect and monitor system performance. Other tools also likely have the ability to receive data from collectd. Here's how to set it up to work with Grafana.

Note: Collectd will need to be installed on each instance of Neo4j that you would like to monitor.

To install (on Ubuntu): `apt-get install collectd`

Configure to communicate with Grafana server:

In the file **/etc/collectd/collectd.conf.d/** create (or edit) the graphite.conf file with:

```
LoadPlugin write_graphite

<Plugin write_graphite>
  <Node "example">
    Host "<Your Grafana Server IP/Hostname"
    Port "2003"
    Protocol "tcp"
    LogSendErrors true
    StoreRates true
    AlwaysAppendDS false
    EscapeCharacter "_"
  </Node>
</Plugin>
~
```

Set a unique name for the server

In the **/etc/collectd/collectd.conf**

```
        HOSTNAME "<Unique Server Name>"
```

Restart collectd: `sudo service collectd restart`

11. Performance

System-wide

For System-wide performance concerns one should first start with memory configurations as detailed in the conf/neo4j.conf. In a perfect world the ideal is to have the entire contents of the graph in RAM memory (i.e. the size reported by linux command `du -hc graph.db/*store.db*` should be a starting value for dbms.memory.pagecache.size). If one is using HTTP REST to interact with the database verify the number of concurrent Cypher queries does not exceed conf/neo4j.conf parameter of dbms.threads.worker_count.

If all looks appropriate then you will want to utilize other tools to understand performance such

- Running Cypher `call dbms.listQueries();` so as to determine the number of queries in flight
- linux `top` to check for CPU and Memory Utilization
- if using Oracle Java you may want to consider running and capturing a Java Flight Recording (https://support.neo4j.com/hc/en-us/articles/217664337-How-do-I-enable-Java-Flight-Recorder-and-view-the-Results)

Single Query

- Determine if this is this query previously ran fast and now has degraded over time? Is the performance degradation as a result of an increase in data?
- Verify that the query performance is slow for the second running. By running it two times, the 2nd invocation may be faster as it will access the data in the dbms.memory.pagecache.size objects rather than the file system.
- Preface the query with either PROFILE or EXPLAIN so as to generate a query plan. Are indexes defined to help out the query in question. Does the query start with a NodeIndexScan or NodeIndexSeek? Running **CALL db.indexes()** will report the currently defined indexes. If an index is defined but not used adding a USING INDEX can force the optimizer to use the index.

- If query.log is available (i.e. conf/neo4j.conf defines **dbms.logs.query.enabled=true**) and if page cache hit/miss ratio is enabled (conf/neo4j.conf **dbms.logs.query.page_logging_enabled=true**) then upon completion of the query details will be written into the query.log similar to

```
2017-12-27 20:55:54.461+0000 INFO  0 ms: (planning: 0, cpu: 0, waiting: 0) - 3344 B - 0
page hits, 0 page faults - bolt-session  bolt        neo4j-java/dev
client/127.0.0.1:33038  server/127.0.0.1:7687>   -
match (n:Person) return count(n); - {} - {}
```

In the above *page hits* refers to the amount of data read from the **dbms.memory.pagecache.size** to satisfy the query, whereas *page faults* refers to the amount of data read from the filesystem to satisfy the query. Ideally we want to see page faults at 0.

- Improving certain types of Count operations.
  Neo4j allows for performance shortcuts by writing Cypher statements that reference node /relationship metadata. For example the metadata will record

  # of Nodes in the entire graph

  ```
  match (n) return count(n);
  ```

  will result in a NodeCountFromCountStore

  Including a WHERE clause
  # of Nodes per label

  ```
  match (n:Person) return count(n);
  ```

  will result in a NodeCountFromCountStore

  # of Relationships by Type from a Node for both incoming and outgoing direction

  ```
  match (n:Person)-[r:Acted_in]->() return count(*);
  ```

  will result in a RelationshipCountFromCountStore

  In the above examples, including a WHERE clause will not allow the query to use the metadata. Also for queries involving a path qualifying on both the source label and destination label will result in the metadata not being utilized.

  # of Relationships for a specific type from a specific node

  ```
  match (n:Person {name:'Tom Hanks'}) return size ( (n)-[:Acted_In]->() );
  ```

  will result in GetDegreePrimitive(0,Some(Acted_in),OUTGOING)

12. Logs

About Neo4j's Logs

Some logs are written by default, while others are not, and would need to be enabled via a configuration setting in neo4j.conf.

- **debug.log**
  - This is the main diagnostic log for the database. It contains information about startup, recovery, long garbage collections, cluster communication, checkpointing, along with exceptions and errors which may need investigation.
  - This log should be provided with any support ticket that is not a general question or cypher issue.
  - Enabled by default.
- **neo4j.log**
  - This is where standard output is logged. Startup text, some exceptions, and anything that shows up at the command line will be in here. It is also possible for some user defined procedures or functions to log here, or anything else that extends the main process.
  - If run as a service or from the console, this will not be written to neo4j.log, but rather to the journal (on Linux). If running linux and this file doesn't exist, refer to the following KB: https://support.neo4j.com/hc/en-us/articles/115014151428
  - Enabled by default.
- **query.log**
  - This log contains cypher statements that were completed on the database. It contains some general information about the query.
  - Controlled by: `dbms.logs.query.enabled=true`
  - By default it logs ALL QUERIES. Generally you should set this threshold to something like 1-2 seconds in production to prevent noise and excessive logging.
  - There are numerous additional debug options to log more information about each query. Note: Enabling these can have an impact on query performance, so do not keep these enabled in production.
    These are:
    - `dbms.logs.query.allocation_logging_enabled=true`
      - Logs total heap memory used during life of query (can be larger than max heap in some cases)
    - `dbms.logs.query.page_logging_enabled=true`
      - Logs page cache hits and faults to see if the query is slow due to hitting disk or not

                - dbms.logs.query.parameter_logging_enabled=true
                    - Log the query parameters so you can re-run with exact context
                - dbms.logs.query.time_logging_enabled=true
                    - Log the time breakdown of planning + CPU + waiting

- **security.log**
    - This log reports login attempts, and is a good place to look if you are having trouble setting up LDAP with Neo4j.
- **http.log**
    - This contains all HTTP requests, including HA/CC status endpoint requests.
    - It's only useful if using HTTP instead of Bolt to interact with the database.
    - If this log is heavily written to, it can impact heap memory usage in extreme cases.
    - Disabled by default.
- **gc.log**
    - This logs all garbage collection information. It's generally only needed when troubleshooting long garbage collection issues.
    - Disabled by default.
- **bolt.log (3.3+)**
- **Log Rotations**

    Log rotation is enabled by default for the system generated log files noted below:

    - HTTP logs
    - GC logs
    - Debug Logs
    - Query Logs
    - Security Logs
    - Raft Logs

    The log rotation capability is further configurable with respect to retention period, size, number of files, or minimum delay since last rotation occurred by way of a number of parameters in the configuration files, as noted below:

    - **dbms.logs.xxx.rotation.keep_number**
        - Default should be fine, but increase this if you want to keep more files
    - **dbms.logs.xxx.rotation.size**
        - Default is usually fine, but you may want this to be larger or smaller depending on your needs
    - **Note: neo4j.log does NOT rotate.** This can be achieved by using the method described in the following KB article: https://support.neo4j.com/hc/en-us/articles/226251327

13. Troubleshooting

Troubleshooting Diagnostics

Troubleshooting efforts must include as many of the following information as possible as collectively they will provide most everything that is needed in order to work on an issue:

- neo4j.conf
- debug.log
- neo4j.log
- query.log (if exists)
- security.log (if exists)
- http.log (if exists)
- gc.log (if exists)
- bolt.log (3.3+)
- csv metrics (if enabled)
- dbms.listQueries() output
- dbms.cluster.overview() output
- Current graph.db content and size
- Thread dump of running neo4j process
- List of contents in import and plugin directories
- ‘top’ output
- Optionally get Transaction logs
- Optionally get Raft logs

Troubleshooting Query Performance Issues

1) Establish overall health of the node by running
   a) :sysinfo
   b) call dbms.cluster.overview()   (on Causal clusters)
   c) call apoc.meta.stats()

2) Identify which queries are slow, when and relative to what benchmark/ baseline/timeframe.
   a) Call dbms.listQueries() output
   b) examine query.log (Recommended to be enabled)

3) Collect debug.log, neo4j.log, and neo4j.conf files as well as query.log

4) Is JVM heap memory settings configured and sized properly properly?

   a) Ideally, **dbms.memory.heap.initial_size**, and **dbms.memory.heap.max_size** should be set to the same value and generally less between 8-16 GB for production like implementations.   It should be sized according to the underlying workload which can be a function of concurrency, query complexity, etc. If the heap memory is not sized properly, the following issues could occur:

      i) The application could experience frequent and/or long Garbage Collection(GC) related pauses.
      ii) Or the application cold simply run out of memory, resulting in the client application to abort.
      iii) GC related issues typically result similar messages in **debug.log**:
         (1) WARN [o.n.k.i.c.MonitorGc] GC Monitor: Application threads blocked for 46789ms
         (2) OutOfMemoryError: GC overhead limit exceeded
         (3) *java.lang.OutOfMemoryError: Java heap space*

   b) Use the following command to find the longest GC Pauses in debug or messages log:

```
grep -n -i blocked debug.log | sort -r -n -k 11 | head -10
```

   Note: A long GC would be something close to the cluster heartbeat timeout, or of significant length and frequency as to cause persistent performance issues. Generally, for a Causal Cluster, anything over 7 seconds on the leader will cause an election, and on a HA cluster it would need to be 40 seconds or longer. A single GC of 5 seconds, though, should not be cause for alarm.

c) As a matter of best practice, JVM overall utilization should be always profiled during the testing phase to ensure that it appropriately correlates to the overall characteristics of the application workload with respect to concurrency, throughput, as well as query complexities.

d) JVM as well as overall memory utilization can be captured and analyzed by:
   (1) Enabling Metrics to generate either CSV formatted files and/or feed Graphite/Grafana for real-time charting/monitoring of its usage.
   (2) Enable GC Logging
   (3) Alternatively, JFR can also be used as such

5) Is the page cache set properly? As with any database, the more the actual data is fit in the memory, the better the performance, as it minimizes pagecache misses. The amount of memory allocated to pagecache is primarily determined by **dbms.memory.pagecache.size** in neo4j.conf file. Thus size it accordingly with respect to the overall database size (du -hc $NEO4j_HOME/data/databases/graph.db/store.db). This can be monitored by examining the query attributes obtained from:
   a) Call **dbms.listQueries()** output
   b) query.log (needs to be enabled - see below)

6) Enable query.log in order to capture a log of executed queries that takes longer than a specified threshold. Query.log is enabled by setting the following parameters:

   a) **dbms.logs.query.enabled** = TRUE
   b) **dbms.logs.query.threshold**, which is the time interval after which a query is logged (defaults to 0s so all queries will be logged, so you may want to adjust this);
   c) **dbms.logs.query.time_logging_enabled**, which if set to true adds detailed time breakdowns for how a query's execution time was spent;**dbms.logs.query.page_logging_enabled**, which will provide information about pagecache hits and misses with regard to the page cache.

   d) **dbms.transaction.timeout** can be set to a system-wide timeout after which the query will be killed. Note that if a query is waiting on a lock to become available, it will not be killed for exceeding the transaction timeout until it acquires the lock it's waiting on. This can be addressed by setting a timeout value for dbms.lock.acquisition.timeout, which should kill a query waiting too long to acquire a lock.

7) Identify the most expensive queries and
   a) Assess if there is a common denominator involving (cypher syntax, indexes, etc)
   b) Are there any blocking GC's in debug.log
   c) Index definitions
   d) Obtain the explain plans and analyze - Look for opportunities to minimize dbhits.
      i) Prefix cypher query with PROFILE or EXPLAIN

8) Once your particular query problem has been resolved, repeat the above process for additional queries which fall within the scope of a smaller threshold

9) In rare cases, if it seems that the query is going nowhere, obtain thread-dump as follows:

   a) Obtain neo4j process id:

      `ps -eaf | grep neo4j`

   b) Run jcmd

      `jcmd <pid> > Thread.txt`

      i) If **jcmd** is not available, use **jstack**

   c) `Docker` -

      i) The thread dump output is stored in the docker logs rather than the terminal.

      ii) Obtain a thread dump using **Jconsole** or using `kill -3 <pid>` (specifying the pid of the java process - This outputs the thread dump into the console log)

HA Cluster Troubleshooting

This topic has been covered quite comprehensively in the following KB:
https://support.neo4j.com/hc/en-us/articles/226339848-Troubleshooting-Cluster-Issues

Useful Tools and Commands Summary

Thread Dump

1. Obtain neo4j process id:

   ```
   ps -eaf | grep neo4j
   ```

2. Run jcmd:

   ```
   jcmd <pid> > Thread.txt
   ```

   If **jcmd** is not available, use `jstack -f <neo4j_pid>`

3. `Docker` -
   a. The thread dump output is stored in the docker logs rather than the terminal.
   b. Obtain a thread dump using **Jconsole** or using `kill -3 <pid>` (specifying the pid of the java process - This outputs the thread dump into the console log)

Heap Dump

Obtain a heap dump for a specific process by running the following command below which by default will generate a file called "heap.bin" in the current directory.

```
$JAVA_HOME/bin/jmap -dump:format=b,file=/<path>/heap.dump <processID>
```

The obtained heap dump file can be further analyzed using jhat command:

```
jhat -J-mx768m -stack false "/<path>/heap.dump"
```

Enabling JFR (Oracle JDK Only)

Set the following parameters in neo4j.cfg and restart the database:

- `dbms.jvm.additional=-XX:+UnlockCommercialFeatures`
- `dbms.jvm.additional=-XX:+FlightRecorder`

To begin a time based recording, at the command line run:

`$JAVA_HOME/jcmd <pid> JFR.start duration=3600s filename=myrecording.jfr settings=/usr/lib/jvm/java-8-oracle/jre/lib/jfr/profile.jfc `

View the following KB for more details:
KB: https://support.neo4j.com/hc/en-us/articles/217664337

Enable GC Logging

Enable GC Logging by uncommenting the following entries in neo4j.log and restart the database:

- dbms.logs.gc.enabled=true
- dbms.logs.gc.options=-XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCApplicationStoppedTime -XX:+PrintPromotionFailure
- dbms.logs.gc.rotation.keep_number=5
- dbms.logs.gc.rotation.size=20m

See the following article for more details:
http://docs.oracle.com/cd/E19957-01/819-0084-10/pt_tuningjava.html#wp57013 for more information.

Validating Network Connectivity

- KB: https://support.neo4j.com/hc/en-us/articles/115009593907-A-light-weight-approach-to-validating-network-port-connectivity

Miscellaneous helpful commands

- Find a class within a directory of jars:

  ```
  for i in *.jar; do jar -tvf "$i" | grep -Hsi MyClass && echo "$i"; done
  ```

- Top 50 Slowest queries from Query log:

```
grep -i "SUCCESS" query.log | sort -r -n -k +6 | head -50 >
long_queries.log
```

- Find Longest GC Pauses in debug or messages log:

```
grep -n -i blocked debug.log | sort -r -n -k 11 | head -10
```

- Strip all comments / empty lines of neo4j.conf file:

```
grep -v "^#" neo4j.conf | sed -e '/^$/d' | sort
```

- **Top 5 memory consuming commands**

```
ps -eo pmem,pcpu,vsize,pid | sort -k 1 -nr | head -5
```