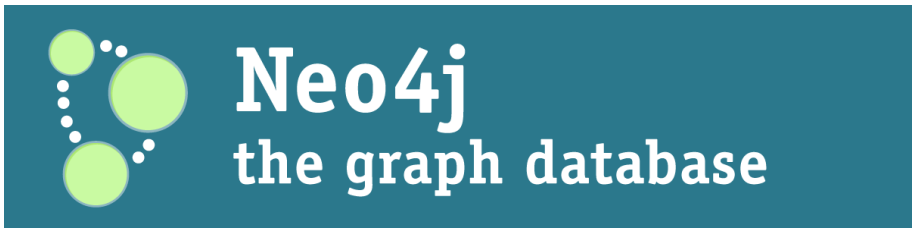

Neo4j
the graph database

The Neo4j Manual v1.3

Tobias Ivarsson
Andreas Kollegger
Peter Neubauer
Johan Svensson
Andrés Taylor
Jim Webber
Edited by Anders Nawroth

The Neo4j Manual v1.3

by Tobias Ivarsson, Andreas Kollegger, Peter Neubauer, Johan Svensson, Andrés Taylor, Jim Webber, and Anders Nawroth

License

Note

Table of Contents

List of Figures

List of Tables

Introduction

This is a reference manual. The material is practical, technical, and focused on answering specific questions. It addresses how things work, what to do and what to avoid to successfully run Neo4j in a production environment. After a brief introduction, each topic area assumes general familiarity as it addresses the particular details of Neo4j.

The goal is to be thumb-through and rule-of-thumb friendly.

Each section should stand on its own, so you can hop right to whatever interests you. When possible, the sections distill "rules of thumb" which you can keep in mind whenever you wander out of the house without this manual in your back pocket.

1. Who should read this

The topics should be relevant to architects, administrators, developers and operations personnel. You should already know about Neo4j and using graphs to store data. If you are completely new to Neo4j please check out http://neo4j.org first.

2. Neo4j highlights

As a robust, scalable and high-performance database, Neo4j is suitable for lightweight projects or full enterprise deployment.

It features:

- true ACID transactions
- high availability
- scales to billions of nodes and relationships
- high speed querying through traversals

Proper ACID behavior is the foundation of data reliability. Neo4j enforces that all mutating operations occur within a transaction, guaranteeing consistent data. This robustness extends from single instance embedded graphs to multi-server high availability installations. For details, see Chapter 3, *Transaction management*.

Reliable graph storage can easily be added to any application. A property graph can scale in size and complexity as the application evolves, with little impact on performance. Whether starting new development, or augmenting existing functionality, Neo4j is only limited by physical hardware.

A single server instance can handle a graph of billions of nodes and relationships. When data throughput is insufficient, the graph database can be distributed among multiple servers in a high availability configuration. See Chapter 7, *High Availability* to learn more.

The graph database storage shines when storing richly-connected data. Querying is performed through traversals, which can perform millions of "joins" per second.

Part I. Reference Documentation

Chapter 1. Installation & Deployment

1.1. Deployment Scenarios

Neo4j can be embedded into your application, run as a standalone server or deployed on several machines to provide high availability.

Table 1.1. Neo4j deployment options

	Single Instance	Multiple Instances
Embedded	EmbeddedGraphDatabase	HighlyAvailableGraphDatabase
Standalone	Neo4j Server	Neo4j Server high availability mode

1.1.1. Server

Neo4j is normally accessed as a standalone server, either directly through a REST interface or through a language-specific driver. More information about Neo4j server is found in Chapter 4, *Neo4j Server*. For running the server in high availability mode, see Section 4.4, “Starting the Neo4j server in high availability mode”.

1.1.2. Embedded

Neo4j can be embedded directly in a server application by including the appropriate Java libraries. When programming, you can refer to the `GraphDatabaseService` API. To switch from a single instance to multiple highly available instances, simply switch from the concrete `EmbeddedGraphDatabase` to the `HighlyAvailableGraphDatabase`.

1.2. System Requirements

Memory constrains graph size, disk I/O constrains read/write performance, as always.

1.2.1. CPU

Performance is generally memory or I/O bound for large graphs, and compute bound for graphs which fit in memory.

Minimum
: Intel 486

Recommended
: Intel Core i7

1.2.2. Memory

More memory allows even larger graphs, but runs the risk of inducing larger Garbage Collection operations.

Minimum
: 1GB

Recommended
: 4-8GB

1.2.3. Disk

Aside from capacity, the performance characteristics of the disk are the most important when selecting storage.

Minimum
: SCSI, EIDE

Recommended
: SSD w/ SATA

1.2.4. Filesystem

For proper ACID behavior, the filesystem must support flush (fsync, fdatasync).

Minimum
: ext3 (or similar)

Recommended
: ext4, ZFS

1.2.5. Software

Neo4j is Java-based.

Java
: 1.6+

Operating Systems
: Linux, Windows XP, Mac OS X

1.3. Installation

Neo4j can be installed as a server, running either as a headless application or system service. For Java developers, it is also possible to use Neo4j as a library, embedded in your application.

For information on installing Neo4j as a server, see Section 4.1, “Server Installation”.

1.3.1. Embedded Installation

The latest release is always available from from http://neo4j.org/download, packaged as part of the Neo4j server. After selecting the appropriate version for your platform, embed Neo4j in your Java

application, by including the Neo4j library jars in your build. Either take the jar files from the `lib` directory of the download, or directly use the artifacts available from Maven Central Repository [1].

Maven dependency.

```
<project>
...
 <dependencies>
    <dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j</artifactId>
    <version>{neo4j-version}</version>
    <packaging>pom</packaging>
    </dependency>
    ...
 </dependencies>
...
</project>
```

Where {neo4j-version} is for example 1.3.

Note

Stable and milestone releases are available at Maven Central Repository.

1.4. Upgrading

Normally a properly shutdown Neo4j database can be upgraded directly to a new minor version. A database can be upgraded from a minor version to the next, e.g. 1.1 –> 1.2, and 1.2 –> 1.3, but you can not jump directly from 1.1 –> 1.3. The upgrade process is a one way step; databases cannot be downgraded.

However, some upgrades make significant changes to the database store. Neo4j will refuse to start when a significant upgrade is required, requiring explicit upgrade configuration.

1.4.1. Normal Upgrade

To perform a normal upgrade (for minor changes to the database store):

1. download the newer version of Neo4j
2. cleanly shutdown the database to upgrade, if it is running
3. startup the database with the newer version of Neo4j

1.4.2. Special Upgrade

To perform a special upgrade (for significant changes to the database store):

1. make sure the database you are upgrading has been cleanly shut down
2. set the Neo4j configuration parameter "allow_store_upgrade=true"

[1] http://repo1.maven.org/maven2/org/neo4j/

3. start the database
4. the upgrade will happen during startup and the process is done when the database has been successfully started
5. "allow_store_upgrade=true" configuration parameter should be removed, set to "false" or commented out

1.4.3. Upgrade 1.3.M03 –> 1.3.M04

Warning

Upgrading from 1.3.M03 –> 1.3.M04 must be done explicitly since store format has changed between those two versions.

The store format, as well as logical log format, have changed between these two versions to allow for bigger stores.

1.4.4. Upgrade 1.2 –> 1.3

Warning

Upgrading from 1.2 –> 1.3 must be done explicitly since store format has changed between those two versions.

The store format, as well as logical log format, have changed between these two versions to allow for bigger stores.

Important

Although id ranges has been increased the space used to store the database will not increase compared to the previous version.

Upgrading between these two version needs to be performed explicitly using a configuration parameter at startup (see "Special Upgrade").

Caution

Upgrade cannot be performed if either the number of relationship types or the configured block size for either the dynamic array store or string store is greater than 65534.

Caution

Indexes created using the old IndexService/LuceneIndexService are no longer accessible out of the box in 1.3 in favor of the integrated index. An automatic upgrade isn't possible so a full rebuild of the index data into the integrated index framework is required.
For reference the legacy index can be downloaded from the Neo4j repository, http://m2.neo4j.org/org/neo4j/neo4j-legacy-index/

1.4.5. Upgrade 1.1 –> 1.2

Upgrading from Neo4j 1.1 to Neo4j 1.2 is a "normal" upgrade.

1.5. Usage Data Collector

The Neo4j Usage Data Collector is a sub-system that gathers usage data, reporting it to the UDC-server at udc.neo4j.org. It is easy to disable, and does not collect any data that is confidential. For more information about what is being sent, see below.

The Neo4j team uses this information as a form of automatic, effortless feedback from the Neo4j community. We want to verify that we are doing the right thing by matching download statistics with usage statistics. After each release, we can see if there is a larger retention span of the server software.

The data collected is clearly stated here. If any future versions of this system collect additional data, we will clearly announce those changes.

The Neo4j team is very concerned about your privacy. We do not disclose any personally identifiable information.

1.5.1. Technical Information

To gather good statistics about Neo4j usage, UDC collects this information:

- Kernel version - the build number, and if there are any modifications to the kernel.
- Store id - it is 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 - this is either "neo4j" or "maven". If you downloaded Neo4j from the Neo4j website, it’s "neo4j", if you are using Maven to get Neo4j, it will be "maven".
- Java version - the referrer string shows which version of Java is being used.

After startup, UDC waits for ten minutes before sending the first ping. It does this for two reasons; first, we don’t want the startup to be slower because of UDC, and secondly, we want to keep pings from automatic tests to a minimum. The ping to the UDC servers is done with a HTTP GET.

1.5.2. How to disable UDC

We’ve tried to make it extremely easy to disable UDC. In fact, the code for UDC is not even included in the kernel jar but as a completely separate component.

There are three ways you can disable UDC:

1. The easiest way is to just remove the neo4j-udc-*.jar file. By doing this, the kernel will not load UDC, and no pings will be sent.
2. If you are using Maven, and want to make sure that UDC is never installed in your system, a dependency element like this will do that:

```
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j</artifactId>
```

```
  <version>{neo4j-version}</version>
  <type>pom</type>
  <exclusions>
    <exclusion>
      <groupId>org.neo4j</groupId>
      <artifactId>neo4j-udc</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Where {neo4j-version} is for example 1.3.

3. Lastly, if you are using a packaged version of Neo4j, and do not want to make any change to the jars, a system property setting like this will also make sure that UDC is never activated: `-Dneo4j.ext.udc.disable=true`.

Chapter 2. Configuration & Performance

In order to get optimum performance out of Neo4j for your application there are a few parameters that can be tweaked. The two main components that can be configured are the Neo4j caches and the JVM that Neo4j runs in. The following sections describe how to tune these.

2.1. Caches in Neo4j

Neo4j utilizes two different types of caches: A file buffer cache and an object cache. The file buffer cache caches the storage file data in the same format as it is stored on the durable storage media. The object cache caches the nodes, relationships and properties in a format that is optimized for high traversal speeds and transactional mutation.

2.1.1. File buffer cache

Quick info

- The file buffer cache is sometimes called *low level cache* or *file system cache*.
- It caches the Neo4j data as stored on the durable media.
- It uses the operating system memory mapping features when possible.
- Neo4j will configure the cache automatically as long as the heap size of the JVM is configured properly.

The file buffer cache caches the Neo4j data in the same format as it is represented on the durable storage media. The purpose of this cache layer is to improve both read and write performance. The file buffer cache improves write performance by writing to the cache and deferring durable write until the logical log is rotated. This behavior is safe since all transactions are always durably written to the logical log, which can be used to recover the store files in the event of a crash.

Since the operation of the cache is tightly related to the data it stores, a short description of the Neo4j durable representation format is necessary background. Neo4j stores data in multiple files and relies on the underlying file system to handle this efficiently. Each Neo4j storage file contains uniform fixed size records of a particular type:

Store file	Record size	Contents
nodestore	9 B	Nodes
relstore	33 B	Relationships
propstore	25 B	Properties for nodes and relationships
stringstore	133 B	Values of string properties
arraystore	133 B	Values of array properties

For strings and arrays, where data can be of variable length, data is stored in one or more 120B chunks, with 13B record overhead. The sizes of these blocks can actually be configured when the store is created using the `string_block_size` and `array_block_size` parameters. The size of each record type can also be used to calculate the storage requirements of a Neo4j graph or the appropriate cache size for each file buffer cache. Note that some strings can be stored without using the string store, see Section 2.3, “Compressed storage of short strings”.

Neo4j uses multiple file buffer caches, one for each different storage file. Each file buffer cache divides its storage file into a number of equally sized windows. Each cache window contains an even number of storage records. The cache holds the most active cache windows in memory and tracks hit vs. miss ratio for the windows. When the hit ratio of an uncached window gets higher than the miss ratio of a cached window, the cached window gets evicted and the previously uncached window is cached instead.

Configuration

Parameter	Possible values	Effect
`use_memory_mapped_buffers`	`true` or `false`	If set to `true` Neo4j will use the operating systems memory mapping functionality for the file buffer cache windows. If set to `false` Neo4j will use its own buffer implementation. In this case the buffers will reside in the JVM heap which needs to be increased accordingly. The default value for this parameter is `true`, except on Windows.
`neostore.nodestore.db.mapped_memory`	The maximum amount of memory to use for memory mapped buffers for this file buffer cache. The default unit is `MiB`, for other units use any of the following suffixes: `B`, `k`, `M` or `G`.	The maximum amount of memory to use for the file buffer cache of the node storage file.
`neostore.relationshipstore.db.mapped_memory`		The maximum amount of memory to use for the file buffer cache of the relationship store file.
`neostore.propertystore.db.index.keys.mapped_memory`		The maximum amount of memory to use for the file buffer cache of the something-something file.
`neostore.propertystore.db.index.mapped_memory`		The maximum amount of memory to use for the file buffer cache of the something-something file.
`neostore.propertystore.db.mapped_memory`		The maximum amount of memory to use for the file buffer cache of the property storage file.
`neostore.propertystore.db.strings.mapped_memory`		The maximum amount of memory to use for the file buffer cache of the string property storage file.

Parameter	Possible values	Effect
`neostore.propertystore.db.arrays.mapped_memory`		The maximum amount of memory to use for the file buffer cache of the array property storage file.
`string_block_size`	The number of bytes per block.	Specifies the block size for storing strings. This parameter is only honored when the store is created, otherwise it is ignored. Note that each character in a string occupies two bytes, meaning that a block size of 120 (the default size) will hold a 60 character long string before overflowing into a second block. Also note that each block carries an overhead of 13 bytes. This means that if the block size is 120, the size of the stored records will be 133 bytes.
`array_block_size`		Specifies the block size for storing arrays. This parameter is only honored when the store is created, otherwise it is ignored. The default block size is 120 bytes, and the overhead of each block is the same as for string blocks, i.e., 13 bytes.
`dump_configuration`	`true` or `false`	If set to `true` the current configuration settings will be written to the default system output, mostly the console or the logfiles.

When memory mapped buffers are used (`use_memory_mapped_buffers = true`) the heap size of the JVM must be smaller than the total available memory of the computer, minus the total amount of memory used for the buffers. When heap buffers are used (`use_memory_mapped_buffers = false`) the heap size of the JVM must be large enough to contain all the buffers, plus the runtime heap memory requirements of the application and the object cache.

When reading the configuration parameters on startup Neo4j will automatically configure the parameters that are not specified. The cache sizes will be configured based on the available memory on the computer, how much is used by the JVM heap, and how large the storage files are.

2.1.2. Object cache

Quick info

- The object cache is sometimes called *high level cache*.
- It caches the Neo4j data in a form optimized for fast traversal.

The object cache caches individual nodes and relationships and their properties in a form that is optimized for fast traversal of the graph. The content of this cache are objects with a representation geared towards supporting the Neo4j object API and graph traversals. Reading from this cache is 5 to 10 times faster than reading from the file buffer cache. This cache is contained in the heap of the JVM and the size is adapted to the current amount of available heap memory.

Nodes and relationships are added to the object cache as soon as they are accessed. The cached objects are however populated lazily. The properties for a node or relationship are not loaded until properties are accessed for that node or relationship. String (and array) properties are not loaded until that particular property is accessed. The relationships for a particular node is also not loaded until the relationships are accessed for that node. Eviction from the cache happens in an LRU manner when the memory is needed.

Configuration

The main configuration parameter for the object cache is the `cache_type` parameter. This specifies which cache implementation to use for the object cache. The available cache types are:

`cache_type`	Description
`none`	Do not use a high level cache. No objects will be cached.
`soft`	Provides optimal utilization of the available memory. Suitable for high performance traversal. May run into GC issues under high load if the frequently accessed parts of the graph does not fit in the cache. This is the default cache implementation.
`weak`	Provides short life span for cached objects. Suitable for high throughput applications where a larger portion of the graph than what can fit into memory is frequently accessed.
`strong`	This cache will cache **all data** in the **entire graph**. It will never release memory held by the cache. Provides optimal performance if your graph is small enough to fit in memory.

Heap memory usage

This table can be used to calculate how much memory the data in the object cache will occupy on a 64bit JVM:

Object	Size	Comment
Node	`344 B`	*Size for each node (not counting its relationships or properties).*

Object	Size	Comment
	48 B	*Object overhead.*
	136 B	*Property storage (ArrayMap 48B, HashMap 88B).*
	136 B	*Relationship storage (ArrayMap 48B, HashMap 88B).*
	24 B	*Location of first / next set of relationships.*
Relationship	208 B	*Size for each relationship (not counting its properties).*
	48 B	*Object overhead.*
	136 B	*Property storage (ArrayMap 48B, HashMap 88B).*
Property	116 B	*Size for each property of a node or relationship.*
	32 B	*Data element - allows for transactional modification and keeps track of on disk location.*
	48 B	*Entry in the hash table where it is stored.*
	12 B	*Space used in hash table, accounts for normal fill ratio.*
	24 B	*Property key index.*
Relationships	108 B	*Size for each relationship type for a node that has a relationship of that type.*
	48 B	*Collection of the relationships of this type.*
	48 B	*Entry in the hash table where it is stored.*
	12 B	*Space used in hash table, accounts for normal fill ratio.*
Relationships	8 B	*Space used by each relationship related to a particular node (both incoming and outgoing).*
Primitive	24 B	*Size of a primitive property value.*
String	64+B	*Size of a string property value. `64 + 2*len(string) B` (64 bytes, plus two bytes for each character in the string).*

2.2. JVM Settings

Properly configuring memory utilization of the JVM is crucial for optimal performance. As an example, a poorly configured JVM could spend all CPU time performing garbage collection (blocking all threads from performing any work). Requirements such as latency, total throughput and available hardware have to be considered to find the right setup. In production, Neo4j should run on a multi core/CPU platform with the JVM in server mode.

2.2.1. Configuring heap size and GC

A large heap allows for larger node and relationship caches — which is a good thing — but large heaps can also lead to latency problems caused by full garbage collection. The different high level cache implementations available in Neo4j together with a suitable JVM configuration of heap size and garbage collection (GC) should be able to handle most workloads.

The default cache (soft reference based LRU cache) works best with a heap that never gets full: a graph where the most used nodes and relationships can be cached. If the heap gets too full there is a

risk that a full GC will be triggered; the larger the heap, the longer it can take to determine what soft references should be cleared.

Using the strong reference cache means that *all* the nodes and relationships being used must fit in the available heap. Otherwise there is a risk of getting out-of-memory exceptions. The soft reference and strong reference caches are well suited for applications were the overal throughput is important.

The weak reference cache basically needs enough heap to handle the peak load of the application — peak load multiplied by the average memory required per request. It is well suited for low latency requirements were GC interuptions are not acceptable.

When running Neo4j on Windows, keep in mind that the memory mapped buffers are allocated on heap by default, so need to be taken into consideration when determining heap size.

Table 2.1. Guidelines for heap size

Number of primitives	RAM size	Heap configuration	Reserved RAM for the OS
10M	2GB	512MB	the rest
100M	8GB+	1-4GB	1-2GB
1B+	16GB-32GB+	4GB+	1-2GB

The recommended garbage collector to use when running Neo4j in production is the Concurrent Mark and Sweep Compactor turned on by supplying -XX:+UseConcMarkSweepGC as a JVM parameter.

2.3. Compressed storage of short strings

Neo4j will classify your strings and store them accordingly. If a string is classified as a short string it will be stored without indirection in the property store. This means that there will be no string records created for storing that string. Additionally, when no string record is needed to store the property, it can be read and written in a single lookup. This leads to improvements in performance and lower storage overhead.

For a string to be classified as a short string, one of the following must hold:

- It is encodable in UTF-8 or Latin-1, 7 bytes or less.
- It is alphanumerical, and 10 characters or less (9 if using accented european characters).
- It consists of only upper case, or only lower case characters, including the punctuation characters space, underscore, period, dash, colon, or slash. Then it is allowed to be up to 12 characters.
- It consists of only numerical characters, inlcuding the punctuation characters plus, comma, single quote, space, period, or dash. Then it is allowed to be up to 15 characters.

Chapter 3. Transaction management

In order to fully maintain data integrity and ensure good transactional behavior, Neo4j supports the ACID properties:

- atomicity - if any part of a transaction fails, the database state is left unchanged
- consistency - any transaction will leave the database in a consistent state
- isolation - during a transaction, modified data cannot be accessed by other operations
- durability - the DBMS can always recover the results of a committed transaction

Specifically:

- All modifications to Neo4j data must be wrapped in transactions.
- The default isolation level is `READ_COMMITTED`.
- Data retrieved by traversals is not protected from modification by other transactions.
- Non-repeatable reads may occur (i.e., only write locks are acquired and held until the end of the transaction).
- One can manually acquire write locks on nodes and relationships to achieve higher level of isolation (`SERIALIZABLE`).
- Locks are acquired at the Node and Relationship level.
- Deadlock detection is built into the core transaction management.

3.1. Interaction cycle

All write operations that work with the graph must be performed in a transaction. Transactions are thread confined and can be nested as “flat nested transactions”. Flat nested transactions means that all nested transactions are added to the scope of the top level transaction. A nested transaction can mark the top level transaction for rollback, meaning the entire transaction will be rolled back. To only rollback changes made in a nested transaction is not possible.

When working with transactions the interaction cycle looks like this:

1. Begin a transaction.
2. Operate on the graph performing write operations.
3. Mark the transaction as successful or not.
4. Finish the transaction.

It is very important to finish each transaction. The transaction will not release the locks or memory it has acquired until it has been finished. The idiomatic use of transactions in Neo4j is to use a try-finally block, starting the transaction and then try to perform the write operations. The last operation

in the try block should mark the transaction as successful while the finally block should finish the transaction. Finishing the transaction will perform commit or rollback depending on the success status.

Caution

All modifications performed in a transaction are kept in memory. This means that very large updates have to be split into several top level transactions to avoid running out of memory. It must be a top level transaction since splitting up the work in many nested transactions will just add all the work to the top level transaction.

In an environment that makes use of *thread pooling* other errors may occur when failing to finish a transaction properly. Consider a leaked transaction that did not get finished properly. It will be tied to a thread and when that thread gets scheduled to perform work starting a new (what looks to be a) top level transaction it will actually be a nested transaction. If the leaked transaction state is “marked for rollback” (which will happen if a deadlock was detected) no more work can be performed on that transaction. Trying to do so will result in error on each call to a write operation.

3.2. Isolation levels

By default a read operation will read the last committed value unless a local modification within the current transaction exist. The default isolation level is very similar to `READ_COMMITTED`: reads do not block or take any locks so non-repeatable reads can occur. It is possible to achieve a stronger isolation level (such as `REPETABLE_READ` and `SERIALIZABLE`) by manually acquiring read and write locks.

3.3. Default locking behavior

- When adding, changing or removing a property on a node or relationship a write lock will be taken on the specific node or relationship.
- When creating or deleting a node a write lock will be taken for the specific node.
- When creating or deleting a relationship a write lock will be taken on the specific relationship and both its nodes.

The locks will be added to the transaction and released when the transaction finishes.

3.4. Deadlocks

Since locks are used it is possible for deadlocks to happen. Neo4j will however detect any deadlock (caused by acquiring a lock) before they happen and throw an exception. Before the exception is thrown the transaction is marked for rollback. All locks acquired by the transaction are still being held but will be released when the transaction is finished (in the finally block as pointed out earlier). Once the locks are released other transactions that were waiting for locks held by the transaction causing the deadlock can proceed. The work performed by the transaction causing the deadlock can then be retried by the user if needed.

Experiencing frequent deadlocks is an indication of concurrent write requests happening in such a way that it is not possible to execute them while at the same time live up to the intended isolation

and consistency. The solution is to make sure concurrent updates happen in a reasonable way. For example given two specific nodes (A and B), adding or deleting relationships to both these nodes in random order for each transaction will result in deadlocks when there are two or more transactions doing that concurrently. One solution is to make sure that updates always happens in the same order (first A then B). Another solution is to make sure that each thread/transaction does not have any conflicting writes to a node or relationship as some other concurrent transaction. This can for example be achieved by letting a single thread do all updates of a specific type.

Important

Deadlocks caused by the use of other synchronization than the locks managed by Neo4j can still happen. Since all operations in the Neo4j API are thread safe unless specified otherwise, there is no need for external synchronization. Other code that requires synchronization should be synchronized in such a way that it never performs any Neo4j operation in the synchronized block.

3.5. Delete semantics

When deleting a node or a relationship all properties for that entity will be automatically removed but the relationships of a node will not be removed.

Caution

Neo4j enforces a constraint (upon commit) that all relationships must have a valid start node and end node. In effect this means that trying to delete a node that still has relationships attached to it will throw an exception upon commit. It is however possible to choose in which order to delete the node and the attached relationships as long as no relationships exist when the transaction is committed.

The delete semantics can be summarized in the following bullets:

- All properties of a node or relationship will be removed when it is deleted.
- A deleted node can not have any attached relationships when the transaction commits.
- It is possible to acquire a reference to a deleted relationship or node that has not yet been committed.
- Any write operation on a node or relationship after it has been deleted (but not yet committed) will throw an exception
- After commit trying to acquire a new or work with an old reference to a deleted node or relationship will throw an exception.

Chapter 4. Neo4j Server

4.1. Server Installation

Neo4j can be installed as a server, running either as a headless application or system service.

1. Download the latest release from http://neo4j.org/download
   - select the appropriate version for your platform
2. Extract the contents of the archive
   - refer to the top-level extracted directory as `NEO4J-HOME`
3. Use the scripts in the `bin` directory
   - for Linux/MacOS, run `$NEO4J_HOME/bin/neo4j start`
   - for Windows, double-click on `%NEO4J_HOME%\bin\Neo4j.bat`
4. Refer to the packaged information in the `doc` directory for details

4.1.1. As a Windows service

With administrative rights, Neo4j can be installed as a Windows service.

1. Click Start –> All Programs –> Accessories
2. Right click Command Prompt –> Run as Administrator
3. Provide authorization and/or the Administrator password
4. Navigate to `%NEO4J_HOME%`
5. Run `bin\Neo4j.bat install`

To uninstall, run `bin\Neo4j.bat remove` as Administrator.

To query the status of the service, run `bin\Neo4j.bat query`

To start/stop the service from the command prompt, run `bin\Neo4j.bat +action+`

4.1.2. Linux Service

Neo4j can participate in the normal system startup and shutdown process. The following procedure should work on most popular Linux distributions:

1. `cd $NEO4J_HOME`
2. `sudo ./bin/neo4j install`
   - if asked, enter your password to gain super-user privileges

3. `service neo4jd status`
   - should indicate that the server is not running
4. `service neo4jd start`
   - will start the server

4.1.3. Macintosh Service

Neo4j can be installed as a Mac launchd job:

1. `cd $NEO4J_HOME`
2. `sudo ./bin/neo4j install`
   - if asked, enter your password to gain super-user privileges
3. `launchctl load ~/Library/LaunchAgents/wrapper.neo4jd.plist`
   - needed to tell launchd about the "job"
4. `launchctl list | grep neo`
   - should reveal the launchd "wrapper.neo4jd" job for running the Neo4j Server
5. `launchctl start wrapper.neo4jd`
   - to start the Neo4j Server under launchd control
6. `./bin/neo4j status`
   - should indicate that the server is running

4.1.4. Multiple Server instances on one machine

Neo4j can be set up to run as several instances on one machine, providing for instance several databases for development. To configure, install two instances of the Neo4j Server in two different directories. Before running the Windows install or startup, change in conf/neo4j-wrapper.conf

```
# Name of the service for the first instance
wrapper.name=neo4j_1
```

and for the second instance

```
# Name of the service for the second instance
wrapper.name=neo4j_2
```

in order not to get name clashes installing and starting the instances as services.

Also, the port numbers for the web administration and the servers should be changed to non-clashing values in conf/neo4j-server.properties:

Server 1 (port 7474):

```
org.neo4j.server.webserver.port=7474
```

```
org.neo4j.server.webadmin.data.uri=http://localhost:7474/db/data/

org.neo4j.server.webadmin.management.uri=http://localhost:7474/db/manage/
```

Server 2 (port 7475):

```
org.neo4j.server.webserver.port=7475

org.neo4j.server.webadmin.data.uri=http://localhost:7475/db/data/

org.neo4j.server.webadmin.management.uri=http://localhost:7475/db/manage/
```

4.2. Server Configuration

Quick info

- The server's primary configuration file is found under conf/neo4j-server.properties
- The conf/log4j.properties file contains the default server logging configuration
- Low-level performance tuning parameters are found in conf/neo4j.properties

The main configuration file for the server can be found:

```
conf/neo4j-server.properties
```

This file contains several important settings, and although the defaults are sensible administrators might choose to make changes (especially to the port settings).

Set the location on disk of the database directory

```
org.neo4j.server.database.location=data/graph.db
```

Note that on Windows systems, absolute locations including drive letters need to read "c:/data/db"

Specifying the port for the HTTP server that supports data, administrative, and UI access:

```
org.neo4j.server.webserver.port=7474
```

Set the location of the round-robin database directory which gathers metrics on the running server instance.

```
org.neo4j.server.webadmin.rrdb.location=data/graph.db/../rrd
```

Set the URI path for the REST data API through which the database is accessed. For non-local access, consider to put in the external hostname of your server instead of localhost, e.g. http://my.host:7474/db/data .

```
org.neo4j.server.webadmin.data.uri=http://localhost:7474/db/data/
```

The management URI for the administration API that the Webadmin tool uses. If you plan to connect to the Webadmin from other than localhost, put in the external hostname of your server instead of localhost, e.g. http://my.host:7474/db/manage .

```
org.neo4j.server.webadmin.management.uri=http://localhost:7474/db/manage
```

Low-level performance tuning parameters can be explicitly set by referring to the following property:

```
org.neo4j.server.db.tuning.properties=neo4j.properties
```

If this property isn't set, the server will look for a file called `neo4j.properties` in the same directory as the `neo4j-server.properties` file.

If this property isn't set, and there is no `neo4j.properties` file in the default configuration directory, then the server will log a warning. Subsequently at runtime the database engine will attempt tune itself based on the prevailing conditions.

The default log4j.properties file uses a rolling appender and outputs logs by default to the data/log directory. Most deployments will choose to use their own configuration here to meet local standards. During development, much useful information can be found in the logs so some form of logging to disk is well worth keeping.

```
conf/log4j.properties
```

The fine-tuning of the low-level Neo4j graph database engine is specified in a separate properties file.

```
conf/neo4j.properties
```

The graph database engine has a range of performance tuning options which are enumerated in Section 4.6, "Tuning the server performance". Note that other factors than Neo4j tuning should be considered when performance tuning a server, including general server load, memory and file contention, and even garbage collection penalties on the JVM, though such considerations are beyond the scope of this configuration document.

4.3. Setup for remote debugging

In order to configure the Neo4j server for remote debugging sessions, the java debugging parameters need to be passed to the java process through the configuration. They live in

```
conf/neo4j-wrapper.properties
```

In order to specify the parameters, add a line for the additional java arguments to read.

```
# Java Additional Parameters
wrapper.java.additional.1=-Dorg.neo4j.server.properties=conf/neo4j-server.properties
wrapper.java.additional.2=-Dlog4j.configuration=file:conf/log4j.properties
wrapper.java.additional.3=-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005 -Xdebug-Xnoagent-
```

which will start a Neo4j server ready for remote debugging attachement at localhost and port 5005. Use these parameters to attach to the process after starting the server from Eclipse, IntelliJ or your remote debugger of choice.

4.4. Starting the Neo4j server in high availability mode

Note

The High Availability features are only available in the Neo4j Enterprise Edition.

To run the Neo4j server in high availability mode there are two things you need to do. You have to configure the server to start up the database in high availability mode and you have to configure the Neo4j database for operating in high availability mode.

Instructing the server to start the database in high availability mode is as easy as setting the `org.neo4j.server.database.mode` property in the server properties file (`conf/neo-server.properties`) to `ha`. The default value for this parameter is `single`, which will start the database in standalone mode without participating in a cluster, still giving you Online Backup.

Configuring the Neo4j database for operating in high availability mode requires specifying a few properties in `conf/neo4j.properties`. First you need to specify `ha.machine_id`, this is a positive integer id that uniquely identifies this server in the cluster.

Example: `ha.machine_id = 1`

Then you have to specify `ha.zoo_keeper_servers`, this is a comma separated list of hosts and ports for communicating with each member of the Neo4j Coordinator cluster.

For example: `ha.zoo_keeper_servers = neo4j-manager-01:2180,neo4j-manager-02:2180,neo4j-manager-03:2180`.

You can also, optionally, configure the `ha.cluster_name`. This is the name of the cluster this instance is supposed to join. Accepted characters are alphabetical, numerical, dot, dash, and underscore. This configuration is useful if you have multiple Neo4j HA clusters managed by the same Coordinator cluster.

Example: `ha.cluster_name = my_neo4j_ha_cluster`

4.4.1. Starting a Neo4j Coordinator

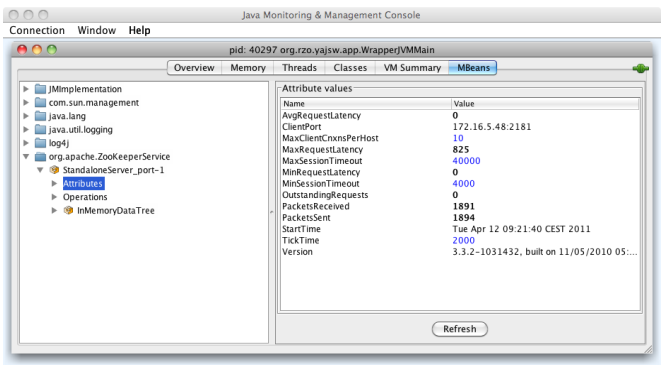
A Neo4j Coordinator cluster provides the Neo4j HA Data cluster with reliable coordination of lifecycle activities, like electing the master. Neo4j Server includes everything needed for running a Neo4j Coordinator.

Configuration of a Coordinator is specified in these files:

- `conf/coord.cfg` - coordinator operational settings
- `data/coordinator/myid` - unqiue identification of the coordinator

Once a Neo4j Coordinator instance has been configured, you can use the `bin/neo4j-coordinator` command to start the Neo4j Coordinator server on all desired servers with the same configuration, just changing the `data/coordinator/myid` to unique numbers. You can check that the coordinator is up by running `jconsole` , attaching to the JVM and check for org.apache.zookeeper MBeans.

Figure 4.1. Neo4j Coordinator MBeans View



4.4.2. Starting the Neo4j Server

Once the desired neo4j Coordinators are up and running, you are ready to start your Neo4j HA instance using `bin/neo4j start`. The details of the HA logs are available in the `messages.log` of the graph database data directory, normally `data/graph.db/mesages.log`. You should see an entry like

```
Tue Apr 12 09:25:58 CEST 2011: MasterServer communication server started and bound to 6361
Tue Apr 12 09:25:58 CEST 2011: Started as master
Tue Apr 12 09:25:58 CEST 2011: master-rebound set to 1
```

4.5. Server Plugins

Quick info

- The server's functionality can be extended by adding plugins. Plugins are user-specified code which extend the capabilities of the database, nodes, or relations. The neo4j server will then advertise the plugin functionality within representations as clients interact via HTTP.

Plugins provide an easy way to extend the Neo4j REST API with new functionality, without the need to invent your own API. Think of plugins as server-side scripts that can add functions for retrieving and manipulating nodes, relationships, paths or properties.

Warning

If you want to have full control over your API, and are willing to put in the effort, and understand the risks, then Neo4j server also provides hooks for unmanaged extensions based on JAX-RS.

To create a plugin, your code must inherit from the ServerPlugin class, and ensure that it can produce an (Iterable of) Node, Relationship or Path, specify parameters, a point of extension and of course the application logic. An example of a plugin which augments the database (as opposed to nodes or relations) follows:

```
@Description( "An extension to the Neo4j Server for getting all nodes or relationships" )
public class GetAll extends ServerPlugin
{
  @Name( "get_all_nodes" )
  @Description( "Get all nodes from the Neo4j graph database" )
  @PluginTarget( GraphDatabaseService.class )
  public Iterable<Node> getAllNodes( @Source GraphDatabaseService graphDb )
  {
      return graphDb.getAllNodes();
  }


  @Description( "Get all relationships from the Neo4j graph database" )
  @PluginTarget( GraphDatabaseService.class )
  public Iterable<Relationship> getAllRelationships( @Source GraphDatabaseService graphDb )
  {
      return new NestingIterable<Relationship, Node>( graphDb.getAllNodes() )
      {
          @Override
          protected Iterator<Relationship> createNestedIterator( Node item )
```

```
            {
                return item.getRelationships( Direction.OUTGOING ).iterator();
            }
        };
    }

    @Description( "Find the shortest path between two nodes." )
    @PluginTarget( Node.class )
    public Iterable<Path> shortestPath(
            @Source Node source,
            @Description( "The node to find the shortest path to." ) @Parameter( name = "target" ) Node target,
            @Description( "The relationship types to follow when searching for the shortest path(s). Order is i
            @Description( "The maximum path length to search for, default value (if omitted) is 4." ) @Paramete
    {
        Expander expander;
        if ( types == null )
        {
            expander = Traversal.expanderForAllTypes();
        }
        else
        {
            expander = Traversal.emptyExpander();
            for ( int i = 0; i < types.length; i++ )
            {
                expander = expander.add( DynamicRelationshipType.withName( types[i] ) );
            }
        }
        PathFinder<Path> shortestPath = GraphAlgoFactory.shortestPath( expander, depth == null ? 4
                : depth.intValue() );
        return shortestPath.findAllPaths( source, target );
    }
}
```

To deploy the code, simply compile it into a .jar file and place it onto the server classpath (which by convention is the plugins directory under the neo4j server home directory). The .jar file must include the file META-INF/services/org.neo4j.server.plugins.ServerPlugin with the fully qualified name of the implementation class. In this case, we'd have only a single entry in our config file, though multiple entries are allowed, each on a separate line:

```
org.neo4j.server.examples.GetAll
# Any other plugins in the same jar file must be listed here
```

The code above makes an extension visible in the database representation (via the @PluginTarget annotation) whenever it is served from the Neo4j Server. Simply changing the @PluginTarget parameter to Node.class or Relationship.class allows us to target those parts of the datamodel should we wish. The functionality extensions provided by the plugin are automatically advertised in representations on the wire. For example, clients can discover the extension implemented by the above plugin easily by examining the representations they receive as responses from the server, e.g. by performing a GET on the default database URI:

```
curl -v http://localhost:7474/db/data/
```

The response to the GET request will contain (by default) a JSON container that itself contains a container called "extensions" where the available plugins are listed. In the following case, we only have the GetAll plugin registered with the server, so only its extension functionality is available:

```
{
"extensions-info" : "http://localhost:7474/db/data/ext",
"node" : "http://localhost:7474/db/data/node",
"node_index" : "http://localhost:7474/db/data/index/node",
"relationship_index" : "http://localhost:7474/db/data/index/relationship",
```

```
"reference_node" : "http://localhost:7474/db/data/node/0",
"extensions_info" : "http://localhost:7474/db/data/ext",
"extensions" : {
  "GetAll" : {
    "get_all_nodes" : "http://localhost:7474/db/data/ext/GetAll/graphdb/get_all_nodes",
    "get_all_relationships" : "http://localhost:7474/db/data/ext/GetAll/graphdb/getAllRelationships"
  }
}
```

Performing a GET on one of the two extension URIs gives back the meta information about the service:

```
curl http://localhost:7474/db/data/ext/GetAll/graphdb/get_all_nodes
```

```
{
  "extends" : "graphdb",
  "description" : "Get all nodes from the Neo4j graph database",
  "name" : "get_all_nodes",
  "parameters" : [ ]
}
```

To use it, just POST to this URL, with parameters as specified in the description (though there are none in this case).

Through this model, any plugin can naturally fit into the general hypermedia scheme that Neo4j espouses - meaning that clients can still take advantage of abstractions like Nodes, Relationships and Paths with a straightforward upgrade path as servers are enriched with plugins (old clients don't break).

4.6. Tuning the server performance

At the heart of the Neo4j server is a regular Neo4j storage engine instance. That engine can be tuned in the same way as the other embedded configurations, using the same file format. The only difference is that the server must be told where to find the fine-tuning configuration.

Quick info

- The neo4j.properties file is a standard configuration file that databases load in order to tune their memory use and caching strategies.
- See Section 2.1, "Caches in Neo4j" for more information.

4.6.1. Specifying Neo4j tuning properties

The `conf/neo4j-server.properties` file in the server distribution, is the main configuration file for the server. In this file we can specify a second properties file that contains the database tuning settings (that is, the `neo4j.properties` file). This is done by setting a single property to point to a valid `neo4j.properties` file:

```
org.neo4j.server.db.tuning.properties={neo4j.properties file}
```

On restarting the server the tuning enhancements specified in the `neo4j.properties` file will be loaded and configured into the underlying database engine.

4.6.2. Specifying JVM tuning properties

Tuning the standalone server is achieved by editing the `neo4j-wrapper.conf` file in the `conf` directory of `NEO4J_HOME`.

Edit the following properties:

Table 4.1. neo4j-wrapper.conf JVM tuning properties

Property Name	Meaning
`wrapper.java.initmemory`	initial heap size (in MB)
`wrapper.java.maxmemory`	maximum heap size (in MB)
`wrapper.java.additional.N`	additional literal JVM parameter, where N is a number for each

For more information on the tuning properties, see Section 2.2, “JVM Settings”.

4.7. Unmanaged Extensions

Quick info

- Danger Men at Work! The unmanaged extensions are a way of deploying arbitrary JAX-RS code into the Neo4j server.
- The unmanaged extensions are exactly that: unmanaged. If you drop poorly tested code into the server, it’s highly like you’ll degrade its performance, so be careful.

Some projects want extremely fine control over their server-side code. For this we’ve introduced an unmanaged extension API. It’s a sharp tool, allowing users to deploy arbitrary JAX-RS classes to the server and so you should be careful when thinking about using this. In particular you should understand that it’s easy to consume lots of heap space on the server and hinder performance if you’re not careful. Still, if you understand the disclaimer, then you load your JAX-RS classes into the Neo4j server simply by adding adding a @Context annotation to your code, compiling against the Neo4j server jar, and then adding your classes to the runtime classpath (just drop it in the lib directory of the Neo4j server). In return you get access to the hosted environment of the Neo4j server like logging through the org.neo4j.server.logging.Logger.

In your code, you get access to the underlying GraphDatabaseService through the @Context annotation like so:

```
public MyCoolService(@Context GraphDatabaseService database)
{
  // Have fun here, but be safe!
}
```

Remember, the unmanaged API is a very sharp tool. It’s all to easy to compromise the server by deploying code this way, so think first and see if you can’t use the managed extensions in preference. However, a number of context parameters can be automatically provided for you, like the reference to the database.

In order to specify the mount point of your extension, a full class looks like

```
@Path( "/helloworld" )
public class HelloWorldResource
{

    private final GraphDatabaseService database;

    public HelloWorldResource( @Context GraphDatabaseService database)
    {
      this.database = database;
    }

    @GET
    @Produces( MediaType.TEXT_PLAIN )
    @Path( "/{nodeId}" )
    public Response hello( @PathParam( "nodeId" ) long nodeId )
    {
        // Do stuff with the database
        return Response.status( Status.OK ).entity(
                ( "Hello World, nodeId=" + nodeId).getBytes() ).build();
    }
}
```

Build this code, and place the resulting jar file (and any custom dependencies) into the $NEO4J_SERVER_HOME/plugins directory, and include this class in neo4j-server.propertie, like so:

```
#Comma separated list of JAXRS packages containing JAXRS Resource, one package name for each mountpoint.
org.neo4j.server.thirdparty_jaxrs_classes=org.neo4j.examples.server.unmanaged=/examples/unmanaged
```

```
curl http://localhost:7474/examples/unmanaged/helloworld/123
```

which results in

```
Hello World, nodeId=123
```

Chapter 5. Indexing

Indexing in Neo4j can be done in two different ways:

1. The database itself is a *natural index* consisting of its relationships of different types between nodes. For example a tree structure can be layered on top of the data and used for index lookups performed by a traverser.

2. Separate index engines can be used, with Apache Lucene [http://lucene.apache.org/java/3_0_1/index.html] being the default backend included with Neo4j.

This chapter demonstrate how to use the second type of indexing, focussing on Lucene.

5.1. Introduction

Indexing operations are part of the Neo4j index API [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/index/package-summary.html].

Each index is tied to a unique, user-specified name (for example "first_name" or "books") and can index either nodes [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/Node.html] or relationships [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/Relationship.html].

The default index implementation is provided by the `neo4j-lucene-index` component, which is included in the standard Neo4j download. It can also be downloaded separately from http://repo1.maven.org/maven2/org/neo4j/neo4j-lucene-index/ . For Maven users, the `neo4j-lucene-index` component has the coordinates `org.neo4j:neo4j-lucene-index` and should be used with the same version of `org.neo4j:neo4j-kernel`. Different versions of the index and kernel components are not compatible in the general case. Both components are included transitively by the `org.neo4j:neo4j:pom` artifact which makes it simple to keep the versions in sync.

Note

All modifying index operations must be performed inside a transaction, as with any mutating operation in Neo4j.

5.2. Create

An index is created if it doesn't exist when you ask for it. Unless you give it a custom configuration, it will be created with default configuration and backend.

To set the stage for our examples, let's create some indices to begin with:

```
IndexManager index = graphDb.index();
Index<Node> actors = index.forNodes( "actors" );
Index<Node> movies = index.forNodes( "movies" );
RelationshipIndex roles = index.forRelationships( "roles" );
```

This will create two node indices and one relationship index with default configuration. See Section 5.8, "Relationship indices" for more information specific to relationship indices.

See Section 5.10, "Configuration and fulltext indices" for how to create *fulltext* indices.

You can also check if an index exists like this:

```
IndexManager index = graphDb.index();
boolean indexExists = index.existsForNodes( "actors" );
```

5.3. Delete

Indices can be deleted. When deleting, the entire contents of the index will be removed as well as its associated configuration. A new index can be created with the same name at a later point in time.

```
IndexManager index = graphDb.index();
Index<Node> actors = index.forNodes( "actors" );
actors.delete();
```

Note that the actual deletion of the index is made during the commit of *the surrounding transaction*. Calls made to such an index instance after delete() [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/index/Index.html#delete%28%29] has been called are invalid inside that transaction as well as outside (if the transaction is successful), but will become valid again if the transaction is rolled back.

5.4. Add

Each index supports associating any number of key-value pairs with any number of entities (nodes or relationships), where each association between entity and key-value pair is performed individually. To begin with, let's add a few nodes to the indices:

```
// Actors
Node reeves = graphDb.createNode();
actors.add( reeves, "name", "Keanu Reeves" );
Node bellucci = graphDb.createNode();
actors.add( bellucci, "name", "Monica Bellucci" );
// multiple values for a field
actors.add( bellucci, "name", "La Bellucci" );
// Movies
Node theMatrix = graphDb.createNode();
movies.add( theMatrix, "title", "The Matrix" );
movies.add( theMatrix, "year", 1999 );
Node theMatrixReloaded = graphDb.createNode();
movies.add( theMatrixReloaded, "title", "The Matrix Reloaded" );
movies.add( theMatrixReloaded, "year", 2003 );
Node malena = graphDb.createNode();
movies.add( malena, "title", "Malèna" );
movies.add( malena, "year", 2000 );
```

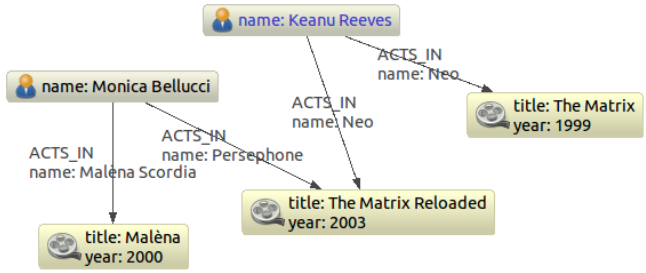
Note that there can be multiple values associated with the same entity and key.

Next up, we'll create relationships and index them as well:

```
// we need a relationship type
DynamicRelationshipType ACTS_IN = DynamicRelationshipType.withName( "ACTS_IN" );
// create relationships
Relationship role1 = reeves.createRelationshipTo( theMatrix, ACTS_IN );
roles.add( role1, "name", "Neo" );
Relationship role2 = reeves.createRelationshipTo( theMatrixReloaded, ACTS_IN );
roles.add( role2, "name", "Neo" );
Relationship role3 = bellucci.createRelationshipTo( theMatrixReloaded, ACTS_IN );
roles.add( role3, "name", "Persephone" );
Relationship role4 = bellucci.createRelationshipTo( malena, ACTS_IN );
```

```
roles.add( role4, "name", "Malèna Scordia" );
```

Assuming we set the same key-value pairs as properties as well, our example graph looks like this:



5.5. Remove

Removing [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/index/Index.html#remove%28T,%20java.lang.String,%20java.lang.Object%29] from an index is similar to adding, but can be done by supplying one of the following combinations of arguments:

- entity
- entity, key
- entity, key, value

```
// completely remove bellucci from the actors index
actors.remove( bellucci );
// remove any "name" entry of bellucci from the actors index
actors.remove( bellucci, "name" );
// remove the "name" -> "La Bellucci" entry of bellucci
actors.remove( bellucci, "name", "La Bellucci" );
```

5.6. Update

Important

To update an index entry, old one must be removed and a new one added.

Remember that a node or relationship can be associated with any number of key-value pairs in an index, which means that you can index a node or relationship with many key-value pairs that have the same key. In the case where a property value changes and you'd like to update the index, it's not enough to just index the new value - you'll have to remove the old value as well.

Here's a code example for that demonstrates how it's done:

```
// create a node with a property
Node fishburn = graphDb.createNode();
fishburn.setProperty( "name", "Fishburn" );
// index it
actors.add( fishburn, "name", fishburn.getProperty( "name" ) );
// update the index entry
actors.remove( fishburn, "name", fishburn.getProperty( "name" ) );
fishburn.setProperty( "name", "Laurence Fishburn" );
```

```
actors.add( fishburn, "name", fishburn.getProperty( "name" ) );
```

5.7. Search

An index can be searched in two ways, get [http://components.neo4j.org/neo4j1.3/apidocs/org/neo4j/graphdb/index/Index.html#get%28java.lang.String,%20java.lang.Object%29] and query [http://components.neo4j.org/neo4j1.3/apidocs/org/neo4j/graphdb/index/Index.html#query%28java.lang.String,%20java.lang.Object%29]. The `get` method will return exact matches to the given key-value pair, whereas `query` exposes querying capabilities directly from the backend used by the index. For example the Lucene query syntax [http://lucene.apache.org/java/3_0_1/queryparsersyntax.html] can be used directly with the default indexing backend.

5.7.1. Get

This is how to search for a single exact match:

```
IndexHits<Node> hits = actors.get( "name", "Keanu Reeves" );
Node reeves = hits.getSingle();
```

IndexHits [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/index/IndexHits.html] is an `Iterable` with some additional useful methods. For example getSingle() [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/index/IndexHits.html#getSingle%28%29] returns the first and only item from the result iterator, or `null` if there isn't any hit.

Here's how to get a single relationship by exact matching and retrieve its start and end nodes:

```
Relationship persephone = roles.get( "name", "Persephone" ).getSingle();
Node actor = persephone.getStartNode();
Node movie = persephone.getEndNode();
```

Finally, we can iterate over all exact matches from a relationship index:

```
for ( Relationship role : roles.get( "name", "Neo" ) )
{
    // this will give us Reeves twice
    Node reeves = role.getStartNode();
}
```

Important

In you don't iterate through all the hits, IndexHits.close() [http://components.neo4j.org/neo4j1.3/apidocs/org/neo4j/graphdb/index/IndexHits.html#close%28%29] must be called explicitly.

5.7.2. Query

There are two query methods, one which uses a key-value signature where the value represents a query for values with the given key only. The other method is more generic and supports querying for more than one key-value pair in the same query.

Here's an example using the key-query option:

```
for ( Node actor : actors.query( "name", "*e*" ) )
{
```

```
    // This will return Reeves and Bellucci
}
```

In the following example the query uses multiple keys:

```
for ( Node movie : movies.query( "title:*Matrix* AND year:1999" ) )
{
    // This will return "The Matrix" from 1999 only.
}
```

Note

Beginning a wildcard search with "*" or "?" is discouraged by Lucene, but will nevertheless work.

Caution

You can't have *any whitespace* in the search term with this syntax. See Section 5.11.3, "Querying with Lucene Query objects" for how to do that.

5.8. Relationship indices

An index for relationships is just like an index for nodes, extended by providing support to constrain a search to relationships with a specific start and/or end nodes These extra methods reside in the RelationshipIndex [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/index/RelationshipIndex.html] interface which extends Index<Relationship> [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/index/Index.html].

Example of querying a relationship index:

```
// find relationships filtering on start node
// using exact matches
IndexHits<Relationship> reevesAsNeoHits;
reevesAsNeoHits = roles.get( "name", "Neo", reeves, null );
Relationship reevesAsNeo = reevesAsNeoHits.iterator().next();
reevesAsNeoHits.close();
// find relationships filtering on end node
// using a query
IndexHits<Relationship> matrixNeoHits;
matrixNeoHits = roles.query( "name", "*eo", null, theMatrix );
Relationship matrixNeo = matrixNeoHits.iterator().next();
matrixNeoHits.close();
```

And here's an example for the special case of searching for a specific relationship type:

```
// find relationships filtering on end node
// using a relationship type.
// this is how to add it to the index:
roles.add( reevesAsNeo, "type", reevesAsNeo.getType().name() );
// Note that to use a compound query, we can't combine committed
// and uncommitted index entries, so we'll commit before querying:
tx.success();
tx.finish();
// and now we can search for it:
IndexHits<Relationship> typeHits;
typeHits = roles.query( "type:ACTS_IN AND name:Neo", null, theMatrix );
Relationship typeNeo = typeHits.iterator().next();
typeHits.close();
```

Such an index can be useful if your domain has nodes with a very large number of relationships between them, since it reduces the search time for a relationship between two nodes. A good example where this approach pays dividends is in time series data, where we have readings represented as a relationship per occurrence.

5.9. Scores

The `IndexHits` interface exposes scoring [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/index/IndexHits.html#currentScore%28%29] so that the index can communicate scores for the hits. Note that the result is not sorted by the score unless you explicitly specify that. See Section 5.11.2, “Sorting” for how to sort by score.

```
IndexHits<Node> hits = movies.query( "title", "The*" );
for ( Node movie : hits )
{
    System.out.println( movie.getProperty( "title" ) + " " + hits.currentScore() );
}
```

5.10. Configuration and fulltext indices

At the time of creation extra configuration can be specified to control the behavior of the index and which backend to use. For example to create a Lucene fulltext index:

```
IndexManager index = graphDb.index();
Index<Node> fulltextMovies = index.forNodes( "movies-fulltext",
        MapUtil.stringMap( "provider", "lucene", "type", "fulltext" ) );
fulltextMovies.add( theMatrix, "title", "The Matrix" );
fulltextMovies.add( theMatrixReloaded, "title", "The Matrix Reloaded" );
// search in the fulltext index
Node found = fulltextMovies.query( "title", "reloAdEd" ).getSingle();
```

Tip

In order to search for tokenized words, the `query` method has to be used. The `get` method will only match the full string value, not the tokens.

The configuration of the index is persisted once the index has been created. The `provider` configuration key is interpreted by Neo4j, but any other configuration is passed onto the backend index (e.g. Lucene) to interpret.

Table 5.1. Lucene indexing configuration parameters

Parameter	Possible values	Effect
`type`	`exact`, `fulltext`	`exact` is the default and uses a Lucene keyword analyzer [http://lucene.apache.org/java/3_0_1/api/core/org/apache/lucene/analysis/KeywordAnalyzer.html]. `fulltext` uses a white-space tokenizer in its analyzer.
`to_lower_case`	`true`, `false`	This parameter goes together with `type`: `fulltext` and converts values to lower case during both additions and querying, making the index case insensitive. Defaults to `true`.

Parameter	Possible values	Effect
`analyzer`	the full class name of an Analyzer [http://lucene.apache.org/java/3_0_1/api/core/org/apache/lucene/analysis/Analyzer.html]	Overrides the `type` so that a custom analyzer can be used. Note: `to_lower_case` still affects lowercasing of string queries. If the custom analyzer uppercases the indexed tokens, string queries will not match as expected.

5.11. Extra features for Lucene indices

5.11.1. Numeric ranges

Lucene supports smart indexing of numbers, querying for ranges and sorting such results, and so does its backend for Neo4j. To mark a value so that it is indexed as a numeric value, we can make use of the ValueContext [http://components.neo4j.org/neo4j-lucene-index/1.3/apidocs/org/neo4j/index/lucene/ValueContext.html] class, like this:

```
movies.add( theMatrix, "year-numeric", new ValueContext( 1999L ).indexNumeric() );
movies.add( theMatrixReloaded, "year-numeric", new ValueContext( 2003L ).indexNumeric() );

// Query for range
long startYear = 1997;
long endYear = 2001;
hits = movies.query( NumericRangeQuery.newLongRange( "year-numeric", startYear, endYear, true, true ) );
```

Note

Values that are indexed numerically must be queried using NumericRangeQuery [http://lucene.apache.org/java/3_0_1/api/core/org/apache/lucene/search/NumericRangeQuery.html].

5.11.2. Sorting

Lucene performs sorting very well, and that is also exposed in the index backend, through the QueryContext [http://components.neo4j.org/neo4j-lucene-index/1.3/apidocs/org/neo4j/index/lucene/QueryContext.html] class:

```
hits = movies.query( "title", new QueryContext( "*" ).sort( "title" ) );
for ( Node hit : hits )
{
    // all movies with a title in the index, ordered by title
}
// or
hits = movies.query( new QueryContext( "title:*" ).sort( "year", "title" ) );
for ( Node hit : hits )
{
    // all movies with a title in the index, ordered by year, then title
}
```

We sort the results by relevance (score) like this:

```
hits = movies.query( "title", new QueryContext( "The*" ).sortByScore() );
for ( Node movie : hits )
{
```

```
    // hits sorted by relevance (score)
}
```

5.11.3. Querying with Lucene Query objects

Instead of passing in Lucene query syntax queries, you can instantiate such queries programmatically and pass in as argument, for example:

```
// a TermQuery will give exact matches
Node actor = actors.query( new TermQuery( new Term( "name", "Keanu Reeves" ) ) ).getSingle();
```

Note that the TermQuery [http://lucene.apache.org/java/3_0_1/api/core/org/apache/lucene/search/TermQuery.html] is basically the same thing as using the `get` method on the index.

This is how to perform *wildcard* searches using Lucene Query Objects:

```
hits = movies.query( new WildcardQuery( new Term( "title", "The Matrix*" ) ) );
for ( Node movie : hits )
{
    System.out.println( movie.getProperty( "title" ) );
}
```

Note that this allows for whitespace in the search string.

5.11.4. Compound queries

Lucene supports querying for multiple terms in the same query, like so:

```
hits = movies.query( "title:*Matrix* AND year:1999" );
```

Caution

Compound queries can't search across committed index entries and those who haven't got committed yet at the same time.

5.11.5. Default operator

The default operator (that is whether `AND` or `OR` is used in between different terms) in a query is `OR`. Changing that behavior is also done via the QueryContext [http://components.neo4j.org/neo4j-lucene-index/1.3/apidocs/org/neo4j/index/lucene/QueryContext.html] class:

```
QueryContext query = new QueryContext( "title:*Matrix* year:1999" ).defaultOperator( Operator.AND );
hits = movies.query( query );
```

5.11.6. Caching

If your index lookups becomes a performance bottle neck, caching can be enabled for certain keys in certain indices (key locations) to speed up get requests. The caching is implemented with an LRU [http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used] cache so that only the most recently accessed results are cached (with "results" meaning a query result of a get request, not a single entity). You can control the size of the cache (the maximum number of results) per index key.

```
Index<Node> index = graphDb.index().forNodes( "actors" );
( (LuceneIndex<Node>) index ).setCacheCapacity( "name", 300000 );
```

Caution

This setting is not persisted after shutting down the database. This means: set this value after each startup of the database if you want to keep it.

5.12. Batch insertion

Neo4j has a batch insertion mode intended for initial imports, which must run in a single thread and bypasses transactions and other checks in favor of performance. Indexing during batch insertion is done using BatchInserterIndex [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/index/BatchInserterIndex.html] which are provided via BatchInserterIndexProvider [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/index/BatchInserterIndexProvider.html]. An example:

```
BatchInserter inserter = new BatchInserterImpl( "target/neo4jdb-batchinsert" );
BatchInserterIndexProvider indexProvider = new LuceneBatchInserterIndexProvider( inserter );
BatchInserterIndex actors = indexProvider.nodeIndex( "actors", MapUtil.stringMap( "type", "exact" ) );
actors.setCacheCapacity( "name", 100000 );

Map<String, Object> properties = MapUtil.map( "name", "Keanu Reeves" );
long node = inserter.createNode( properties );
actors.add( node, properties );

// Make sure to shut down the index provider
indexProvider.shutdown();
inserter.shutdown();
```

The configuration parameters are the same as mentioned in Section 5.10, “Configuration and fulltext indices”.

5.12.1. Best practices

Here are some pointers to get the most performance out of `BatchInserterIndex`:

- Try to avoid flushing [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/index/BatchInserterIndex.html#flush%28%29] too often because each flush will result in all additions (since last flush) to be visible to the querying methods, and publishing those changes can be a performance penalty.
- Have (as big as possible) phases where one phase is either only writes or only reads, and don’t forget to flush after a write phase so that those changes becomes visible to the querying methods.
- Enable caching [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphdb/index/BatchInserterIndex.html#setCacheCapacity%28java.lang.String,%20int%29] for keys you know you’re going to do lookups for later on to increase performance significantly (though insertion performance may degrade slightly).

Chapter 6. Graph Algorithms

Neo4j graph algorithms is a component that contains Neo4j implementations of some common algorithms for graphs. It includes algorithms like:

- Shortest paths,
- all paths,
- all simple paths,
- Dijkstra and
- A*.

6.1. Introduction

The graph algorithms are found in the `neo4j-graph-algo` component, which is included in the standard Neo4j download.

- Javadocs: http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphalgo/package-summary.html
- Separate download: http://repo1.maven.org/maven2/org/neo4j/neo4j-graph-algo/
- Source code: https://github.com/neo4j/graphdb/tree/master/graph-algo

For Maven users, the component has the coordinates `org.neo4j:neo4j-graph-algo` and should be used with the same version of `org.neo4j:neo4j-kernel`. Different versions of the graph-algo and kernel components are not compatible in the general case. Both components are included transitively by the `org.neo4j:neo4j:pom` artifact which makes it simple to keep the versions in sync.

The starting point to find and use graph algorithms is GraphAlgoFactory [http://components.neo4j.org/neo4j/1.3/apidocs/org/neo4j/graphalgo/GraphAlgoFactory.html].

6.2. Path finding examples

Calculating the shortest path (least number of relationships) between two nodes:

```
Node startNode = graphDb.createNode();
Node middleNode1 = graphDb.createNode();
Node middleNode2 = graphDb.createNode();
Node middleNode3 = graphDb.createNode();
Node endNode = graphDb.createNode();
createRelationshipsBetween( startNode, middleNode1, endNode );
createRelationshipsBetween( startNode, middleNode2, middleNode3, endNode );

// Will find the shortest path between startNode and endNode via
// "MY_TYPE" relationships (in OUTGOING direction), like f.ex:
//
// (startNode)-->(middleNode1)-->(endNode)
//
PathFinder<Path> finder = GraphAlgoFactory.shortestPath(
```

```
        Traversal.expanderForTypes( ExampleTypes.MY_TYPE, Direction.OUTGOING ), 15 );
Iterable<Path> paths = finder.findAllPaths( startNode, endNode );
```

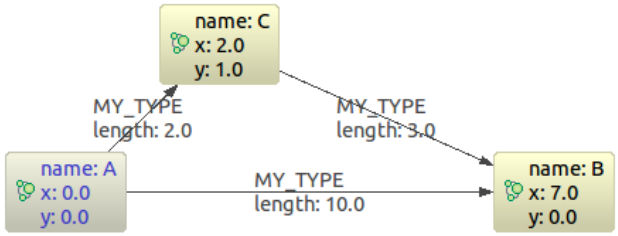
Using Dijkstra's algorithm [http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm] to calculate cheapest path between node A and B where each relationship can have a weight (i.e. cost) and the path(s) with least cost are found.

```
PathFinder<WeightedPath> finder = GraphAlgoFactory.dijkstra(
        Traversal.expanderForTypes( ExampleTypes.MY_TYPE, Direction.BOTH ), "cost" );

WeightedPath path = finder.findSinglePath( nodeA, nodeB );

// Get the weight for the found path
path.weight();
```

Using A* [http://en.wikipedia.org/wiki/A*_search_algorithm] to calculate the cheapest path between node A and B, where cheapest is for example the path in a network of roads which has the shortest length between node A and B. Here's our example graph:




```
Node nodeA = createNode( "name", "A", "x", 0d, "y", 0d );
Node nodeB = createNode( "name", "B", "x", 7d, "y", 0d );
Node nodeC = createNode( "name", "C", "x", 2d, "y", 1d );
Relationship relAB = createRelationship( nodeA, nodeC, "length", 2d );
Relationship relBC = createRelationship( nodeC, nodeB, "length", 3d );
Relationship relAC = createRelationship( nodeA, nodeB, "length", 10d );

EstimateEvaluator<Double> estimateEvaluator = new EstimateEvaluator<Double>()
{
    public Double getCost( final Node node, final Node goal )
    {
        double dx = (Double) node.getProperty( "x" ) - (Double) goal.getProperty( "x" );
        double dy = (Double) node.getProperty( "y" ) - (Double) goal.getProperty( "y" );
        double result = Math.sqrt( Math.pow( dx, 2 ) + Math.pow( dy, 2 ) );
        return result;
    }
};
PathFinder<WeightedPath> astar = GraphAlgoFactory.aStar(
        Traversal.expanderForAllTypes(),
        CommonEvaluators.doubleCostEvaluator( "length" ), estimateEvaluator );
WeightedPath path = astar.findSinglePath( nodeA, nodeB );
```

The full source code of the path finding examples are found at https://github.com/neo4j/graphdb/blob/master/graph-algo/src/test/java/examples/PathFindingExamplesTest.java.

Chapter 7. High Availability

Note

The High Availability features are only available in the Neo4j Enterprise Edition.

Neo4j High Availability or “Neo4j HA” provides the following two main features:

1. It enables a *fault-tolerant database architecture*, where several Neo4j slave databases can be configured to be exact replicas of a single Neo4j master database. This allows the end-user system to be fully functional and both read and write to the database in the event of hardware failure.

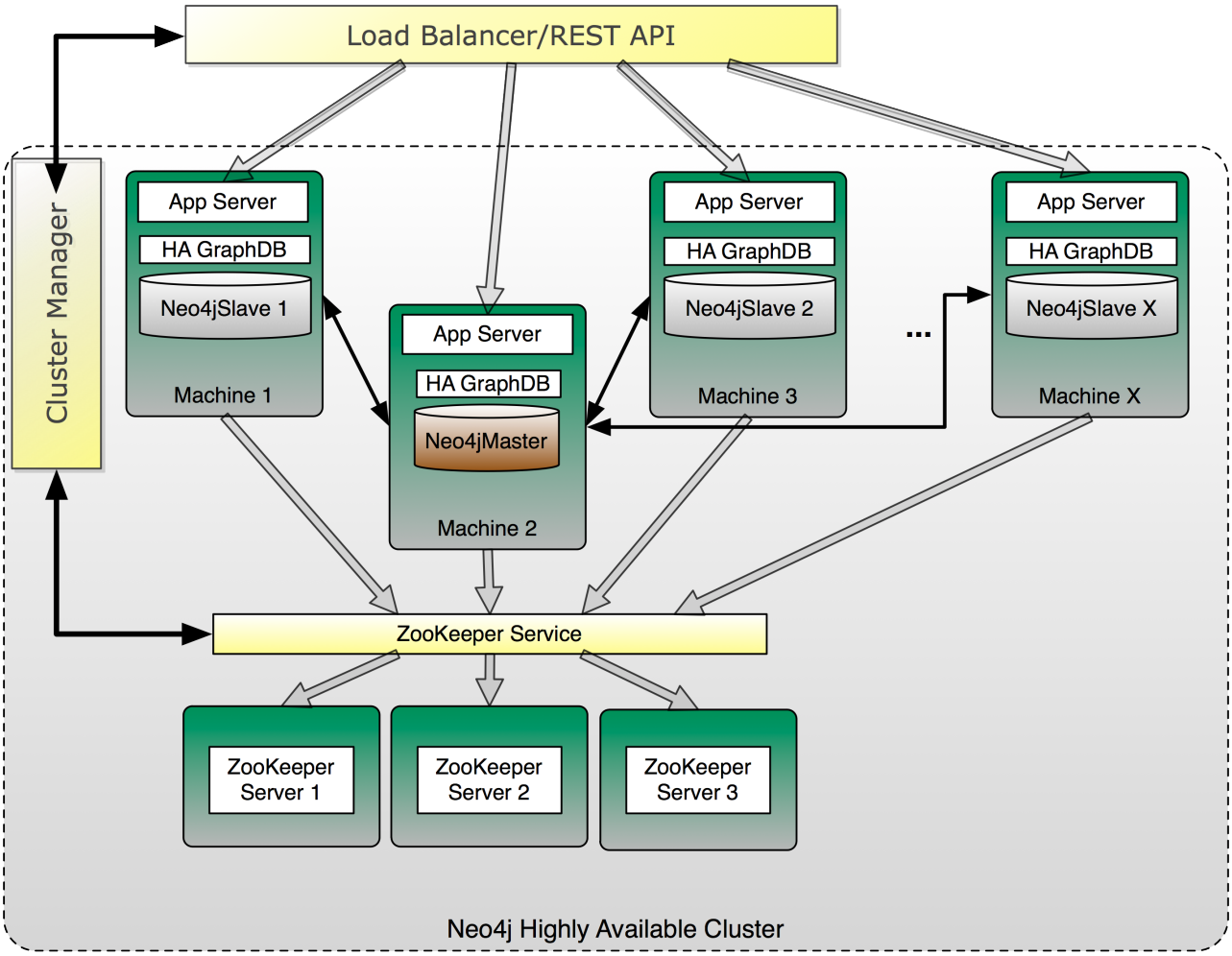
2. It enables a *horizontally scaling read-mostly architecture* that enables the system to handle more read load than a single Neo4j database instance can handle.

7.1. Architecture

Neo4j HA has been designed to make the transition from single machine to multi machine operation simple, by not having to change the already existing application.

Consider an existing application with Neo4j embedded and running on a single machine. To deploy such an application in a multi machine setup the only required change is to switch the creation of the `GraphDatabaseService` from `EmbeddedGraphDatabase` to `HighlyAvailableGraphDatabase`. Since both implement the same interface, no additional changes are required.

Figure 7.1. Typical setup when running multiple Neo4j instances in HA mode



When running Neo4j in HA mode there is always a single master and zero or more slaves. Compared to other master-slave replication setups Neo4j HA can handle writes on a slave so there is no need to redirect writes to the master.

A slave will handle writes by synchronizing with the master to preserve consistency. Updates will however propagate from the master to other slaves eventually so a write from one slave is not immediately visible on all other slaves. This is the only difference between multiple machines running in HA mode compared to single machine operation. All other ACID characteristics are the same.

7.2. Setup and configuration

Neo4j HA can be set up to accommodate differing requirements for load, fault tolerance and available hardware.

Within a cluster, Neo4j HA uses Apache ZooKeeper [1] for master election and propagation of general cluster and machine status information. ZooKeeper can be seen as a distributed coordination service. Neo4j HA requires a ZooKeeper service for initial master election, new master election (current master failing) and to publish general status information about the current Neo4j HA

[1] http://hadoop.apache.org/zookeeper/

cluster (for example when a machine joined or left the cluster). Read operations through the `GraphDatabaseService` API will always work and even writes can survive ZooKeeper failures if a master is present.

ZooKeeper requires a majority of the ZooKeeper instances to be available to operate properly. This means that the number of ZooKeeper instances should always be an odd number since that will make best use of available hardware.

To further clarify the fault tolerance characteristics of Neo4j HA here are a few example setups:

7.2.1. Small

- 3 physical (or virtual) machines
- 1 ZooKeeper instance running on each machine
- 1 Neo4j HA instance running on each machine

This setup is conservative in the use of hardware while being able to handle moderate read load. It can fully operate when at least 2 of the ZooKeeper instances are running. Since the ZooKeeper service and Neo4j HA are running together on each machine this will in most scenarios mean that only one server is allowed to go down.

7.2.2. Medium

- 5-7+ machines
- ZooKeeper running on 3, 5 or 7 machines
- Neo4j HA can run on 5+ machines

This setup may mean that two different machine setups have to be managed (some machines run both ZooKeeper and Neo4j HA). The fault tolerance will depend on how many machines there are that are running ZooKeeper. With 3 ZooKeeper instances the cluster can survive one ZooKeeper going down, with 5 it can survive 2 and with 7 it can handle 3 ZooKeeper instances failing. The number of Neo4j HA instances that can fail for normal operations is theoretically all but 1 (but for each required master election the ZooKeeper service must be available).

7.2.3. Large

- 8+ total machines
- 3+ Neo4j HA machines
- 5+ Zookeeper, on separate dedicated machines

In this setup all ZooKeeper instances are running on separate machines as a dedicated ZooKeeper service. The dedicated ZooKeeper cluster can handle half of the instances, minus 1, going down. The Neo4j HA cluster will be able to operate with at least a single live machine. Adding more Neo4j HA instances is very easy in this setup since Zookeeper is operating as a separate service.

7.2.4. Installation Notes

For installation instructions of a High Availability cluster please visit the Neo4j Wiki [2].

Note that while the `HighlyAvailableGraphDatabase` supports the same API as the `EmbeddedGraphDatabase`, it does have additional configuration parameters.

Table 7.1. HighlyAvailableGraphDatabase configuration parameters

Parameter Name	Value	Example value	Required?
`ha.machine_id`	integer >= 0	`1`	yes
`ha.server`	(auto-discovered) host & port to bind when acting as master	`my-domain.com:6001`	no
`ha.zoo_keeper_servers`	comma delimited zookeeper connections	`localhost:2181, localhost:2182, localhost:2183`	yes
`ha.pull_interval`	interval for polling master from a slave, in seconds	`30`	no

7.3. How Neo4j HA operates

A Neo4j HA cluster operates cooperatively, coordinating activity through Zookeeper.

On startup a Neo4j HA instance will connect to the ZooKeeper service to register itself and ask, "who is master?" If some other machine is master, the new instance will start as slave and connect to that master. If the machine starting up was the first to register — or should become master according to the master election algorithm — it will start as master.

When performing a write transaction on a slave each write operation will be synchronized with the master (locks will be acquired on both master and slave). When the transaction commits it will first occur on the master. If the master commit is successful the transaction will be committed on the slave as well. To ensure consistency, a slave has to be up to date with the master before performing a write operation. This is built into the communication protocol between the slave and master, so that updates will happen automatically if needed.

When performing a write on the master it will execute in the same way as running in normal embedded mode. Currently the master will not push updates to the slave. Instead, slaves can be configured to have a pull interval. Without polling, updates will only happen on slaves whenever they synchronize a write with the master.

Having all writes go through slaves has the benefit that the data will be replicated on two machines. This is recommended to avoid rollbacks in case of a master failure that could potentially happen when the new master is elected.

[2] http://wiki.neo4j.org/content/High_Availability_Cluster

Whenever a machine becomes unavailable the ZooKeeper service will detect that and remove it from the cluster. If the master goes down a new master will automatically be elected. Normally a new master is elected and started within just a few seconds and during this time no writes can take place (the write will throw an exception). A machine that becomes available after being unavailable will automatically reconnect to the cluster. The only time this is not true is when an old master had changes that did not get replicated to any other machine. If the new master is elected and performs changes before the old master recovers, there will two different versions of the data. The old master will not be able to attach itself to the cluster and will require maintenance (replace the wrong version of the data with the one running in the cluster).

All this can be summarized as:

- Slaves can handle write transactions.
- Updates to slaves are eventual consistent.
- Neo4j HA is fault tolerant and (depending on ZooKeeper setup) can continue to operate from X machines down to a single machine.
- Slaves will be automatically synchronized with the master on a write operation.
- If the master fails a new master will be elected automatically.
- Machines will be reconnected automatically to the cluster whenever the issue that caused the outage (network, maintenance) is resolved.
- Transactions are atomic, consistent and durable but eventually propagated out to other slaves.
- If the master goes down any running write transaction will be rolled back and during master election no write can take place.
- Reads are highly available.

Chapter 8. Operations

This chapter describes how to maintain a Neo4j installation. This includes topics such as backing up the database and monitoring the health of the database as well as diagnosing issues.

8.1. Backup

Backups are performed over the network live from a running graph database onto a local copy. There are two types of backup: full and incremental.

A *full backup* copies the entire store files from the source database and is required in order to, at a later time, do an incremental backup.

An *incremental backup* will only retrieve and apply the changes since the most recent full or incremental backup. Because of that incremental backup is much more efficient than doing full backups every time.

Important

Backups can only be performed on databases which have the configuration parameter `enable_online_backup=true` set. That will make the backup service available on the default port (6362). To enable the backup service on a different port use for example `enable_online_backup=port=9999` instead.

8.1.1. Embedded and Server

To perform a backup from a running embedded or server database run:

```
# Performing a full backup
./neo4j-backup -full -from 192.168.1.34 -to /mnt/backup/neo4j-backup

# Performing an incremental backup
./neo4j-backup -incremental -from 192.168.1.34 -to /mnt/backup/neo4j-backup

# Performing an incremental backup where the service is registered on a custom port
./neo4j-backup -incremental -from 192.168.1.34:9999 -to /mnt/backup/neo4j-backup
```

8.1.2. High Availability

To perform a backup on an HA cluster you specify one or more ZooKeeper services managing that cluster.

```
# Performing a full backup from HA cluster, specifying two possible ZooKeeper services
./neo4j-backup -full -from-ha 192.168.1.15:2181,192.168.1.16:2181 -to /mnt/backup/neo4j-backup

# Performing an incremental backup from HA cluster, specifying only one ZooKeeper service
./neo4j-backup -incremental -from-ha 192.168.1.15:2181 -to /mnt/backup/neo4j-backup
```

8.1.3. Restoring Your Data

The Neo4j backups are fully functional databases. To use a backup, all you need to do replace your database folder with the backup.

8.2. Security

Neo4j in itself does not enforce security on the data level. However, there are different aspects that should be considered when using Neo4j in different scenarios.

8.2.1. Securing access to the Neo4j Server

The Neo4j server currently does not enforce security on the REST access layer. This should be taken care of by external means. We strongly recommend to front a running Neo4j Server with a proxy like Apache `mod_proxy` [1]. This provides a number of advatages:

- Control access to the Neo4j server to specific IP addresses, URL patterns and IP ranges. This can be used to make for instance only the `/db/data` namespace accessible to non-local clients, while the `/db/admin` URLs only respond to a specific IP address.

```
<Proxy *>
  Order Deny,Allow
  Deny from all
  Allow from 192.168.0
</Proxy>
```

- Run Neo4j Server as a non-root user on a Linux/Unix system on a port < 1000 (e.g. port 80) using

```
ProxyPass /neo4jdb/data http://localhost:7474/db/data
ProxyPassReverse /neo4jdb/data http://localhost:7474/db/data
```

- Simple load balancing in a clustered environment to load-balance read load using the Apache `mod_proxy_balancer` [2] plugin

```
<Proxy balancer://mycluster>
BalancerMember http://192.168.1.50:80
BalancerMember http://192.168.1.51:80
</Proxy>
ProxyPass /test balancer://mycluster
```

8.3. Monitoring

Note

Most of the monitoring features are only available in the Advanced and Enterprise editions of Neo4j.

In order to be able to continuously get an overview of the health of a Neo4j database, there are different levels of monitoring facilities available.

8.3.1. JMX

How to connect to a Neo4j instance using JMX and JConsole

First, start your embedded database or the Neo4j Server, for instance using

```
$NEO4J_SERVER_HOME/bin/neo4j start
```

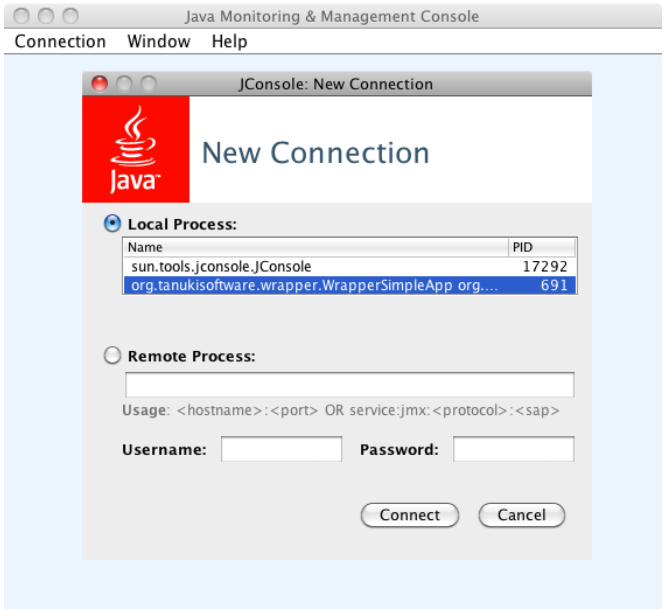
[1] http://httpd.apache.org/docs/2.2/mod/mod_proxy.html

Now, start JConsole with

```
$JAVA_HOME/bin/jconsole
```

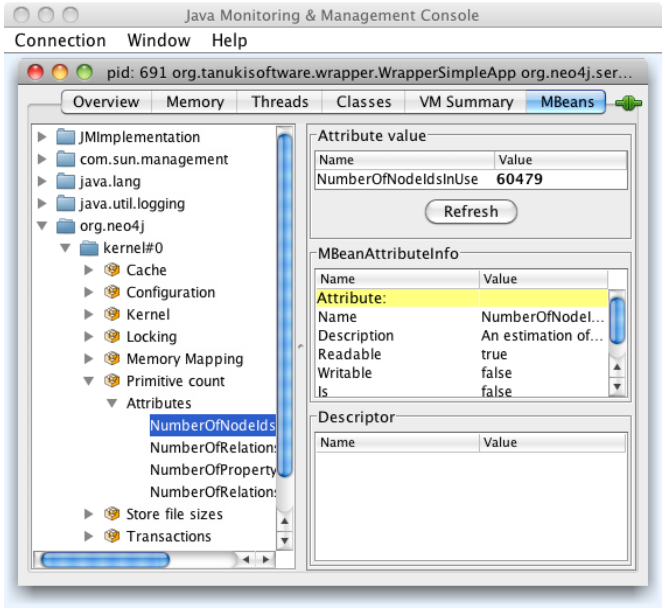
Connect to the process running your Neo4j database instance:

Figure 8.1. Connecting JConsole to the Neo4j Java process



Now, beside the MBeans exposed by the JVM, you will see an `org.neo4j` section in the MBeans tab. Under that, you will have access to all the monitoring information exposed by Neo4j.

Figure 8.2. Neo4j MBeans View



How to connect to the JMX monitoring programmatically

In order to programmatically connect to the Neo4j JMX server, there are some convenience methods in the Neo4j Management component to help you find out the most commonly used monitoring attributes of Neo4j. For instance, the number of node IDs in use can be obtained with code like:

```
Neo4jManager manager = new Neo4jManager( graphDb.getManagementBean( Kernel.class ) );
long nodeIDsInUse    = manager.getPrimitivesBean.getNumberOfNodeIdsInUse();
```

Once you have access to this information, you can use it to for instance expose the values to SNMP [http://en.wikipedia.org/wiki/Simple_Network_Management_Protocol] or other monitoring systems.

Reference of supported JMX MBeans

Table 8.1. MBeans exposed by the Neo4j Kernel

Name	Description
org.neo4j:instance=kernel#0,name=Memory Mapping	The status of Neo4j memory mapping
org.neo4j:instance=kernel#0,name=Locking	Information about the Neo4j lock status
org.neo4j:instance=kernel#0,name=Transactions	Information about the Neo4j transaction manager
org.neo4j:instance=kernel#0,name=Cache	Information about the caching in Neo4j
org.neo4j:instance=kernel#0,name=Configuration	The configuration parameters used to configure Neo4j
org.neo4j:instance=kernel#0,name=Primitive count	Estimates of the numbers of different kinds of Neo4j primitives
org.neo4j:instance=kernel#0,name=XA Resources	Information about the XA transaction manager
org.neo4j:instance=kernel#0,name=Store file sizes	Information about the sizes of the different parts of the Neo4j graph store
org.neo4j:instance=kernel#0,name=Kernel	Information about the Neo4j kernel
org.neo4j:instance=kernel#0,name=High Availability	Information an High Availability cluster, if enabled.

Table 8.2. MBean Memory Mapping

Attribute	Description	Type
MemoryPools	Get information about each pool of memory mapped regions from store files with memory mapping enabled	String

Table 8.3. MBean Locking

Attribute	Description	Type
NumberOfAdvertedDeadlocks	The number of lock sequences that would have lead to a deadlock situation that Neo4j has detected and adverted (by throwing DeadlockDetectedException).	Integer

Table 8.4. MBean Transactions

Attribute	Description	Type
NumberOfOpenTransactions	The number of currently open transactions	Integer
PeakNumberOfConcurrentTransactions	The highest number of transactions ever opened concurrently	Integer
NumberOfOpenedTransactions	The total number started transactions	Integer
NumberOfCommittedTransactions	The total number of committed transactionss	Integer

Table 8.5. MBean Cache

Attribute	Description	Type
CacheType	The type of cache used by Neo4j	String
NodeCacheSize	The number of Nodes currently in cache	Integer
RelationshipCacheSize	The number of Relationships currently in cache	Integer
clear()	clear all caches	function, void

Table 8.6. MBean Configuration

Attribute	Description	Type
`store_dir`	Relative path for where the Neo4j storage directory is located	String
`rebuild_idgenerators_fast`	Use a quick approach for rebuilding the ID generators. This give quicker recovery time, but will limit the ability to reuse the space of deleted entities.	String
`logical_log`	Relative path for where the Neo4j logical log is located	String
`neostore.propertystore.db.index.keys.mapped_memory`	The size to allocate for memory mapping the store for property key strings	String
`neostore.propertystore.db.strings.mapped_memory`	The size to allocate for memory mapping the string property store	String
`neostore.propertystore.db.arrays.mapped_memory`	The size to allocate for memory mapping the array property store	String
`neo_store`	Relative path for where the Neo4j storage information file is located	String
`neostore.relationshipstore.db.mapped_memory`	The size to allocate for memory mapping the relationship store	String
`neostore.propertystore.db.index.mapped_memory`	The size to allocate for memory mapping the store for property key indexes	String
`create`	Configuration attribute	String

Attribute	Description	Type
`enable_remote_shell`	Enable a remote shell server which shell clients can log in to	String
`neostore.propertystore.db.mapped_memory`	The size to allocate for memory mapping the property value store	Integer
`neostore.nodestore.db.mapped_memory`	The size to allocate for memory mapping the node store	String
`dir`	Configuration attribute	String

Table 8.7. MBean Primitive count

Attribute	Description	Type
NumberOfNodeIdsInUse	An estimation of the number of nodes used in this Neo4j instance	Integer
NumberOfRelationshipIdsInUse	An estimation of the number of relationships used in this Neo4j instance	Integer
NumberOfPropertyIdsInUse	An estimation of the number of properties used in this Neo4j instance	Integer
NumberOfRelationshipTypeIdsInUse	The number of relationship types used in this Neo4j instance	Integer

Table 8.8. MBean XA Resources

Attribute	Description	Type
XaResources	Information about all XA resources managed by the transaction manager	String

Table 8.9. MBean Store file sizes

Attribute	Description	Type
TotalStoreSize	The total disk space used by this Neo4j instance, in bytes.	Integer
LogicalLogSize	The amount of disk space used by the current Neo4j logical log, in bytes.	Integer
ArrayStoreSize	The amount of disk space used to store array properties, in bytes.	Integer
NodeStoreSize	The amount of disk space used to store nodes, in bytes.	Integer
PropertyStoreSize	The amount of disk space used to store properties (excluding string values and array values), in bytes.	Integer
RelationshipStoreSize	The amount of disk space used to store relationships, in bytes.	Integer

Attribute	Description	Type
StringStoreSize	The amount of disk space used to store string properties, in bytes.	Integer

Table 8.10. MBean Kernel

Attribute	Description	Type
ReadOnly	Whether this is a read only instance.	boolean
MBeanQuery	An ObjectName that can be used as a query for getting all management beans for this Neo4j instance.	String
KernelStartTime	The time from which this Neo4j instance was in operational mode	Date
StoreCreationDate	The time when this Neo4j graph store was created	Date
StoreId	An identifier that uniquely identifies this Neo4j graph store	String
StoreLogVersion	The current version of the Neo4j store logical log	String
KernelVersion	The version of Neo4j	String
StoreDirectory	The location where the Neo4j store is located	String

Table 8.11. MBean High Availability

Attribute	Description	Type
MachineId	The cluster machine id of this instance	String
Master	True, if this Neo4j instance is currently Master in the cluster	boolean
ConnectedSlaves	A list of conencted slaves in this cluster	String
InstancesInCluster	Information about the other Neo4j instances in this HA cluster	String

Part II. Tools

Chapter 9. Web Administration

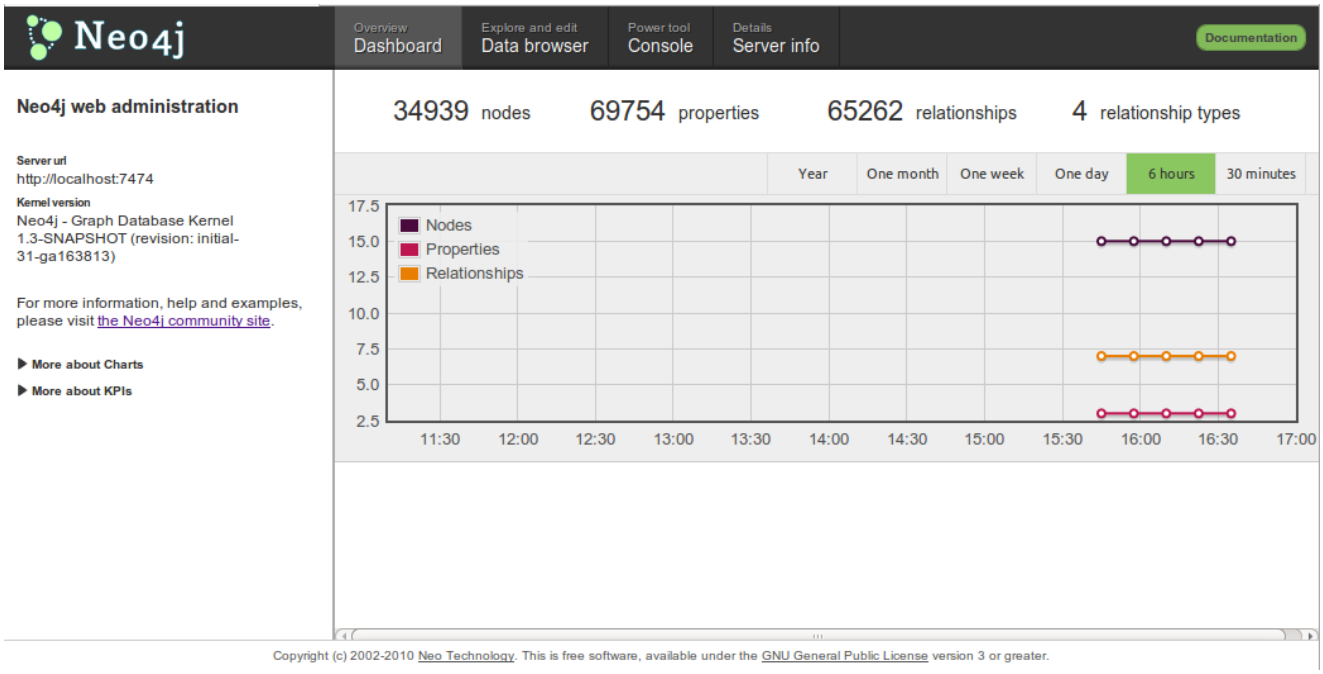
The Neo4j Web Administration is the primary user interface for Neo4j. With it, you can:

- monitor the Neo4j Server
- manipulate and browse data
- interact with the database via a scripting environment
- view raw data management objects (JMX MBeans)

9.1. Dashboard tab

The Dashboard tab provides an overview of a running Neo4j instance.

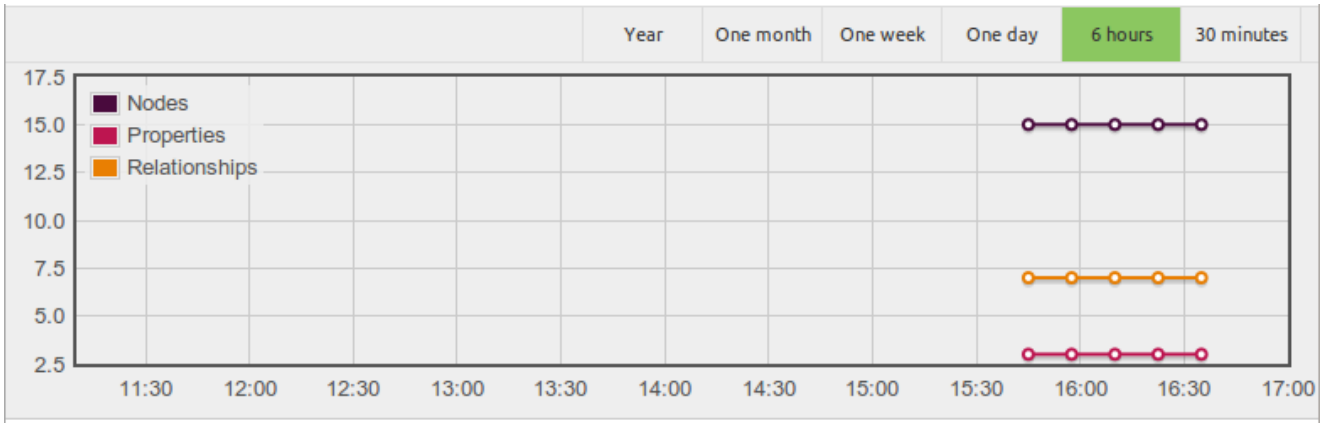
Figure 9.1. Web Administration Dashboard



9.1.1. Entity chart

The charts show entity counts over time: node, relationship and properties.

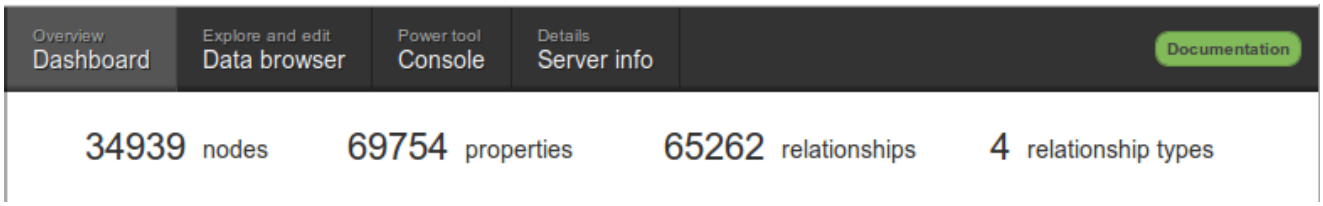
Figure 9.2. Entity charting



9.1.2. Status monitoring

Below the entity chart is a collection of status panels, displaying current resource usage.

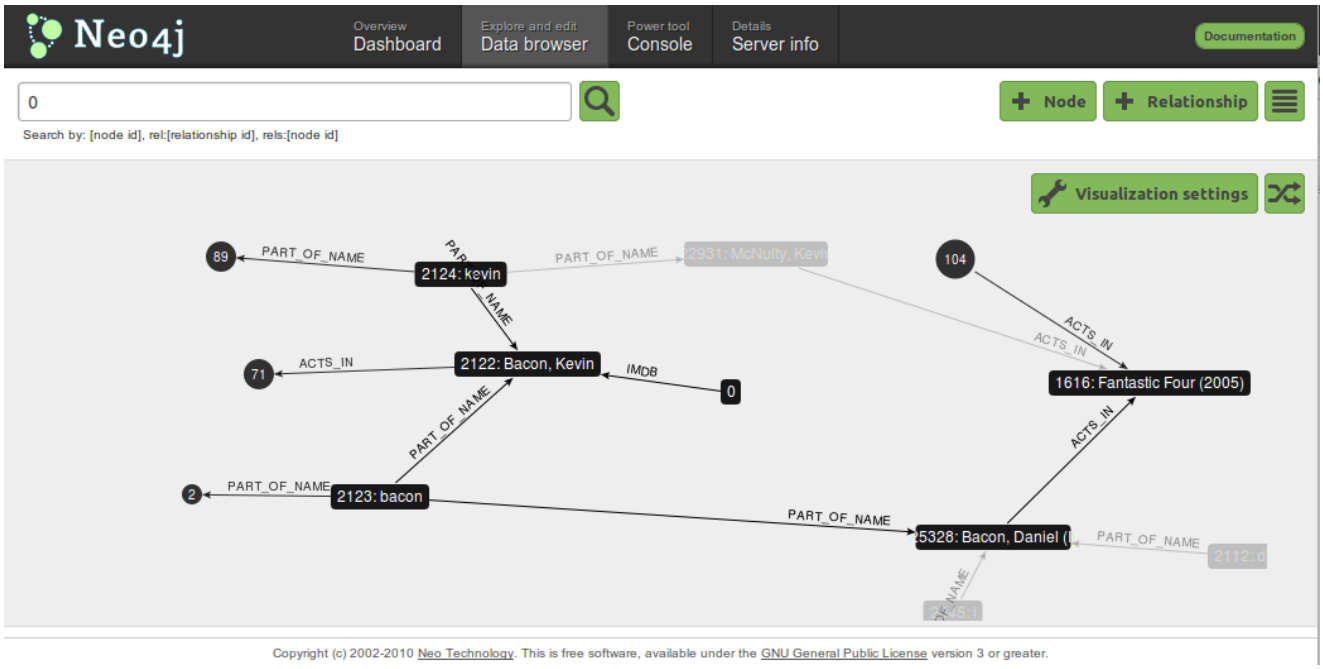
Figure 9.3. Status indicator panels



9.2. Data tab

Use the Data tab to browse, add or modify nodes, relationships and their properties.

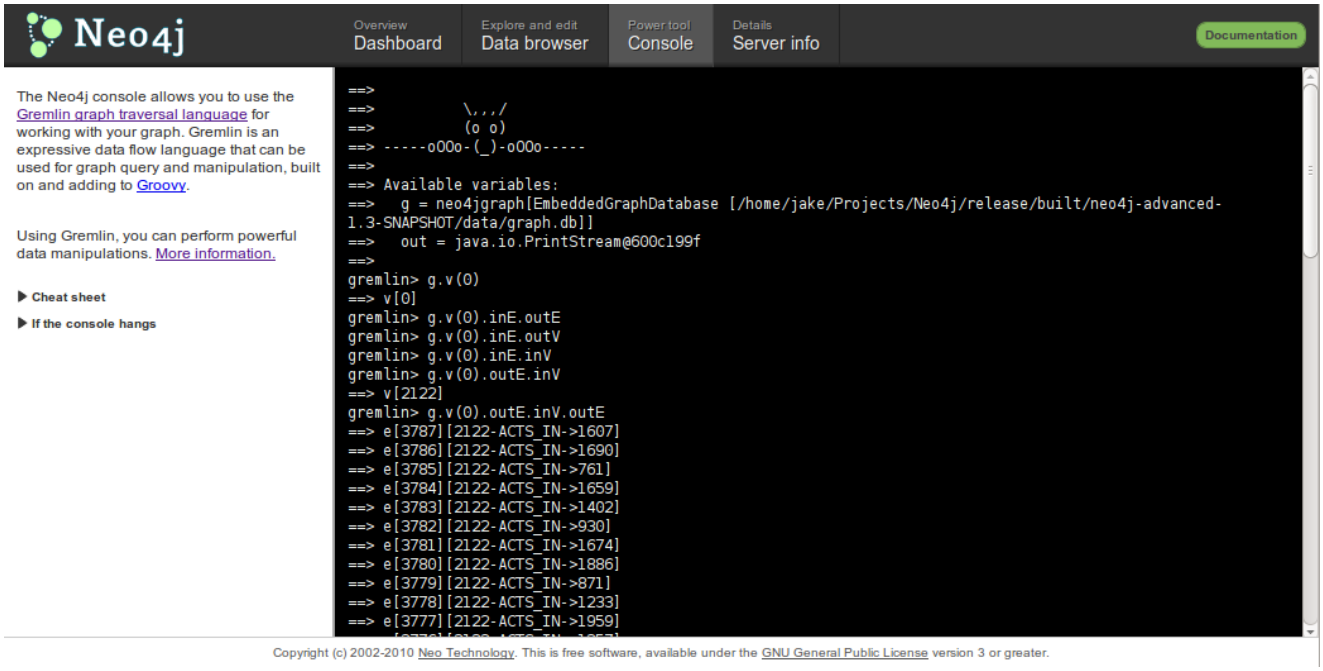
Figure 9.4. Browsing and manipulating data



9.3. Console tab

The Console tab gives scripting access to the database via the Gremlin [http://gremlin.tinkerpop.com] scripting engine.

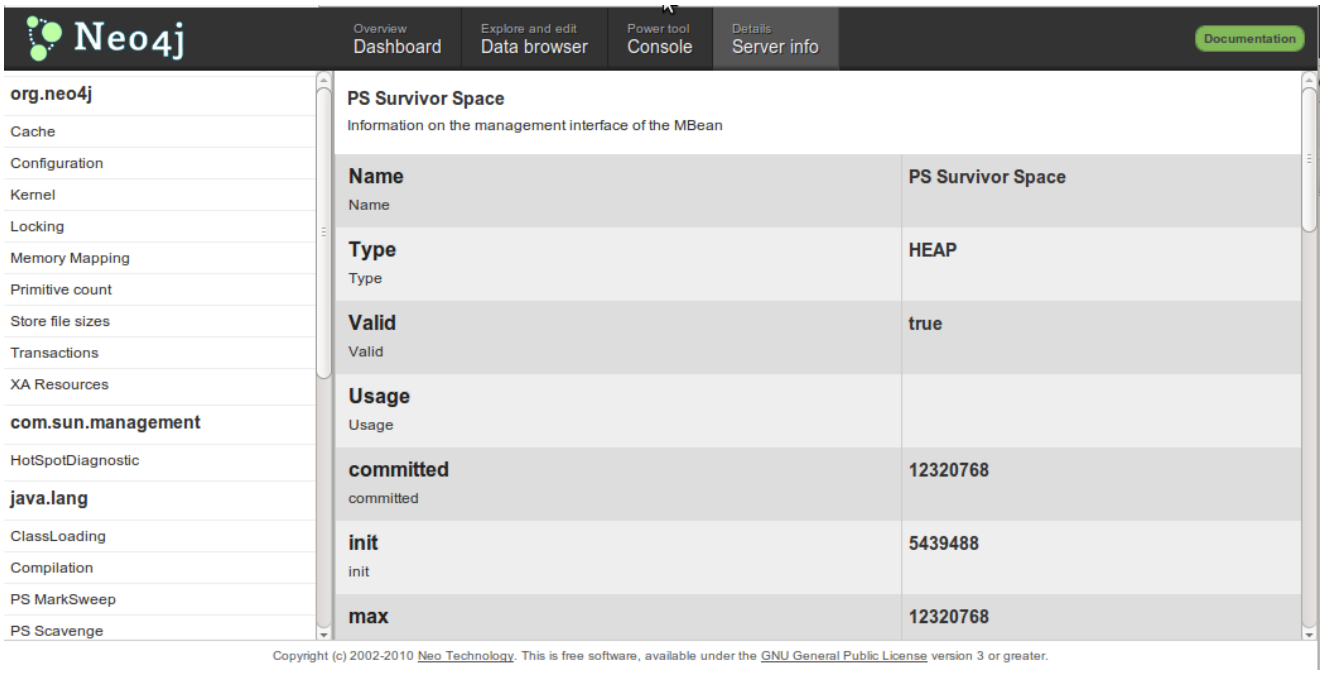
Figure 9.5. Manipulating data with Gremlin



9.4. The JMX tab

The JMX tab provides raw access to all available management objects.

Figure 9.6. JMX Attributes



Chapter 10. Neo4j Shell

Neo4j shell is a command-line shell for browsing the graph, much like how the Unix shell along with commands like `cd`, `ls` and `pwd` can be used to browse your local file system. It consists of two parts:

- a lightweight client that sends commands via RMI and
- a server that processes those commands and sends the result back to the client.

It's a nice tool for development and debugging. This guide will show you how to get it going!

10.1. Starting the shell

When used together with Neo4j started as a server, simply issue the following at the command line:

```
./bin/neo4j-shell
```

For the full list of options, see the reference in the Shell manual page.

To connect to a running Neo4j database, use Section 10.1.4, "Read-only mode" for local databases and see Section 10.1.1, "Enabling the shell server" for remote databases.

You need to make sure that the shell jar file is on the classpath when you start up your Neo4j instance.

10.1.1. Enabling the shell server

Shell is enabled from the configuration of the Neo4j kernel, see Section 4.2, "Server Configuration". Here's some sample configurations:

```
# Using default values
enable_remote_shell = true
# ...or specify custom port, use default values for the others
enable_remote_shell = port=1234
```

When using the Neo4j server, see Section 4.2, "Server Configuration" for how to add configuration settings in that case.

There are two ways to start the shell, either by connecting to a remote shell server or by pointing it to a Neo4j store path.

10.1.2. Connecting to a shell server

To start the shell and connect to a running server, run:

```
neo4j-shell
```

Alternatively supply `-port` and `-name` options depending on how the remote shell server was enabled. Then you'll get the shell prompt like this:

```
neo4j-sh (0)$
```

10.1.3. Pointing the shell to a path

To start the shell by just pointing it to a Neo4j store path you run the shell jar file. Given that the right neo4j-kernel-<version>.jar and jta jar files are in the same path as your neo4j-shell-<version>.jar file you run it with:

```
$ neo4j-shell -path path/to/neo4j-db
```

10.1.4. Read-only mode

By issuing the `-readonly` switch when starting the shell with a store path, changes cannot be made to the database during the session.

```
$ neo4j-shell -readonly -path path/to/neo4j-db
```

10.1.5. Run a command and then exit

It is possible to tell the shell to just start, execute a command and then exit. This opens up for uses of background jobs and also handling of huge output of f.ex. an *"ls*" command where you then could pipe the output to *"less*" or another reader of your choice, or even to a file. So some examples of usage:

```
$ neo4j-shell -c "cd -a 24 && set name Mattias"
$ neo4j-shell -c "trav -r KNOWS" | less
```

10.2. Passing options and arguments

Passing options and arguments to your commands is very similar to many CLI commands in an *nix environment. Options are prefixed with a `-` and can contain one or more options. Some options expect a value to be associated with it. Arguments are string values which aren't prefixed with `-`. Let's look at `ls` as an example:

`ls -r -f KNOWS:out -v 12345` will make a verbose listing of node `12345`'s outgoing relationships of type `KNOWS`. The node id, `12345`, is an argument to `ls` which tells it to do the listing on that node instead of the current node (see `pwd` command). However a shorter version of this can be written:

`ls -rfv KNOWS:out 12345`. Here all three options are written together after a single `-` prefix. Even though `f` is in the middle it gets associated with the `KNOWS:out` value. The reason for this is that the `ls` command doesn't expect any values associated with the `r` or `v` options. So, it can infer the right values for the rights options.

10.3. Enum options

Some options expects a value which is one of the values in an enum, f.ex. direction part of relationship type filtering where there's `INCOMING`, `OUTGOING` and `BOTH`. All such values can be supplied in an easier way. It's enough that you write the start of the value and the interpreter will find what you really meant. F.ex. `out`, `in`, `i` or even `INCOMING`.

10.4. Filters

Some commands makes use of filters for varying purposes. F.ex. `-f` in `ls` and in `trav`. A filter is supplied as a json [http://www.json.org/] object (w/ or w/o the surrounding `{}` brackets. Both keys

and values can contain regular expressions for a more flexible matching. An example of a filter could be `.*url.*:http.*neo4j.*,name:Neo4j`. The filter option is also accompanied by the options `-i` and `-l` which stands for `ignore case` (ignore casing of the characters) and `loose matching` (it's considered a match even if the filter value just matches a part of the compared value, not necessarily the entire value). So for a case-insensitive, loose filter you can supply a filter with `-f -i -l` or `-fil` for short.

10.5. Node titles

To make it easier to navigate your graph the shell can display a title for each node, f.ex. in `ls -r`. It will display the relationships as well as the nodes on the other side of the relationships. The title is displayed together with each node and its best suited property value from a list of property keys.

If you're standing on a node which has two `KNOWS` relationships to other nodes it'd be difficult to know which friend is which. The title feature addresses this by reading a list of property keys and grabbing the first existing property value of those keys and displays it as a title for the node. So you may specify a list (with or without regular expressions), f.ex: `name,title.*,caption` and the title for each node will be the property value of the first existing key in that list. The list is defined by the client (you) using the `TITLE_KEYS` environment variable and the default being `.*name.*,.*title.*`

10.6. How to use (individual commands)

The shell is modeled after Unix shells like bash that you use to walk around your local file system. It has some of the same commands, like `cd` and `ls`. When you first start the shell (see instructions above), you will get a list of all the available commands. Use `man <command>` to get more info about a particular command. Some notes:

10.6.1. Current node/relationship and path

You have a current node/relationship and a "current path" (like a current working directory in bash) that you've traversed so far. You start at the reference node [http://api.neo4j.org/current/org/neo4j/graphdb/GraphDatabaseService.html#getReferenceNode()] and can then `cd` your way through the graph (check your current path at any time with the `pwd` command). `cd` can be used in different ways:

- `cd <node-id>` will traverse one relationship to the supplied node id. The node must have a direct relationship to the current node.
- `cd -a <node-id>` will do an absolute path change, which means the supplied node doesn't have to have a direct relationship to the current node.
- `cd -r <relationship-id>` will traverse to a relationship instead of a node. The relationship must have the current node as either start or end point. To see the relationship ids use the `ls -vr` command on nodes.
- `cd -ar <relationship-id>` will do an absolute path change which means the relationship can be any relationship in the graph.
- `cd` will take you back to the reference node, where you started in the first place.
- `cd ..` will traverse back one step to the previous location, removing the last path item from your current path (`pwd`).

- `cd start` *(only if your current location is a relationship)*. Traverses to the start node of the relationship.
- `cd end` *(only if your current location is a relationship)*. Traverses to the end node of the relationship.

10.6.2. Listing the contents of a node/relationship

List contents of the current node/relationship (or any other node) with the `ls` command. Please note that it will give an empty output if the current node/relationship has no properties or relationships (for example in the case of a brand new graph). `ls` can take a node id as argument as well as filters, see Section 10.4, “Filters” and for information about how to specify direction see Section 10.3, “Enum options”. Use `man ls` for more info.

10.6.3. Creating nodes and relationships

You create new nodes by connecting them with relationships to the current node. For example, `mkrel -t A_RELATIONSHIP_TYPE -d OUTGOING -c` will create a new node (`-c`) and draw to it an `OUTGOING` relationship of type `A_RELATIONSHIP_TYPE` from the current node. If you already have two nodes which you’d like to draw a relationship between (without creating a new node) you can do for example, `mkrel -t A_RELATIONSHIP_TYPE -d OUTGOING -n <other-node-id>` and it will just create a new relationship between the current node and that other node.

10.6.4. Setting, renaming and removing properties

Property operations are done with the `set`, `mv` and `rm` commands. These commands operates on the current node/relationship. * `set <key> <value>` with optionally the `-t` option (for value type) sets a property. Supports every type of value that Neo4j supports. Examples of a property of type `int`:

```
$ set -t int age 29
```

And an example of setting a `double[]` property:

```
$ set -t double[] my_values [1.4,12.2,13]
```

- `rm <key>` removes a property.
- `mv <key> <new-key>` renames a property from one key to another.

10.6.5. Deleting nodes and relationships

Deleting nodes and relationships is done with the `rmrel` command. It focuses on deletion of relationships, but a node can also be deleted if the deleted relationship leaves the opposite node "stranded" (i.e. it no longer has any relationships drawn to it) *"and*" the `-d` options is supplied. See the relationship ids with the `ls -rv` command.

10.6.6. Environment variables

The shell uses environment variables a-la bash to keep session information, such as the current path and more. The commands for this mimics the bash commands `export` and `env`. For example you can at anytime issue a `export STACKTRACES=true` command to set the `STACKTRACES` environment

variable to `true`. This will then result in stacktraces being printed if an exception or error should occur. List environment variables using `env`

10.6.7. Executing groovy/python scripts

The shell has support for executing scripts, such as Groovy [http://groovy.codehaus.org] and Python [http://www.python.org] (via Jython [http://www.jython.org]). As of now the scripts (*.groovy, *.py) must exist on the server side and gets called from a client with for example, `gsh --renamePerson 1234 "Mathias" "Mattias" --doSomethingElse` where the scripts renamePerson.groovy and doSomethingElse.groovy must exist on the server side in any of the paths given by the `GSH_PATH` environment variable (defaults to .:src:src/script). This variable is like the java classpath, separated by a `:`. The python/jython scripts can be executed with the `jsh` in a similar fashion, however the scripts have the .py extension and the environment variable for the paths is `JSH_PATH`.

When writing the scripts assume that there's made available an `args` variable (a String[]) which contains the supplied arguments. In the case of the `renamePerson` example above the array would contain `["1234", "Mathias", "Mattias"]`. Also please write your outputs to the `out` variable, such as `out.println( "My tracing text" )` so that it will be printed at the shell client instead of the server.

10.6.8. Traverse

You can traverse the graph with the `trav` command which allows for simple traversing from the current node. You can supply which relationship types (w/ regex matching) and optionally direction as well as property filters for matching nodes. In addition to that you can supply a command line to execute for each match. An example: `trav -o depth -r KNOWS:both,HAS_.*:incoming -c "ls $n"`. Which means traverse depth first for relationships with type `KNOWS` disregarding direction and incoming relationships with type matching `HAS_.\*` and do a `ls <matching node>` for each match. The node filtering is supplied with the `-f` option, see Section 10.4, "Filters". See Section 10.3, "Enum options" for the traversal order option. Even relationship types/directions are supplied using the same format as filters.

10.6.9. Indexing

It's possible to query and manipulate indexes via the index command. Example: `index -i persons name` (will index the name for the current node or relationship in the "persons" index).

- `-g` will do exact lookup in the index and display hits. You can supply -c with a command to be executed for each hit.
- `-q` will ask the index a query and display hits. You can supply -c with a command to be executed for each hit.
- `--cd` will change current location to the hit from the query. It's just a convenience for using the -c option.
- `--ls` will do a listing of the contents for each hit. It's just a convenience for using the -c option.
- `-i` will index a key-value pair in an index for the current node/relationship. If no value is given the property value for that key for the current node is used as value.

- `-r` will remove a key-value pair (if it exists) from an index for the current node/relationship. If no value is given the property value for that key for the current node is used as value.

10.7. Extending the shell: Adding your own commands

Of course the shell is extendable and has a generic core which has nothing to do with Neo4j… only some of the commands [http://components.neo4j.org/neo4j-shell/1.3/apidocs/org/neo4j/shell/App.html] do.

So you say you'd like to start a Neo4j graph database [http://api.neo4j.org/current/org/neo4j/graphdb/GraphDatabaseService.html], enable the remote shell and add your own apps to it so that your apps and the standard Neo4j apps co-exist side by side? Well, here's an example of how an app could look like:

```
public class LsRelTypes extends GraphDatabaseApp
{
    @Override
    protected String exec( AppCommandParser parser, Session session, Output out )
            throws ShellException, RemoteException
    {
        GraphDatabaseService graphDb = getServer().getDb();
        out.println( "Types:" );
        for ( RelationshipType type : graphDb.getRelationshipTypes() )
        {
            out.println( type.name() );
        }
        return null;
    }
}
```

You make your app discoverable via the Java Service API, so in a file e.g. src/main/resources/META-INF/services/org.neo4j.shell.App include: `org.my.domain.MyShellApp`

And you could now use it in the shell by typing `lsreltypes` (its name is based on the class name).

If you'd like it to display some nice help information when using the `help` (or `man`) app, override the `getDescription` method for a general description and use `addValueType` method to add descriptions about (and logic to) the options you can supply when using your app.

Know that the apps reside server-side so if you have a running server and starts a remote client to it from another JVM you can't add your apps on the client.

Part III. Troubleshooting

Chapter 11. Troubleshooting guide

Problem	Cause	Resolution
OutOfMemoryError	Too large top level transactions or leaking transactions not finished properly.	Split updates into smaller transactions. Always make sure transactions are finished properly.
ResourceAcquisitionFailedException or an error message containing the text “The transaction is marked for rollback only”	Leaked non finished transaction tied to the current thread in state marked for rollback only.	Finish transactions properly.
DeadlockDetectedException	Concurrent updates of contended resources or not finishing transactions properly.	See Section 3.4, “Deadlocks”.

Chapter 12. Community support

Get help from the Neo4j open source community, here are some starting points:

- Searchable user mailing list archive [http://www.mail-archive.com/user@lists.neo4j.org/info.html].
- User mailing list [https://lists.neo4j.org/mailman/listinfo/user].
- Neo4j wiki [http://wiki.neo4j.org/]
- IRC channel: irc://irc.freenode.net/neo4j

Appendix A. Manpages

The Neo4j Unix manual pages are included on the following pages.

Name

neo4j — Neo4j Server control and management

Synopsis

neo4j <command>

DESCRIPTION

Neo4j is a graph database, perfect for working with highly connected data.

COMMANDS

console
Start the server as an application, running as a foreground proces. Stop the server using `CTRL-C`.

start
Start server as daemon, running as a background process.

stop
Stops a running daemonized server.

restart
Restarts a running server.

condrestart
Restarts a server, but only if it was already running.

status
Current running state of the server

install
Installs the server as a platform-appropriate system service.

remove
Uninstalls the system service

dump
Displays thread dump, also saved to the wrapper.log

Usage - Windows

Neo4j.bat

Double-clicking on the Neo4j.bat script will start the server in a console. To quit, just press `control-C` in the console window.

InstallNeo4j/UninstallNeo4j

Neo4j can be installed as a Windows Service, running without a console window. You'll need to run the scripts with Administrator priveleges. Just use either of these bat scripts:

- InstallNeo4j - install as a Windows service
  - will install and automatically start the service
  - use the normal windows administrative controls for start/stop
- UninstallNeo4j - remove the Neo4j service

FILES

conf/neo4j-server.properties
Server configuration.

conf/neo4j-wrapper.conf
Configuration for service wrapper.

conf/neo4j.properties
Tuning configuration for the database.

Name

neo4j-shell — a command-line tool for exploring and manipulating a graph database

Synopsis

neo4j-shell [*REMOTE OPTIONS*]

neo4j-shell [*LOCAL OPTIONS*]

DESCRIPTION

Neo4j shell is a command-line shell for browsing the graph, much like how the Unix shell along with commands like `cd`, `ls` and `pwd` can be used to browse your local file system. The shell can connect directly to a graph database on the file system. To access local a local database used by other processes, use the readonly mode.

REMOTE OPTIONS

-port *PORT*
Port of host to connect to (default: 1337).

-host *HOST*
Domain name or IP of host to connect to (default: localhost).

-name *NAME*
RMI name, i.e. rmi://<host>:<port>/<name> (default: shell).

-readonly
Access the database in read-only mode. The read-only mode enables browsing a database that is used by other processes.

LOCAL OPTIONS

-path *PATH*
The path to the database directory. If there is no database at the location, a new one will e created.

-pid *PID*
Process ID to connect to.

-readonly
Access the database in read-only mode. The read-only mode enables browsing a database that is used by other processes.

-c *COMMAND*
Command line to execute. After executing it the shell exits.

-config *CONFIG*
The path to the Neo4j configuration file to be used.

EXAMPLES

Examples for remote:

```
neo4j-shell
neo4j-shell -port 1337
neo4j-shell -host 192.168.1.234 -port 1337 -name shell
neo4j-shell -host localhost -readonly
```

Examples for local:

```
neo4j-shell -path /path/to/db
neo4j-shell -path /path/to/db -config /path/to/neo4j.config
neo4j-shell -path /path/to/db -readonly
```

Name

neo4j-coordinator — Neo4j Coordinator for High-Availability clusters

Synopsis

neo4j-coordinator <command>

DESCRIPTION

Neo4j Coordinator is a server which provides coordination for a Neo4j High Availability Data cluster. A "coordination cluster" must be started and available before the "data cluster" can be started. This server is a member of the cluster.

COMMANDS

console
Start the server as an application, running as a foreground proces. Stop the server using `CTRL-C`.

start
Start server as daemon, running as a background process.

stop
Stops a running daemonized server.

restart
Restarts a running server.

condrestart
Restarts a server, but only if it was already running.

status
Current running state of the server

install
Installs the server as a platform-appropriate system service.

remove
Uninstalls the system service

dump
Displays thread dump, also saved to the wrapper.log

FILES

conf/coord.cfg
Coordination server configuration.

conf/coord-wrapper.cfg
Configuration for service wrapper.

data/coordinator/myid
Unique identifier for coordinator instance.

Name

neo4j-coordinator-shell — Neo4j Coordinator Shell interactive interface

Synopsis

neo4j-coordinator-shell -server <host:port> [<cmd> <args>]

DESCRIPTION

Neo4j Coordinator Shell provides an interactive text-based interface to a running Neo4j Coordinator server.

OPTIONS

-server *HOST:PORT*
: Connects to a Neo4j Coordinator at the specified host and port.