Neo4j

the graph database

The Neo4j Manual v1.5-SNAPSHOT

The Neo4j Team *neo4j.org* **<http://neo4j.org/>**
neotechnology.com **<http://neotechnology.com/>**

The Neo4j Manual v1.5-SNAPSHOT

by The Neo4j Team *neo4j.org* <http://neo4j.org/> *neotechnology.com* <http://neotechnology.com/>

Publication date 2011-11-10 10:56:48

Table of Contents

List of Figures

List of Tables

Preface

This is the reference manual for Neo4j version 1.5-SNAPSHOT, written by the Neo4j Team.

The main parts of the manual are:

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.

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.

The included code examples are executed when Neo4j is built and tested. Also, the REST API request and response examples are captured from real interaction with a Neo4j server. Thus, the examples are always in sync with Neo4j.

Who should read this?

The topics should be relevant to architects, administrators, developers and operations personnel.

Part I. Introduction

This part gives a bird's eye view of what a graph database is, and then outlines some specifics of Neo4j.

Chapter 1. Neo4j Highlights

As a robust, scalable and high-performance database, Neo4j is suitable for full enterprise deployment or a subset of the full server can be used in lightweight projects.

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 13, *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 19, *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 traversal steps per second. A traversal step resembles a *join* in a RDBMS.

Chapter 2. Graph Database Concepts

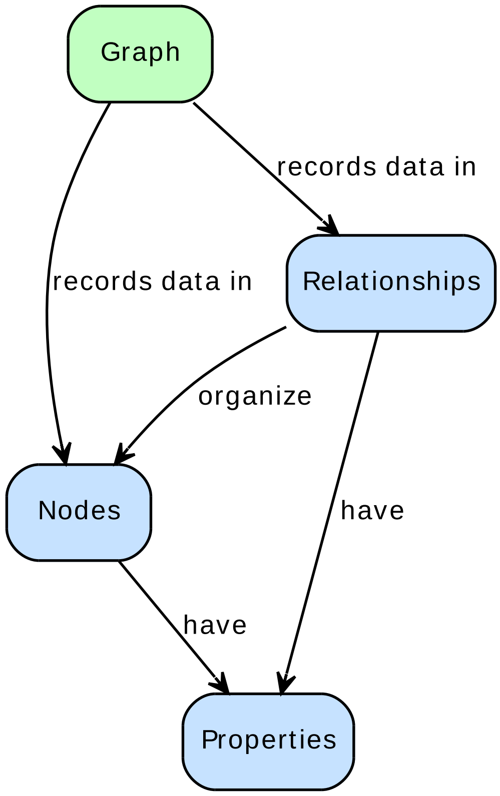
2.1. What is a Graph Database?

A graph database stores data in a graph, the most generic of data structures, capable of elegantly representing any kind of data in a highly accessible way. Let’s follow along some graphs, using them to express graph concepts. We’ll “read” a graph by following arrows around the diagram to form sentences.

2.1.1. A Graph contains Nodes and Relationships

> “A Graph —records data in→ Nodes —which have→ Properties”

The simplest possible graph is a single Node, a record that has named values referred to as Properties. A Node could start with a single Property and grow to a few million, though that can get a little awkward. At some point it makes sense to distribute the data into multiple nodes, organized with explicit Relationships.



2.1.2. Relationships organize the Graph

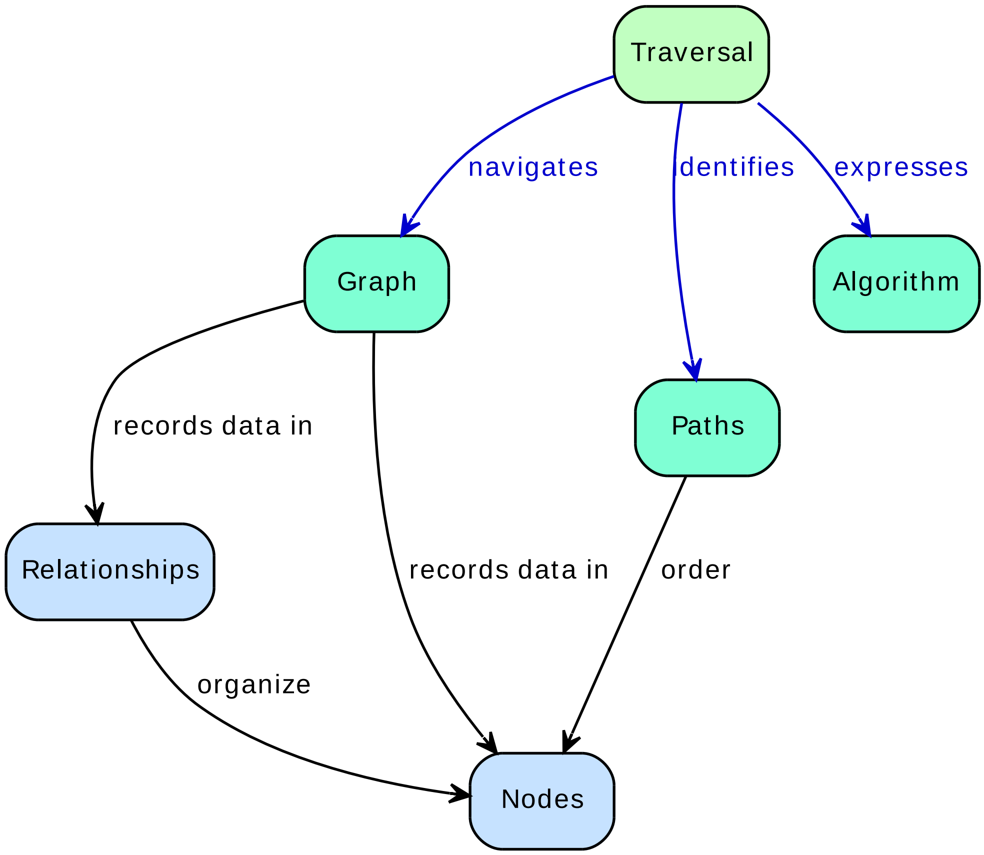
> “Nodes —are organized by→ Relationships —which also have→ Properties”

Relationships organize Nodes into arbitrary structures, allowing a Graph to resemble a List, a Tree, a Map, or a compound Entity – any of which can be combined into yet more complex, richly interconnected structures.

2.1.3. Query a Graph with a Traversal

> “A Traversal —navigates→ a Graph; it —identifies→ Paths —which order→ Nodes”

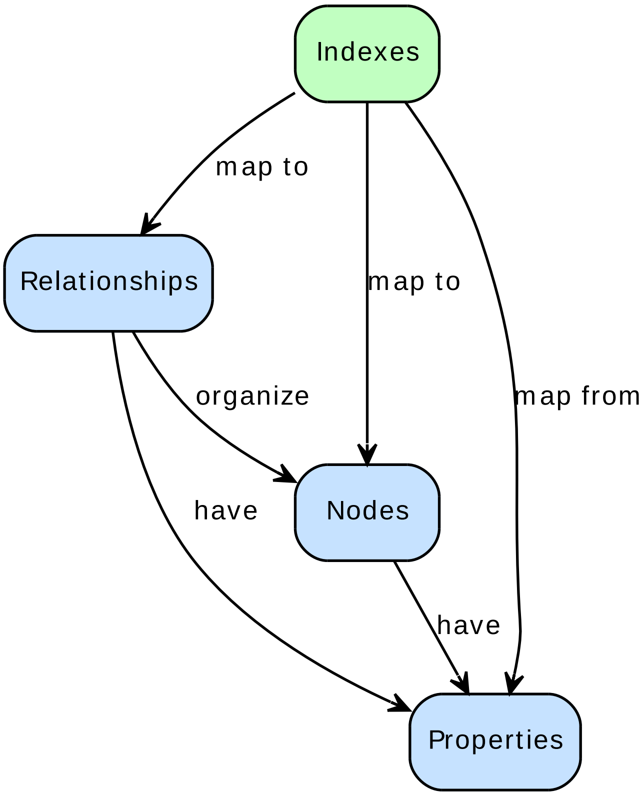
A Traversal is how you query a Graph, navigating from starting Nodes to related Nodes according to an algorithm, finding answers to questions like “what music do my friends like that I don’t yet own,” or “if this power supply goes down, what web services are affected?”



2.1.4. Indexes look-up Nodes or Relationships

"An Index —maps from→ Properties —to either→ Nodes or Relationships"

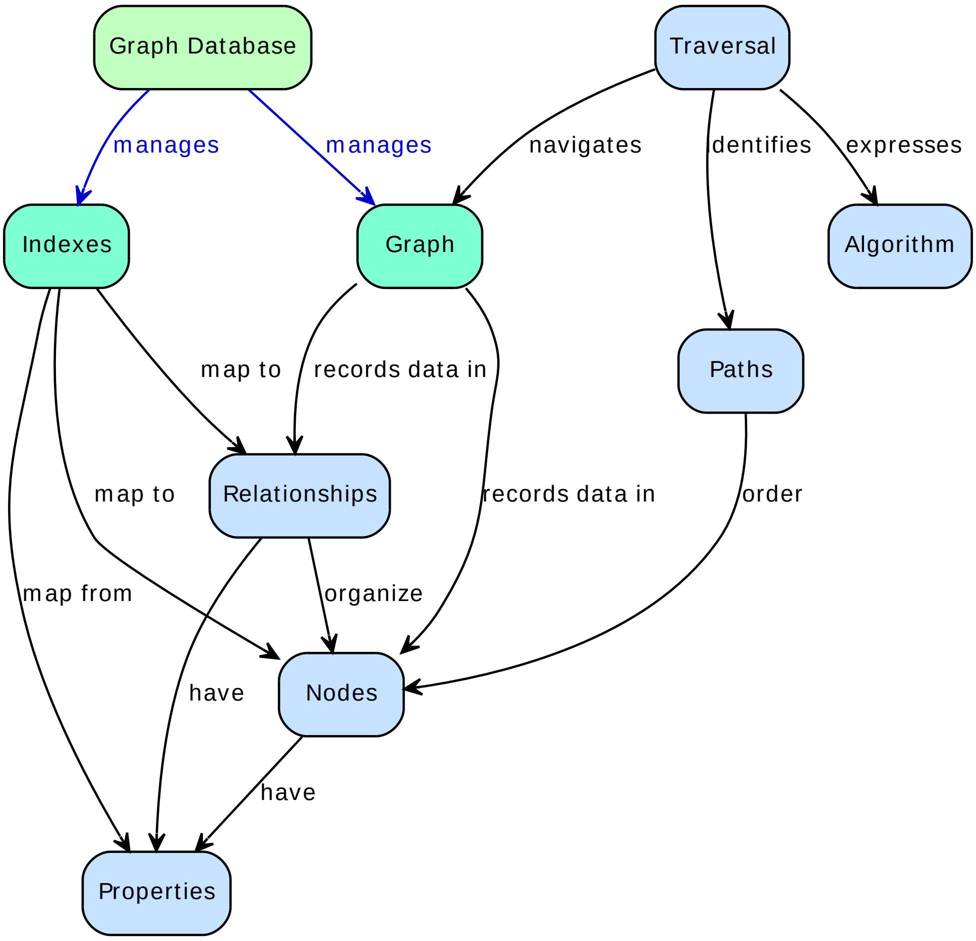
Often, you want to find a specific Node or Relationship according to a Property it has. Rather than traversing the entire graph, use an Index to perform a look-up, for questions like "find the Account for username master-of-graphs."



2.1.5. Neo4j is a Graph Database

"A Graph Database —manages a→ Graph and —also manages related→ Indexes"

Neo4j is a commercially supported open-source graph database. It was designed and built from the ground-up to be a reliable database, optimized for graph structures instead of tables. Working with Neo4j, your application gets all the expressiveness of a graph, with all the dependability you expect out of a database.


Graph Database
Traversal
manages
manages
navigates
dentifies
expresses
Indexes
Graph
Algorithm
map to
records data in
Paths
map to
Relationships
records data in
order
map from
organize
have
Nodes
have
Properties

2.2. Comparing Database Models

A Graph Database stores data structured in the Nodes and Relationships of a graph. How does this compare to other persistence models? Because a graph is a generic structure, let's compare how a few models would look in a graph.

2.2.1. A Graph Database transforms a RDBMS

Topple the stacks of records in a relational database while keeping all the relationships, and you'll see a graph. Where an RDBMS is optimized for aggregated data, Neo4j is optimized for highly connected data.

Figure 2.1. RDBMS

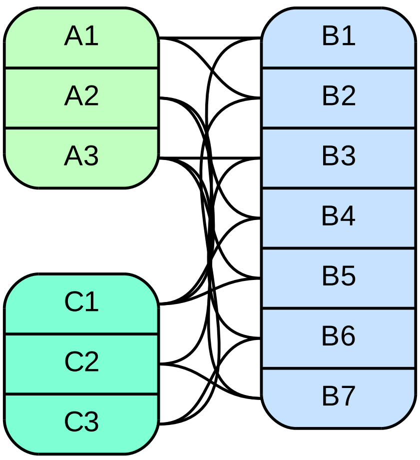
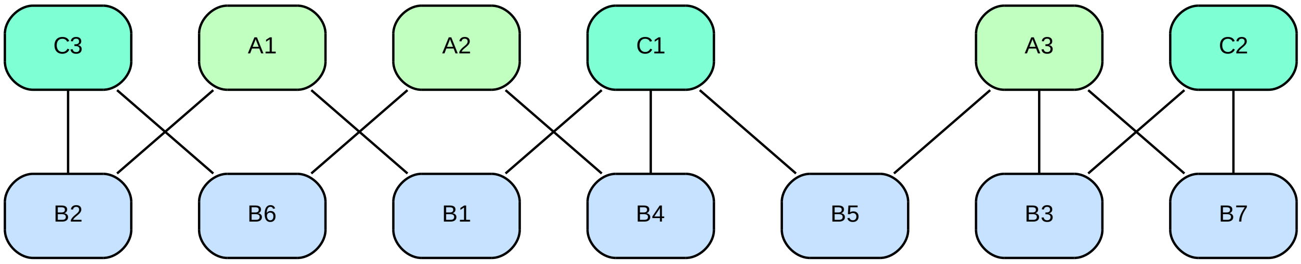


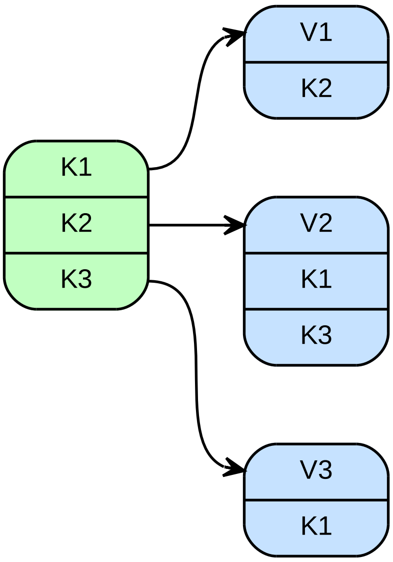
Figure 2.2. Graph Database as RDBMS



2.2.2. A Graph Database elaborates a Key-Value Store

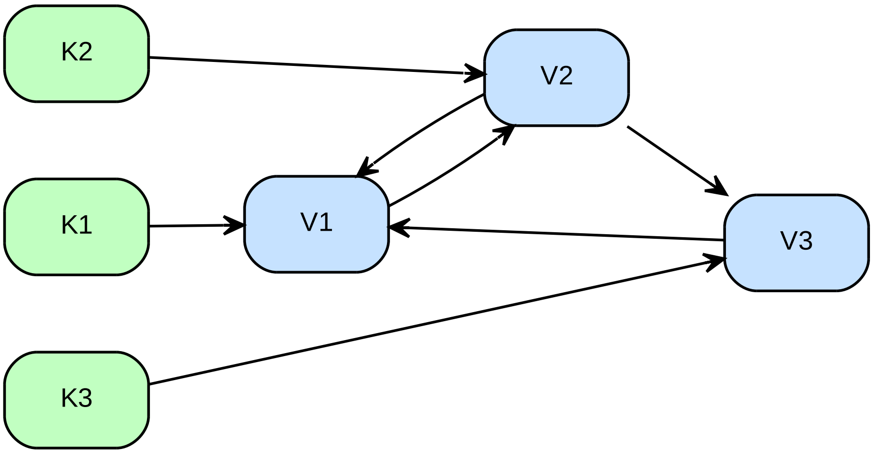
A Key-Value model is great for lookups of simple values or lists. When the values are themselves interconnected, you've got a graph. Neo4j lets you elaborate the simple data structures into more complex, interconnected data.

Figure 2.3. Key-Value Store



`K*` represents a key, `V*` a value. Note that some keys point to other keys as well as plain values.

Figure 2.4. Graph Database as Key-Value Store



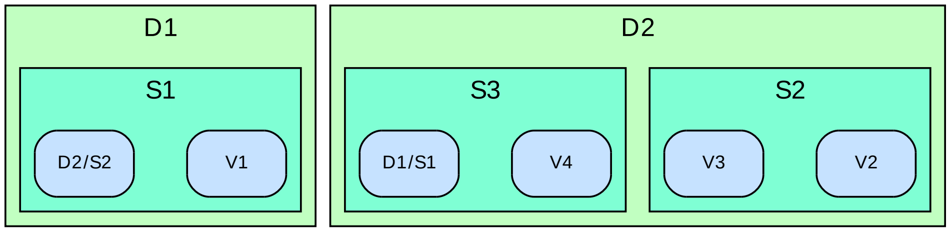
2.2.3. A Graph Database relates Column-Family

Column Family (BigTable-style) databases are an evolution of key-value, using "families" to allow grouping of rows. Stored in a graph, the families could become hierarchical, and the relationships among data becomes explicit.

2.2.4. A Graph Database navigates a Document Store

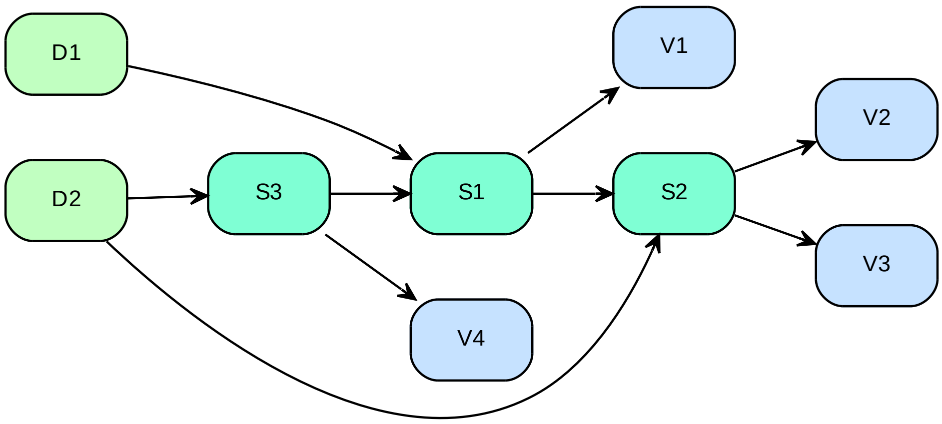
The container hierarchy of a document database accommodates nice, schema-free data that can easily be represented as a tree. Which is of course a graph. Refer to other documents (or document elements) within that tree and you have a more expressive representation of the same data. When in Neo4j, those relationships are easily navigable.

Figure 2.5. Document Store



`D`=Document, `S`=Subdocument, `V`=Value, `D2/S2` = reference to subdocument in (other) document.

Figure 2.6. Graph Database as Document Store



Chapter 3. The Neo4j Graph Database

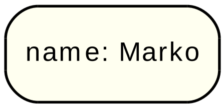
This chapter will introduce more details on the data model and behavior of Neo4j.

3.1. Nodes

The fundamental units that form a graph are nodes and relationships. In Neo4j, both nodes and relationships can contain properties.

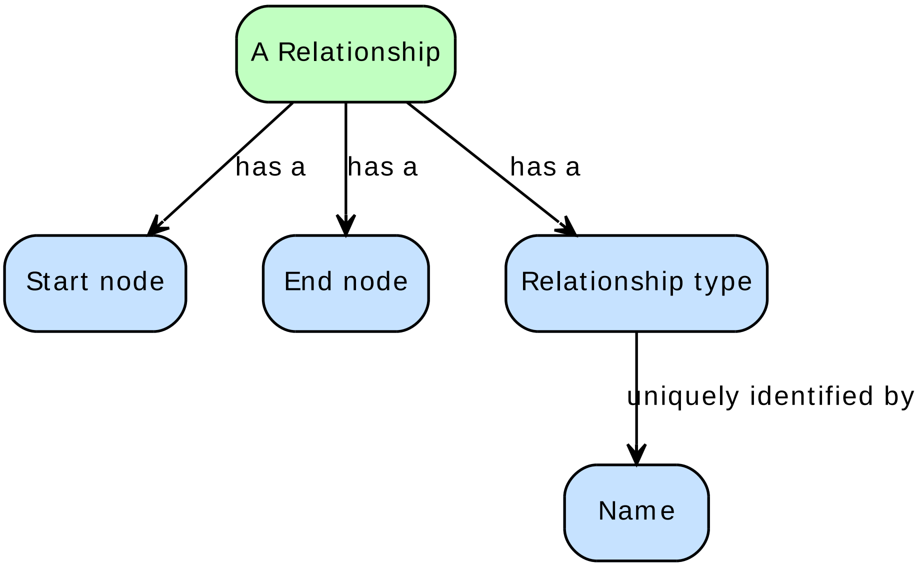
Nodes are often used to represent *entities,* but depending on the domain relationships may be used for that purpose as well.

Let's start out with a really simple graph, containing only a single node with one property:

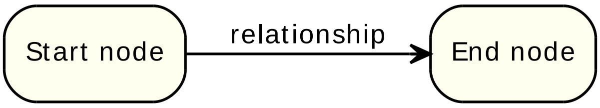


3.2. Relationships

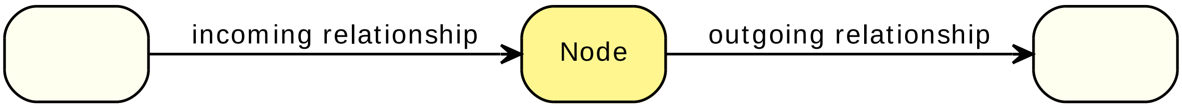
Relationships between nodes are a key part of a graph database. They allow for finding related data.



A relationship connects two nodes, and is guaranteed to have valid start and end nodes.



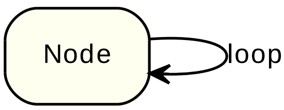
As relationships are always directed, they can be viewed as outgoing or incoming relative to a node, which is useful when traversing the graph:



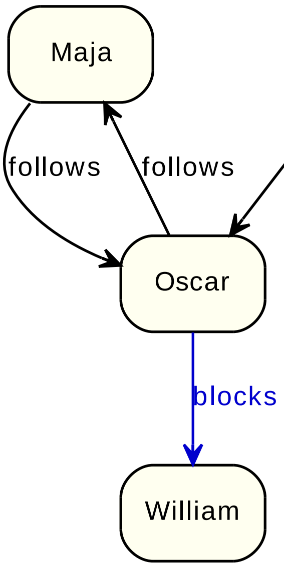
Relationships are equally well traversed in either direction. This means that there is no need to add duplicate relationships in the opposite direction (with regard to traversal or performance).

While relationships always have a direction, you can ignore the direction where it is not useful in your application.

Note that a node can have relationships to itself as well:



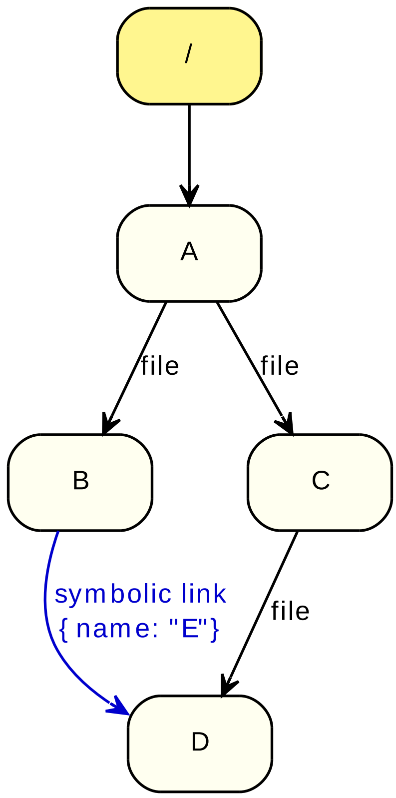
To further enhance graph traversal all relationships have a relationship type. Note that the word *type* might be misleading here, you could rather think of it as a *label*. The following example shows a simple social network with two relationship types.



Using relationship direction and type

What	**How**
get who a person follows	outgoing `follows` relationships, depth one
get the followers of a person	incoming `follows` relationships, depth one
get who a person blocks	outgoing `blocks` relationships, depth one
get who a person is blocked by	incoming `blocks` relationships, depth one

This example is a simple model of a file system, which includes symbolic links:



Depending on what you are looking for, you will use the direction and type of relationships during traversal.

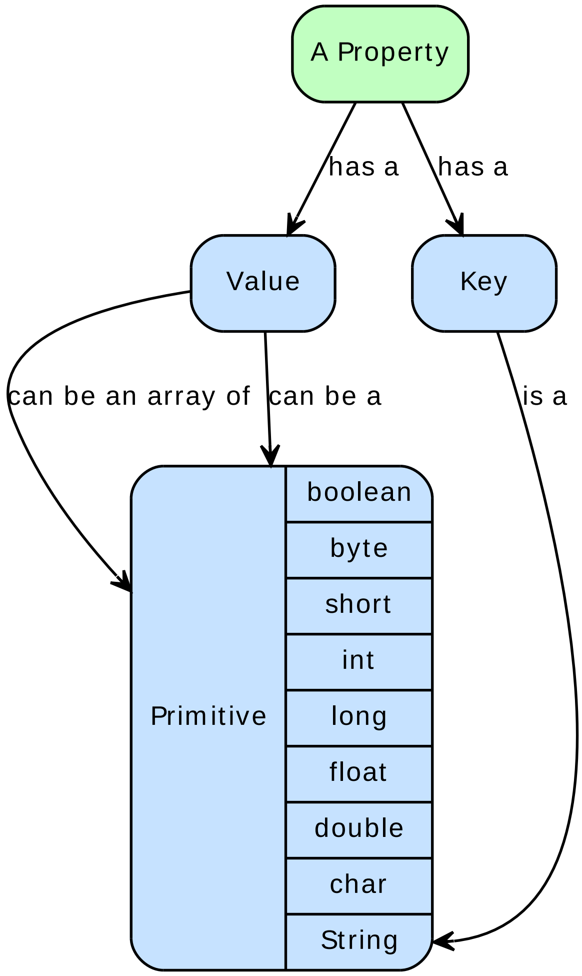
What	How
get the full path of a file	incoming `file` relationships
get all paths for a file	incoming `file` and `symbolic link` relationships
get all files in a directory	outgoing `file` and `symbolic link` relationships, stop at depth one
get all files in a directory, excluding symbolic links	outgoing `file` relationships, stop at depth one
get all files in a directory, recursively	outgoing `file` and `symbolic link` relationships

3.3. Properties

Properties are key-value pairs where the key is a string. Property values can be either a primitive or an array of one primitive type. For example `String`, `int` and `int[]` values are valid for properties.

Note

`null` is not a valid property value. Nulls can instead be modeled by the absence of a key.



Property value types

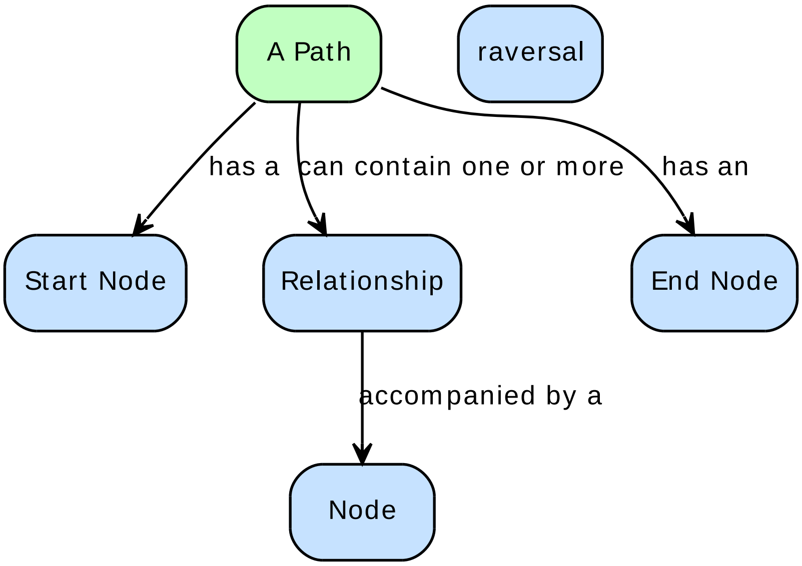
Type	Description	Value range
`boolean`		`true`/`false`
`byte`	8-bit integer	`-128` to `127`, inclusive
`short`	16-bit integer	`-32768` to `32767`, inclusive
`int`	32-bit integer	`-2147483648` to `2147483647`, inclusive
`long`	64-bit integer	`-9223372036854775808` to `9223372036854775807`, inclusive
`float`	32-bit IEEE 754 floating-point number	
`double`	64-bit IEEE 754 floating-point number	

Type	Description	Value range
`char`	16-bit unsigned integers representing Unicode characters	`u0000` to `uffff` (`0` to `65535`)
`String`	sequence of Unicode characters	

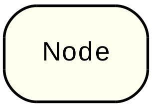
For further details on float/double values, see Java Language Specification <http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.2.3>.

3.4. Paths

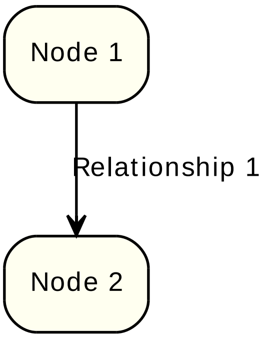
A path is one or more nodes with connecting relationships, typically retrieved as a query or traversal result.



The shortest possible path has length zero and looks like this:



A path of length one:



3.5. Traversal

Traversing a graph means visiting its nodes, following relationships according to some rules. In most cases only a subgraph is visited, as you already know where in the graph the interesting nodes and relationships are found.

Neo4j comes with a callback based traversal API which lets you specify the traversal rules. At a basic level there's a choice between traversing breadth- or depth-first.

For an in-depth introduction to the traversal framework, see Chapter 9, *The Traversal Framework*. For Java code examples see Section 4.4, "Traversal".

Other options to traverse or query graphs in Neo4j are Cypher and Gremlin.

Part II. Tutorials

Chapter 4. Using Neo4j embedded in Java applications

4.1. Include Neo4j in your project

After selecting the appropriate edition for your platform, embed Neo4j in your Java application by including the Neo4j library jars in your build. The following sections will show how to do this by either altering the build path directly or by using dependency management.

4.1.1. Add Neo4j to the build path

Get the Neo4j libraries from one of these sources:

- Extract a Neo4j download <http://neo4j.org/download/> zip/tarball, and use the *jar* files found in the *lib/* directory.
- Use the *jar* files available from Maven Central Repository <http://search.maven.org/#search|ga|1|g%3A%22org.neo4j%22>

Add the jar files to your project:

JDK tools
: Append to `-classpath`

Eclipse
: - Right-click on the project and then go *Build Path → Configure Build Path.* In the dialog, choose *Add External JARs*, browse to the Neo4j *lib/* directory and select all of the jar files.
  - Another option is to use User Libraries <http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/preferences/java/buildpath/ref-preferences-user-libraries.htm>.

IntelliJ IDEA
: See Libraries, Global Libraries, and the Configure Library dialog <http://www.jetbrains.com/idea/webhelp/libraries-global-libraries-and-the-configure-library-dialog.html>

NetBeans
: - Right-click on the *Libraries* node of the project, choose *Add JAR/Folder*, browse to the Neo4j *lib/* directory and select all of the jar files.
  - You can also handle libraries from the project node, see Managing a Project's Classpath <http://netbeans.org/kb/docs/java/project-setup.html#projects-classpath>.

4.1.2. Add Neo4j as a dependency

For an overview of the main Neo4j artifacts, see Neo4j editions. The artifacts listed there are top-level artifacts that will transitively include the actual Neo4j implementation. You can either go with the top-level artifact or include the individual components directly. The examples included here use the top-level artifact approach.

Maven

Maven dependency.

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

```
 </dependencies>
...
</project>
```

Where `${neo4j-version}` *is the desired version and the* `artifactId` *is found in Neo4j editions.*

Eclipse and Maven

For development in Eclipse <http://www.eclipse.org>, it is recommended to install the M2Eclipse plugin <http://www.eclipse.org/m2e/> and let Maven manage the project build classpath instead, see above. This also adds the possibility to build your project both via the command line with Maven and have a working Eclipse setup for development.

Ivy

Make sure to resolve dependencies from Maven Central, for example using this configuration in your *ivysettings.xml* file:

```
<ivysettings>
  <settings defaultResolver="main"/>
  <resolvers>
    <chain name="main">
      <filesystem name="local">
        <artifact pattern="${ivy.settings.dir}/repository/[artifact]-[revision].[ext]" />
      </filesystem>
      <ibiblio name="maven_central" root="http://repo1.maven.org/maven2/" m2compatible="true"/>
    </chain>
  </resolvers>
</ivysettings>
```

With that in place you can add Neo4j to the mix by having something along these lines to your *ivy.xml* file:

```
..
<dependencies>
  ..
  <dependency org="org.neo4j" name="neo4j" rev="${neo4j-version}"/>
  ..
</dependencies>
..
```

Where `${neo4j-version}` *is the desired version and the* `name` *is found in Neo4j editions.*

Gradle

The example below shows an example gradle build script for including the Neo4j libraries.

```
def neo4jVersion = "[set-version-here]"
apply plugin: 'java'
repositories {
   mavenCentral()
}
dependencies {
   compile "org.neo4j:neo4j:${neo4jVersion}"
}
```

Where `neo4jVersion` *is the desired version and the* `name` *("*`neo4j`*" in the example) is found in Neo4j editions.*

4.1.3. Starting and stopping

To create a new database or ópen an existing one you instantiate an `EmbeddedGraphDatabase` `<http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/EmbeddedGraphDatabase.html>`.

```
GraphDatabaseService graphDb = new EmbeddedGraphDatabase( DB_PATH );
registerShutdownHook( graphDb );
```

Note

The `EmbeddedGraphDatabase` instance can be shared among multiple threads. Note however that you can't create multiple instances pointing to the same database.

To stop the database, call the `shutdown()` method:

```
graphDb.shutdown();
```

To make sure Neo4j is shut down properly you can add a shutdown hook:

```
private static void registerShutdownHook( final GraphDatabaseService graphDb )
{
    // Registers a shutdown hook for the Neo4j instance so that it
    // shuts down nicely when the VM exits (even if you "Ctrl-C" the
    // running example before it's completed)
    Runtime.getRuntime().addShutdownHook( new Thread()
    {
        @Override
        public void run()
        {
            graphDb.shutdown();
        }
    } );
}
```

If you want a *read-only view* of the database, use `EmbeddedReadOnlyGraphDatabase` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/EmbeddedReadOnlyGraphDatabase.html>.

To start Neo4j with configuration settings, a Neo4j properties file can be loaded like this:

```
Map<String, String> config = EmbeddedGraphDatabase.loadConfigurations( pathToConfig
                                                                       + "neo4j.properties" );
GraphDatabaseService graphDb = new EmbeddedGraphDatabase(
        "target/database/location", config );
```

Or you could of course create you own `Map<String, String>` programatically and use that instead.

For configuration settings, see Chapter 11, *Configuration & Performance*.

4.2. Hello World

Learn how to create and access nodes, relationships and properties. For information on project setup, see Section 4.1, "Include Neo4j in your project".

Remember, from Section 2.1, "What is a Graph Database?", that a Neo4j graph consist of:

- Nodes that are connected by
- Relationships, with
- Properties on both nodes and relationships.

All relationships have a type. For example, if the graph represents a social network, a relationship type could be `KNOWS`. If a relationship of the type `KNOWS` connects two nodes, that probably represents two people that know each other. A lot of the semantics (that is the meaning) of a graph is encoded in the relationship types of the application. And although relationships are directed they are equally well traversed regardless of which direction they are traversed.

4.2.1. Prepare the database

Relationship types can be created by using an `enum`. In this example we only need a single relationship type. This is how to define it:

```
private static enum RelTypes implements RelationshipType
{
    KNOWS
}
```

The next step is to start the database server. Note that if the directory given for the database doesn't already exist, it will be created.

```
GraphDatabaseService graphDb = new EmbeddedGraphDatabase( DB_PATH );
registerShutdownHook( graphDb );
```

Note that starting a server is an expensive operation, so don't start up a new instance every time you need to interact with the database! The instance can be shared by multiple threads. Transactions are thread confined.

As seen, we register a shutdown hook that will make sure the database shuts down when the JVM exits. Now it's time to interact with the database.

4.2.2. Wrap mutating operations in a transaction

All mutating transactions have to be performed in a transaction. This is a conscious design decision, since we believe transaction demarcation to be an important part of working with a real enterprise database. Now, transaction handling in Neo4j is very easy:

```
Transaction tx = graphDb.beginTx();
try
{
    // Mutating operations go here
    tx.success();
}
finally
{
    tx.finish();
}
```

For more information on transactions, see Chapter 13, *Transaction Management* and Java API for `Transaction` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/Transaction.html>.

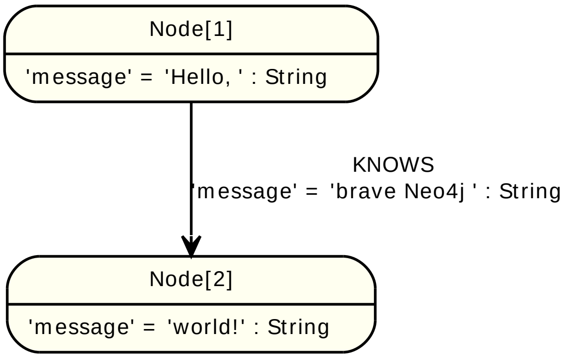
4.2.3. Create a small graph

Now, let's create a few nodes. The API is very intuitive. Feel free to have a look at the JavaDocs at http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/. They're included in the distribution, as well. Here's how to create a small graph consisting of two nodes, connected with one relationship and some properties:

```
Node firstNode = graphDb.createNode();
Node secondNode = graphDb.createNode();

Relationship relationship = firstNode.createRelationshipTo( secondNode, RelTypes.KNOWS );

firstNode.setProperty( "message", "Hello, " );
secondNode.setProperty( "message", "world!" );
relationship.setProperty( "message", "brave Neo4j " );
```

We now have a graph that looks like this:

Figure 4.1. Hello World Graph



4.2.4. Print the result

After we've created our graph, let's read from it and print the result.

```
System.out.print( firstNode.getProperty( "message" ) );
System.out.print( relationship.getProperty( "message" ) );
System.out.print( secondNode.getProperty( "message" ) );
```

Which will output:

```
Hello, brave Neo4j world!
```

4.2.5. Remove the data

In this case we'll remove the data afterwards:

```
// let's remove the data
for ( Node node : graphDb.getAllNodes() )
{
    for ( Relationship rel : node.getRelationships() )
    {
        rel.delete();
    }
    node.delete();
}
```

Note that deleting a node which still has relationships when the transaction commits will fail. This is to make sure relationships always have a start node and an end node.

4.2.6. Shut down the database server

Finally, shut down the database server *when the application finishes:*

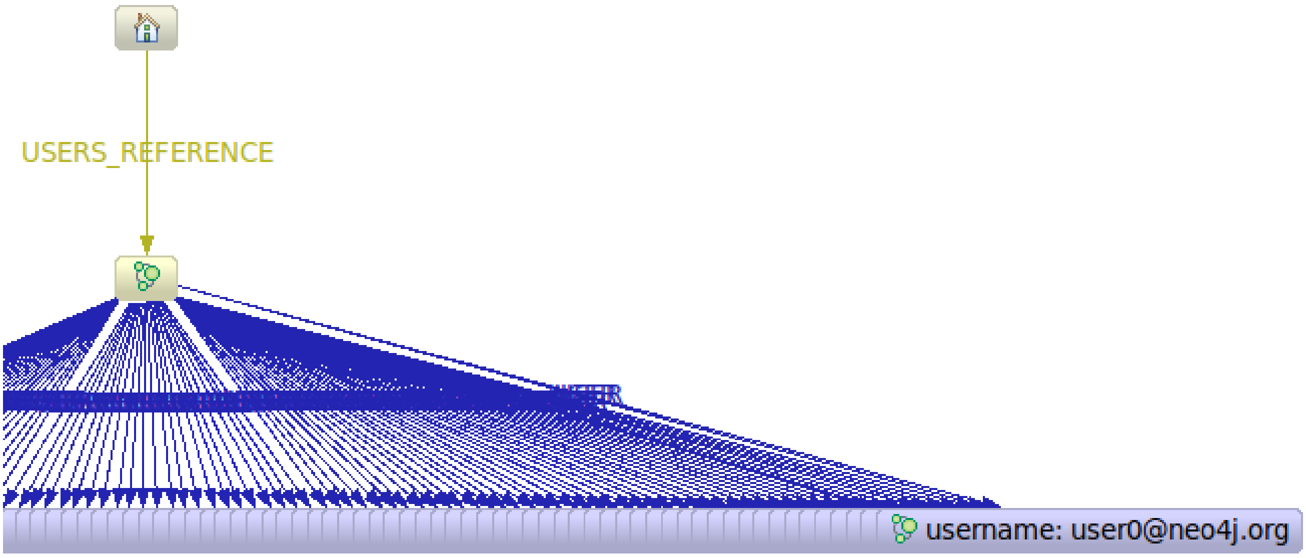
```
graphDb.shutdown();
```

Full source code: HelloWorldTest.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/test/java/org/neo4j/examples/HelloWorldTest.java>

4.3. User database with index

You have a user database, and want to retrieve users by name. To begin with, this is the structure of the database we want to create:

Figure 4.2. Node space view of users



That is, the reference node is connected to a users-reference node to which all users are connected.

To begin with, we define the relationship types we want to use:

```
private static enum RelTypes implements RelationshipType
{
    USERS_REFERENCE,
    USER
}
```

Then we have created two helper methods to handle user names and adding users to the database:

```
private static String idToUserName( final int id )
{
    return "user" + id + "@neo4j.org";
}

private static Node createAndIndexUser( final String username )
{
    Node node = graphDb.createNode();
    node.setProperty( USERNAME_KEY, username );
    nodeIndex.add( node, USERNAME_KEY, username );
    return node;
}
```

The next step is to start the database server:

```
graphDb = new EmbeddedGraphDatabase( DB_PATH );
nodeIndex = graphDb.index().forNodes( "nodes" );
registerShutdownHook();
```

It's time to add the users:

```
Transaction tx = graphDb.beginTx();
try
{
    // Create users sub reference node (see design guidelines on
    // http://wiki.neo4j.org/ )
```

```
    Node usersReferenceNode = graphDb.createNode();
    graphDb.getReferenceNode().createRelationshipTo(
        usersReferenceNode, RelTypes.USERS_REFERENCE );
    // Create some users and index their names with the IndexService
    for ( int id = 0; id < 100; id++ )
    {
        Node userNode = createAndIndexUser( idToUserName( id ) );
        usersReferenceNode.createRelationshipTo( userNode,
            RelTypes.USER );
    }
```

And here's how to find a user by Id:

```
int idToFind = 45;
Node foundUser = nodeIndex.get( USERNAME_KEY,
    idToUserName( idToFind ) ).getSingle();
System.out.println( "The username of user " + idToFind + " is "
    + foundUser.getProperty( USERNAME_KEY ) );
```

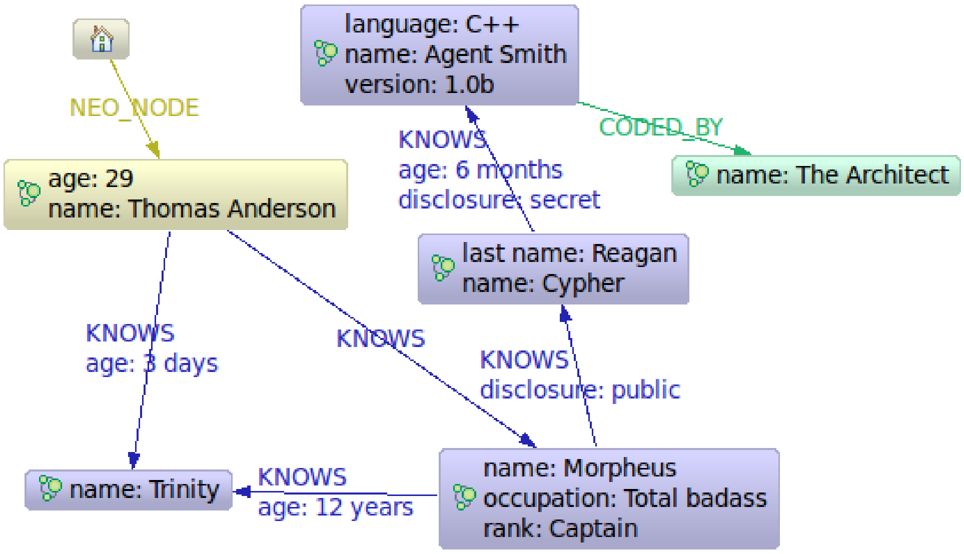
Full source code: EmbeddedNeo4jWithIndexing.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/main/java/org/neo4j/examples/EmbeddedNeo4jWithIndexing.java>

4.4. Traversal

4.4.1. The Matrix

This is the first node space we want to traverse into:

Figure 4.3. Matrix node space view



Friends and friends of friends.

```
private static Traverser getFriends( final Node person )
{
    return person.traverse( Order.BREADTH_FIRST,
            StopEvaluator.END_OF_GRAPH,
            ReturnableEvaluator.ALL_BUT_START_NODE, RelTypes.KNOWS,
            Direction.OUTGOING );
}
```

Let's perform the actual traversal and print the results:

```
Traverser friendsTraverser = getFriends( neoNode );
int numberOfFriends = 0;
for ( Node friendNode : friendsTraverser )
{
    System.out.println( "At depth "
            + friendsTraverser.currentPosition()
            .depth() + " => "
            + friendNode.getProperty( "name" ) );
}
```

Who coded the Matrix?

```
private static Traverser findHackers( final Node startNode )
{
    return startNode.traverse( Order.BREADTH_FIRST,
            StopEvaluator.END_OF_GRAPH, new ReturnableEvaluator()
    {
        @Override
        public boolean isReturnableNode(
                final TraversalPosition currentPos )
        {
            return !currentPos.isStartNode()
            && currentPos.lastRelationshipTraversed()
            .isType( RelTypes.CODED_BY );
```

```
        }
    }, RelTypes.CODED_BY, Direction.OUTGOING, RelTypes.KNOWS,
    Direction.OUTGOING );
}
```

Print out the result:

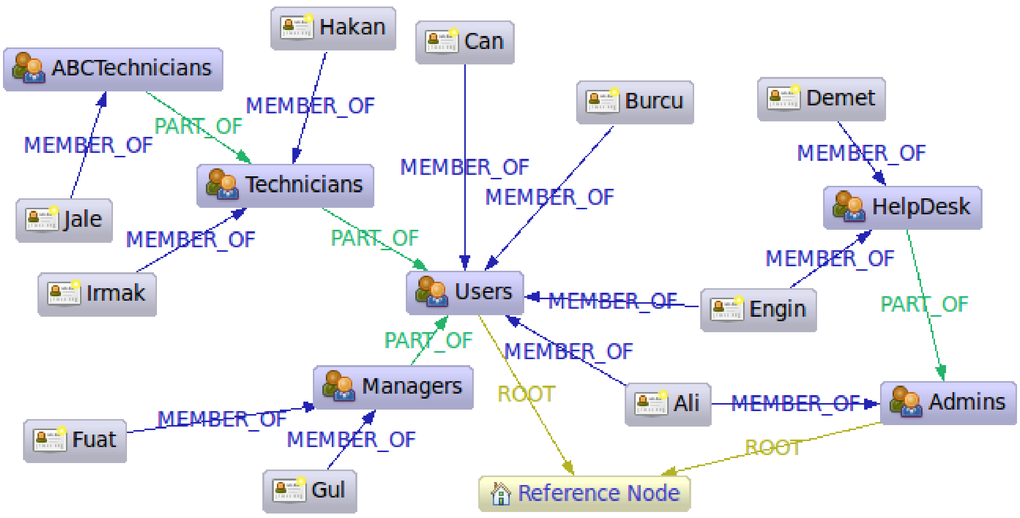
```
Traverser traverser = findHackers( getNeoNode() );
int numberOfHackers = 0;
for ( Node hackerNode : traverser )
{
    System.out.println( "At depth " + traverser.currentPosition()
            .depth() + " => " + hackerNode.getProperty( "name" ) );
}
```

Full source code: MatrixTest.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/test/java/org/neo4j/examples/MatrixTest.java>

4.4.2. User roles

Here we have users assigned to groups, and groups containing other groups. This is the full node space of our example:

Figure 4.4. User roles node space view



Get the admins.

```
Node admins = getNodeByName( "Admins" );
Traverser traverser = admins.traverse( Traverser.Order.BREADTH_FIRST,
        StopEvaluator.END_OF_GRAPH,
        ReturnableEvaluator.ALL_BUT_START_NODE, RoleRels.PART_OF,
        Direction.INCOMING, RoleRels.MEMBER_OF, Direction.INCOMING );
```

Get the group memberships of a user.

```
Node jale = getNodeByName( "Jale" );
traverser = jale.traverse( Traverser.Order.DEPTH_FIRST,
        StopEvaluator.END_OF_GRAPH,
        ReturnableEvaluator.ALL_BUT_START_NODE, RoleRels.MEMBER_OF,
        Direction.OUTGOING, RoleRels.PART_OF, Direction.OUTGOING );
```

Get all groups.

```
Node referenceNode = getNodeByName( "Reference_Node") ;
traverser = referenceNode.traverse(
        Traverser.Order.BREADTH_FIRST, StopEvaluator.END_OF_GRAPH,
```

```
        ReturnableEvaluator.ALL_BUT_START_NODE, RoleRels.ROOT,
        Direction.INCOMING, RoleRels.PART_OF, Direction.INCOMING );
```

Get all members of all groups.

```
traverser = referenceNode.traverse(
        Traverser.Order.BREADTH_FIRST,
        StopEvaluator.END_OF_GRAPH,
        new ReturnableEvaluator()
        {
            @Override
            public boolean isReturnableNode(
                    TraversalPosition currentPos )
            {
                if ( currentPos.isStartNode() )
                {
                    return false;
                }
                Relationship rel = currentPos.lastRelationshipTraversed();
                return rel.isType( RoleRels.MEMBER_OF );
            }
        }, RoleRels.ROOT, Direction.INCOMING, RoleRels.PART_OF,
        Direction.INCOMING, RoleRels.MEMBER_OF, Direction.INCOMING );
```

Full source code: RolesTest.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/test/java/org/neo4j/examples/RolesTest.java>

4.4.3. New traversal framework

> **Note**
> The following examples use a new experimental traversal API. It shares the underlying implementation with the old traversal API, so performance-wise they should be equal. However, expect the new API to evolve and thus undergo changes.

The Matrix

The traversals from the Matrix example above, this time using the new traversal API:

Friends and friends of friends.

```
private static Traverser getFriends( final Node person )
{
    TraversalDescription td = Traversal.description()
            .breadthFirst()
            .relationships( RelTypes.KNOWS, Direction.OUTGOING )
            .evaluator( Evaluators.excludeStartPosition() );
    return td.traverse( person );
}
```

Let’s perform the actual traversal and print the results:

```
Traverser friendsTraverser = getFriends( neoNode );
int numberOfFriends = 0;
for ( Path friendPath : friendsTraverser )
{
    System.out.println( "At depth " + friendPath.length() + " => "
                        + friendPath.endNode()
                                .getProperty( "name" ) );
}
```

Who coded the Matrix?

```
private static Traverser findHackers( final Node startNode )
{
```

```
    TraversalDescription td = Traversal.description()
            .breadthFirst()
            .relationships( RelTypes.CODED_BY, Direction.OUTGOING )
            .relationships( RelTypes.KNOWS, Direction.OUTGOING )
            .evaluator(
                    Evaluators.returnWhereLastRelationshipTypeIs( RelTypes.CODED_BY ) );
    return td.traverse( startNode );
}
```

Print out the result:

```
Traverser traverser = findHackers( getNeoNode() );
int numberOfHackers = 0;
for ( Path hackerPath : traverser )
{
    System.out.println( "At depth " + hackerPath.length() + " => "
                        + hackerPath.endNode()
                                .getProperty( "name" ) );
}
```

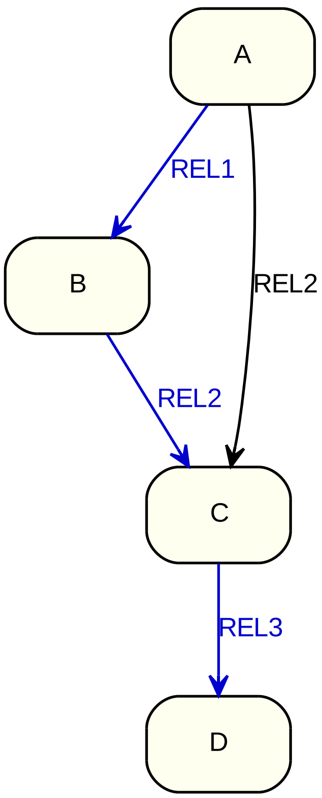
Full source code: MatrixNewTravTest.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/test/java/org/neo4j/examples/MatrixNewTravTest.java>

Walking an ordered path

This example shows how to use a path context holding a representation of a path.

Create a toy graph.

```
Node A = db.createNode();
Node B = db.createNode();
Node C = db.createNode();
Node D = db.createNode();
A.createRelationshipTo( B, REL1 );
B.createRelationshipTo( C, REL2 );
C.createRelationshipTo( D, REL3 );
A.createRelationshipTo( C, REL2 );
```



Now, the order of relationships (`REL1` → `REL2` → `REL3`) is stored in an `ArrayList`. Upon traversal, the `Evaluator` can check against it to ensure that only paths are included and returned that have the predefined order of relationships:

Walk the path.

```
final ArrayList<RelationshipType> orderedPathContext = new ArrayList<RelationshipType>();
orderedPathContext.add( REL1 );
orderedPathContext.add( withName( "REL2" ) );
orderedPathContext.add( withName( "REL3" ) );
TraversalDescription td = Traversal.description().evaluator(
        new Evaluator()
        {
            @Override
            public Evaluation evaluate( final Path path )
            {
                if ( path.length() == 0 )
                {
                    return Evaluation.EXCLUDE_AND_CONTINUE;
                }
                String currentName = path.lastRelationship().getType().name();
                String relationshipType = orderedPathContext.get(
                        path.length() - 1 ).name();
                if ( path.length() == orderedPathContext.size() )
                {
                    if ( currentName.equals( relationshipType ) )
                    {
                        return Evaluation.INCLUDE_AND_PRUNE;
                    }
                    else
                    {
                        return Evaluation.EXCLUDE_AND_PRUNE;
                    }
                }
                else
                {
                    if ( currentName.equals( relationshipType ) )
                    {
                        return Evaluation.EXCLUDE_AND_CONTINUE;
                    }
                    else
                    {
                        return Evaluation.EXCLUDE_AND_PRUNE;
                    }
                }
            }
        } );
Traverser t = td.traverse( db.getNodeById( 1 ) );
for ( Path path : t )
{
    System.out.println( path );
}
```

Full source code: OrderedPathTest.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/test/java/org/neo4j/examples/orderedpath/OrderedPathTest.java>

4.4.4. Social network

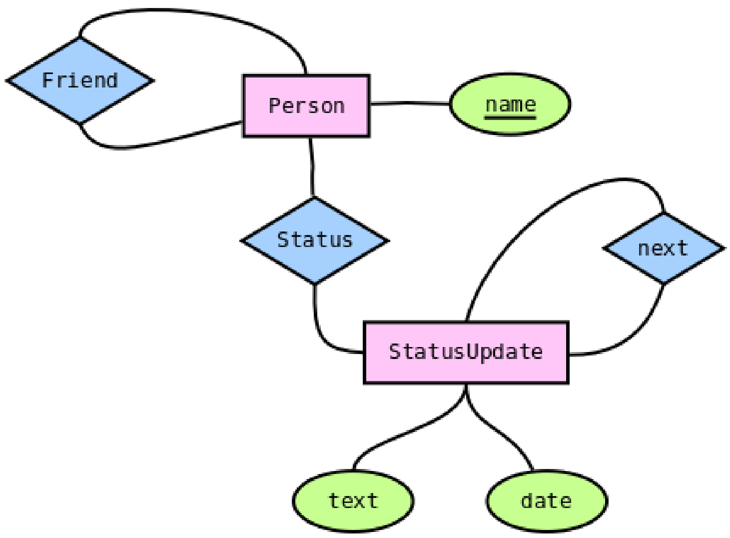
Note

The following example uses the new experimental traversal API.

Social networks (know as social graphs out on the web) are natural to model with a graph. This example shows a very simple social model that connects friends and keeps track of status updates.

Simple social model

Figure 4.5. Social network data model



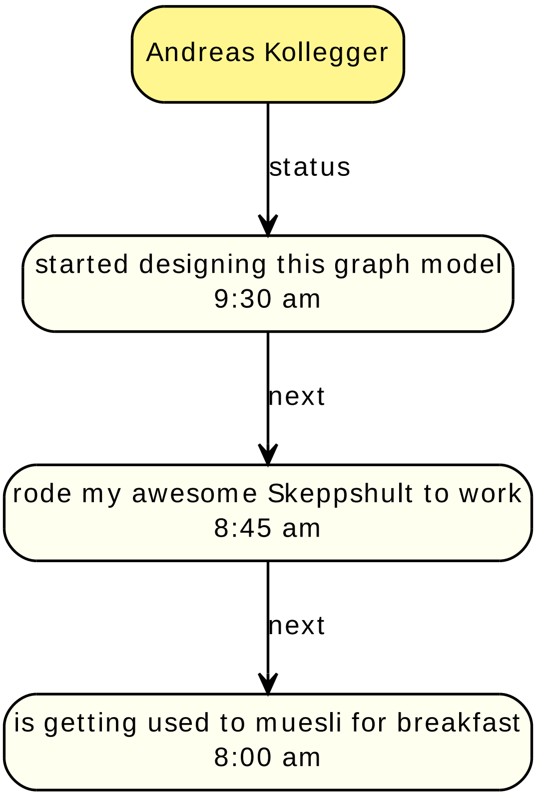
The data model for a social network is pretty simple: `Persons` with names and `StatusUpdates` with timestamped text. These entities are then connected by specific relationships.

- `Person`
  - `friend`: relates two distinct `Person` instances (no self-reference)
  - `status`: connects to the most recent `StatusUpdate`
- `StatusUpdate`
  - `next`: points to the next `StatusUpdate` in the chain, which was posted before the current one

Status graph instance

The `StatusUpdate` list for a `Person` is a linked list. The head of the list (the most recent status) is found by following `status`. Each subsequent `StatusUpdate` is connected by `next`.

Here's an example where Andreas Kollegger micro-blogged his way to work in the morning:



To read the status updates, we can create a traversal, like so:

```
TraversalDescription traversal = Traversal.description().
        depthFirst().
        relationships( NEXT ).
        filter( Traversal.returnAll() );
```

This gives us a traverser that will start at one `StatusUpdate`, and will follow the chain of updates until they run out. Traversers are lazy loading, so it's performant even when dealing with thousands of statuses - they are not loaded until we actually consume them.

Activity stream

Once we have friends, and they have status messages, we might want to read our friends status' messages, in reverse time order - latest first. To do this, we go through these steps:

1. Gather all friend's status update iterators in a list - latest date first.
2. Sort the list.
3. Return the first item in the list.
4. If the first iterator is exhausted, remove it from the list. Otherwise, get the next item in that iterator.
5. Go to step 2 until there are no iterators left in the list.

Animated, the sequence looks like this <http://www.slideshare.net/systay/pattern-activity-stream>.

The code looks like:

```
PositionedIterator<StatusUpdate> first = statuses.get(0);
StatusUpdate returnVal = first.current();

if ( !first.hasNext() )
{
    statuses.remove( 0 );
}
```

```
else
{
    first.next();
    sort();
}

return returnVal;
```

Full source code: socnet <https://github.com/neo4j/community/tree/master/embedded-examples/src/main/java/org/neo4j/examples/socnet>

4.5. Domain entities

This page demonstrates one way to handle domain entities when using Neo4j. The principle at use is to wrap the entities around a node (the same approach can be used with relationships as well).

First off, store the node and make it accessible inside the package:

```
private final Node underlyingNode;

Person( Node personNode )
{
    this.underlyingNode = personNode;
}

protected Node getUnderlyingNode()
{
    return underlyingNode;
}
```

Delegate attributes to the node:

```
public String getName()
{
    return (String)underlyingNode.getProperty( NAME );
}
```

Make sure to override these methods:

```
@Override
public int hashCode()
{
    return underlyingNode.hashCode();
}

@Override
public boolean equals( Object o )
{
    return o instanceof Person &&
            underlyingNode.equals( ( (Person)o ).getUnderlyingNode() );
}

@Override
public String toString()
{
    return "Person[" + getName() + "]";
}
```

Full source code: Person.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/main/java/org/neo4j/examples/socnet/Person.java>

4.6. Graph Algorithm examples

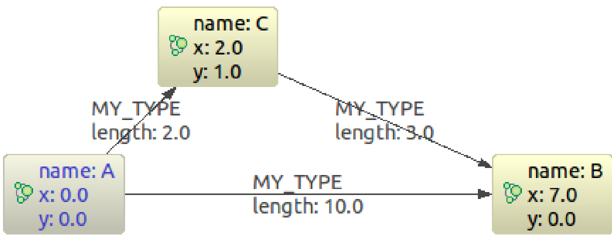
Calculating the shortest path (least number of relationships) between two nodes:

```
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;
    }
```

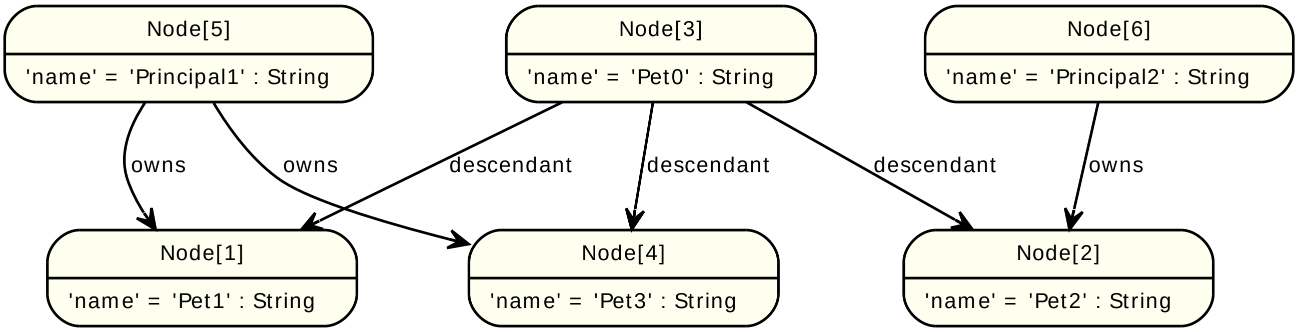
```
};
PathFinder<WeightedPath> astar = GraphAlgoFactory.aStar(
        Traversal.expanderForAllTypes(),
        CommonEvaluators.doubleCostEvaluator( "length" ), estimateEvaluator );
WeightedPath path = astar.findSinglePath( nodeA, nodeB );
```

Full source code: PathFindingExamplesTest.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/test/java/org/neo4j/examples/PathFindingExamplesTest.java>

4.7. Uniqueness of Paths in traversals

This example is demonstrating the use of node uniqueness. Below an imaginary domain graph with Principals that own pets that are descendant to other pets.

Figure 4.6. Descendants Example Graph



In order to return all descendants of `Pet0` which have the relation `owns` to `Principal1` (`Pet1` and `Pet3`), the Uniqueness of the traversal needs to be set to `NODE_PATH` rather than the default `NODE_GLOBAL` so that nodes can be traversed more that once, and paths that have different nodes but can have some nodes in common (like the start and end node) can be returned.

```
final Node target = data.get().get( "Principal1" );
TraversalDescription td = Traversal.description()
        .uniqueness( Uniqueness.NODE_PATH )
        .evaluator( new Evaluator()
{
    @Override
    public Evaluation evaluate( Path path )
    {
        if ( path.endNode().equals( target ) )
        {
            return Evaluation.INCLUDE_AND_PRUNE;
        }
        return Evaluation.EXCLUDE_AND_CONTINUE;
    }
} );

Traverser results = td.traverse( start );
```

This will return the following paths:

```
(3)--[descendant,0]-->(1)<--[owns,3]--(5)
(3)--[descendant,2]-->(4)<--[owns,5]--(5)
```

Let's create a new `TraversalDescription` from the old one, having `NODE_GLOBAL` uniqueness to see the difference.

Tip

The `TraversalDescription` object is immutable, so we have to use the new instance returned with the new uniqueness setting.

```
TraversalDescription nodeGlobalTd = td.uniqueness( Uniqueness.NODE_GLOBAL );
results = nodeGlobalTd.traverse( start );
```

Now only one path is returned:

```
(3)--[descendant,0]-->(1)<--[owns,3]--(5)
```

Full source code: UniquenessOfPathsTest.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/test/java/org/neo4j/examples/UniquenessOfPathsTest.java>

4.8. Reading a management attribute

Reading a management attribute

The EmbeddedGraphDatabase <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/EmbeddedGraphDatabase.html> class includes a convenience method <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/EmbeddedGraphDatabase.html#getManagementBean%28java.lang.Class%29> to get instances of Neo4j management beans. The common JMX service can be used as well, but from your code you probably rather want to use the approach outlined here.

This example shows how to get the start time of a database:

```
private static Date getStartTimeFromManagementBean(
        GraphDatabaseService graphDbService )
{
    // use EmbeddedGraphDatabase to access management beans
    EmbeddedGraphDatabase graphDb = (EmbeddedGraphDatabase) graphDbService;
    Kernel kernel = graphDb.getManagementBean( Kernel.class );
    Date startTime = kernel.getKernelStartTime();
    return startTime;
}
```

Depending on which Neo4j edition you are using different sets of management beans are available.

- For all editions, see the org.neo4j.jmx <http://components.neo4j.org/neo4j-jmx/1.5-SNAPSHOT/apidocs/org/neo4j/jmx/package-summary.html> package.
- For the Advanced and Enterprise editions, see the org.neo4j.management <http://components.neo4j.org/neo4j-management/1.5-SNAPSHOT/apidocs/org/neo4j/management/package-summary.html> package as well.

Full source code: JmxTest.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/test/java/org/neo4j/examples/JmxTest.java>

Chapter 5. Using Neo4j embedded in Python applications

For instructions on how to install the Python Neo4j driver, see Section 20.1, “Installation”.

For general information on the Python language binding, see Chapter 20, *Python embedded bindings*.

5.1. Hello, world!

Here is a simple example to get you started.

```
from neo4j import GraphDatabase

# Create a database
db = GraphDatabase(folder_to_put_db_in)

# All write operations happen in a transaction
with db.transaction:
    firstNode = db.node(name='Hello')
    secondNode = db.node(name='world!')

    # Create a relationship with type 'knows'
    relationship = firstNode.knows(secondNode, name='graphy')

# Read operations can happen anywhere
message = ' '.join([firstNode['name'], relationship['name'], secondNode['name']])

print message

# Delete the data
with db.transaction:
    firstNode.knows.single.delete()
    firstNode.delete()
    secondNode.delete()

# Always shut down your database when your application exits
db.shutdown()
```

5.2. A sample app using traversals and indexes

For detailed documentation on the concepts use here, see Section 20.4, "Indexes" and Section 20.3, "Traversals".

This example shows you how to get started building something like a simple invoice tracking application with Neo4j.

We start out by importing Neo4j, and creating some meta data that we will use to organize our actual data with.

```
from neo4j import GraphDatabase, INCOMING, Evaluation

# Create a database
db = GraphDatabase(folder_to_put_db_in)

# All write operations happen in a transaction
with db.transaction:

    # A node to connect customers to
    customers = db.node()

    # A node to connect invoices to
    invoices = db.node()

    # Connected to the reference node, so
    # that we can always find them.
    db.reference_node.CUSTOMERS(customers)
    db.reference_node.INVOICES(invoices)

    # An index, helps us rapidly look up customers
    customer_idx = db.node.indexes.create('customers')
```

5.2.1. Domain logic

Then we define some domain logic that we want our application to be able to perform. Our application has two domain objects, Customers and Invoices. Let's create methods to add new customers and invoices.

```
def create_customer(name):
    with db.transaction:
        customer = db.node(name=name)
        customer.INSTANCE_OF(customers)

        # Index the customer by name
        customer_idx['name'][name] = customer
    return customer

def create_invoice(customer, amount):
    with db.transaction:
        invoice = db.node(amount=amount)
        invoice.INSTANCE_OF(invoices)

        invoice.RECIPIENT(customer)
    return customer
```

In the customer case, we create a new node to represent the customer and connect it to the "customers" node. This helps us find customers later on, as well as determine if a given node is a customer.

We also index the name of the customer, to allow quickly finding customers by name.

In the invoice case, we do the same, except no indexing. We also connect each new invoice to the customer it was sent to, using a relationship of type "SENT_TO".

Next, we want to be able to retrieve customers and invoices that we have added. Because we are indexing customer names, finding them is quite simple.

```
def get_customer(name):
    return customer_idx['name'][name].single
```

Lets say we also like to do something like finding all invoices for a given customer that are above some given amount. This could be done by writing a traversal, like this:

```
def get_invoices_with_amount_over(customer, min_sum):
    def evaluator(path):
        node = path.end
        if node.has_key('amount') and node['amount'] > min_sum:
            return Evaluation.INCLUDE_AND_CONTINUE
        return Evaluation.EXCLUDE_AND_CONTINUE

    return db.traversal()\
             .relationships('RECIPIENT', INCOMING)\
             .evaluator(evaluator)\
             .traverse(customer)\
             .nodes
```

5.2.2. Creating data and getting it back

Putting it all together, we can create customers and invoices, and use the search methods we wrote to find them.

```
for name in ['Acme Inc.', 'Example Ltd.']:
   create_customer(name)

# Loop through customers
for relationship in customers.INSTANCE_OF:
   customer = relationship.start
   for i in range(1,12):
       create_invoice(customer, 100 * i)

# Finding large invoices
large_invoices = get_invoices_with_amount_over(get_customer('Acme Inc.'), 500)

# Getting all invoices per customer:
for relationship in get_customer('Acme Inc.').RECIPIENT.incoming:
    invoice = relationship.start
```

Chapter 6. Extending the Neo4j Server

The Neo4j Server can be extended by either plugins or unmanaged extensions. For more information on the server, see Chapter 17, *Neo4j Server*.

6.1. 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 relationships.
- 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, properties or indices.

Tip

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.

The needed classes reside in the org.neo4j:server-api <http://search.maven.org/#search|gav|1|g%3A%22org.neo4j%22%20AND%20a%3A%22server-api%22> jar file. See the linked page for downloads and instructions on how to include it using dependency management. For Maven projects, add the Server API dependencies in your `pom.xml` like this:

```
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>server-api</artifactId>
  <version>${neo4j-version}</version>
</dependency>
```

Where ${neo4j-version} is the intended version.

To create a plugin, your code must inherit from the ServerPlugin <http://components.neo4j.org/server-api/1.5-SNAPSHOT/apidocs/org/neo4j/server/plugins/ServerPlugin.html> class. Your plugin should also:

- ensure that it can produce an (Iterable of) Node, Relationship or Path,
- specify parameters,
- specify a point of extension and of course
- contain the application logic.
- make sure that the discovery point type in the `@PluginTarget` and the `@Source` parameter are of the same type.

An example of a plugin which augments the database (as opposed to nodes or relationships) follows:

Get all nodes or relationships plugin.

```
@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();
            }
        };
    }
}
```

The full source code is found here: GetAll.java <https://github.com/neo4j/community/blob/master/server-examples/src/main/java/org/neo4j/examples/server/plugins/GetAll.java>

Find the shortest path between two nodes plugin.

```
public class ShortestPath extends ServerPlugin
{
    @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 insignificant, if omitted all types are followed." )
                @Parameter( name = "types", optional = true ) String[] types,
            @Description( "The maximum path length to search for, default value (if omitted) is 4." )
                @Parameter( name = "depth", optional = true ) Integer depth )
    {
        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 );
    }
}
```

The full source code is found here: ShortestPath.java <https://github.com/neo4j/community/blob/master/server-examples/src/main/java/org/neo4j/examples/server/plugins/ShortestPath.java>

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 data model 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. Extension names will be automatically assigned, based on method names, if not specifically specified using the `@Name` annotation.

```
{
"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 and encoded as JSON data content. F.ex for calling the `shortest path` extension (URI gotten from a `GET` to http://localhost:7474/db/data/node/123):

```
curl -X POST http://localhost:7474/db/data/ext/GetAll/node/123/shortestPath -H "Content-Type: application/json" -d '{"target":"http:/
```

If everything is OK a response code `200` and a list of zero or more items will be returned. If nothing is returned (null returned from extension) an empty result and response code `204` will be returned. If the extension throws an exception response code `500` and a detailed error message is returned.

Extensions that do any kind of write operation will have to manage their own transactions, i.e. transactions aren't managed automatically.

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).

6.2. 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 likely 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.

Warning

This is a sharp tool, allowing users to deploy arbitrary JAX-RS <http://en.wikipedia.org/wiki/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 JAX-RS jar and any Neo4j jars you're making use of. Then add 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 this:

Unmanaged extension example.

```
@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();
    }
}
```

The full source code is found here: HelloWorldResource.java <https://github.com/neo4j/community/blob/master/server-examples/src/main/java/org/neo4j/examples/server/unmanaged/HelloWorldResource.java>

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 the `neo4j-server.properties` file, 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
```

Which binds the hello method to respond to `GET` requests at the URI: `http://{neo4j_server}:{neo4j_port}/examples/unmanaged/helloworld/{nodeId}`

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

which results in

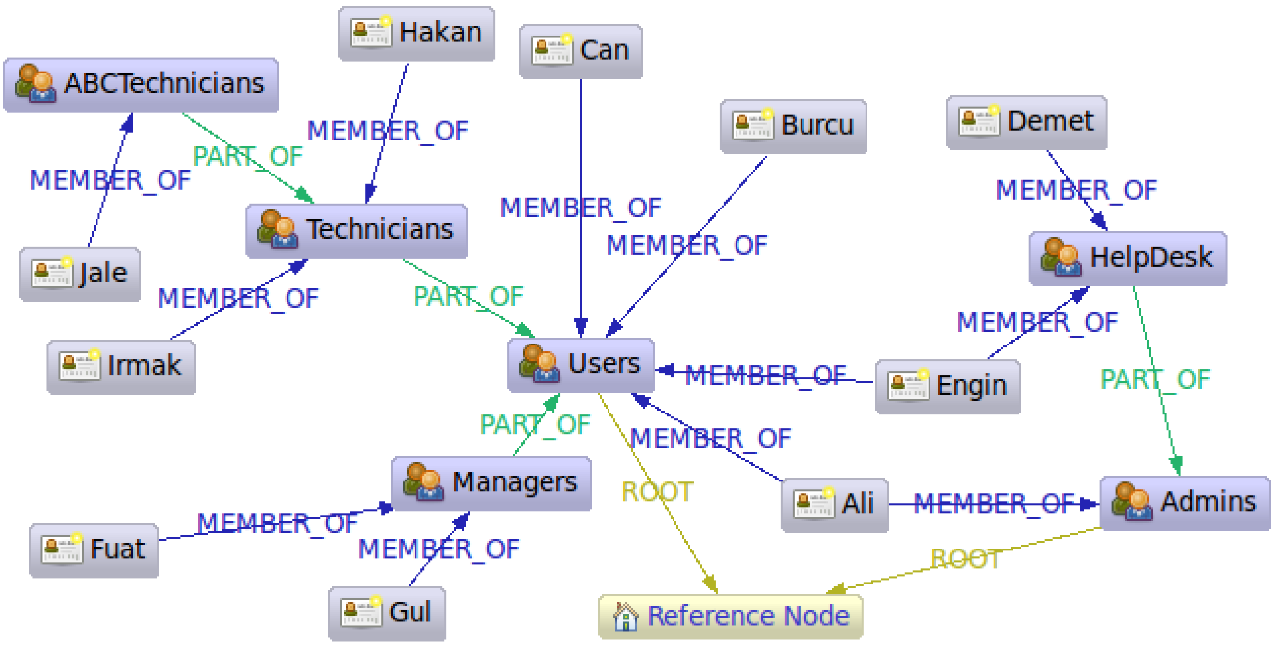
```
Hello World, nodeId=123
```

Chapter 7. Domain Modeling Gallery

The following chapters contains simplified examples of how different domains can be modeled using Neo4j. The aim is not to give full examples, but to suggest possible ways to think using the *graph patterns* and *data locality* in traversals.

7.1. User roles in graphs

This is an example showing a hierarchy of roles. What's interesting is that a tree is not sufficient for storing this structure, as elaborated below.



This is an implementation of an example found in the article A Model to Represent Directed Acyclic Graphs (DAG) on SQL Databases <http://www.codeproject.com/KB/database/Modeling_DAGs_on_SQL_DBs.aspx> by Kemal Erdogan <http://www.codeproject.com/script/Articles/MemberArticles.aspx?amid=274518>. The article discusses how to store directed acyclic graphs <http://en.wikipedia.org/wiki/Directed_acyclic_graph> (DAGs) in SQL based DBs. DAGs are almost trees, but with a twist: it may be possible to reach the same node through different paths. Trees are restricted from this possibility, which makes them much easier to handle. In our case it is "Ali" and "Engin" that brake the tree pattern, as they are both admins and users. Reality often looks this way and can't be captured by tree structures.

In the article an SQL + Stored Procedure solution is provided. The main idea, that also have some support from scientists, is to pre-calculate all possible (transitive) paths. Pros and cons of this approach:

- decent performance on read
- low performance on insert
- wastes *lots* of space
- relies on stored procedures

In Neo4j storing the roles is trivial. In this case we use `PART_OF` (blue edges) relationships to model the group hierarchy and `MEMBER_OF` (green edges) to model membership in groups. We also connect the top level groups to the reference node by `ROOT` relationships. This gives us a useful partitioning of the graph. Neo4j has no predefined relationship types, you are free to create any relationship types and give them any semantics you want.

Lets now have a look at how to retrieve information from the graph. The Java code is using the Neo4j Traversal API (Section 3.5, "Traversal"), the queries are done using the Chapter 16, *Cypher Query Language*.

7.1.1. Get the admins

```
Node admins = getNodeByName( "Admins" );
Traverser traverser = admins.traverse( Traverser.Order.BREADTH_FIRST,
        StopEvaluator.END_OF_GRAPH,
        ReturnableEvaluator.ALL_BUT_START_NODE, RoleRels.PART_OF,
        Direction.INCOMING, RoleRels.MEMBER_OF, Direction.INCOMING );
```

resulting in the output

```
Found: Ali at depth: 0
Found: HelpDesk at depth: 0
Found: Engin at depth: 1
Found: Demet at depth: 1
```

As Cypher, this looks like:

```
START admins=node(14)
MATCH admins<-[:PART_OF*0..]-subrole<-[:MEMBER_OF]-user
RETURN user, subrole
```

resulting in:

user	subrole
Node[15]{name->"Ali"}	Node[14]{name->"Admins"}
Node[8]{name->"Engin"}	Node[4]{name->"HelpDesk"}
Node[7]{name->"Demet"}	Node[4]{name->"HelpDesk"}
3 rows, 43 ms	

7.1.2. Get the group memberships of a user

Using the Neo4j Java Traversal API, this query looks like:

```
Node jale = getNodeByName( "Jale" );
traverser = jale.traverse( Traverser.Order.DEPTH_FIRST,
        StopEvaluator.END_OF_GRAPH,
        ReturnableEvaluator.ALL_BUT_START_NODE, RoleRels.MEMBER_OF,
        Direction.OUTGOING, RoleRels.PART_OF, Direction.OUTGOING );
```

resuling in:

```
Found: ABCTechnicians at depth: 0
Found: Technicians at depth: 1
Found: Users at depth: 2
```

In Cypher:

```
START jale=node(10)
MATCH jale-[:MEMBER_OF]->()-[:PART_OF*0..]->role
RETURN jale, role
```

jale	role
Node[10]{name->"Jale"}	Node[3]{name->"ABCTechnicians"}
Node[10]{name->"Jale"}	Node[6]{name->"Technicians"}
Node[10]{name->"Jale"}	Node[2]{name->"Users"}
3 rows, 3 ms	

7.1.3. Get all groups

In Java:

```
Node referenceNode = getNodeByName( "Reference_Node") ;
traverser = referenceNode.traverse(
        Traverser.Order.BREADTH_FIRST, StopEvaluator.END_OF_GRAPH,
        ReturnableEvaluator.ALL_BUT_START_NODE, RoleRels.ROOT,
        Direction.INCOMING, RoleRels.PART_OF, Direction.INCOMING );
```

resulting in:

```
Found: Admins at depth: 0
Found: Users at depth: 0
Found: HelpDesk at depth: 1
Found: Managers at depth: 1
Found: Technicians at depth: 1
Found: ABCTechnicians at depth: 2
```

In Cypher:

```
START refNode=node(16)
MATCH refNode<-[:ROOT]->()<-[:PART_OF*0..]-group
RETURN group
```

group
Node[14]{name->"Admins"}
Node[4]{name->"HelpDesk"}
Node[2]{name->"Users"}
Node[5]{name->"Managers"}
Node[6]{name->"Technicians"}
Node[3]{name->"ABCTechnicians"}
6 rows, 5 ms

7.1.4. Get all members of all groups

Now, let's try to find all users in the system being part of any group.

in Java:

```
traverser = referenceNode.traverse(
        Traverser.Order.BREADTH_FIRST,
        StopEvaluator.END_OF_GRAPH,
        new ReturnableEvaluator()
        {
            @Override
            public boolean isReturnableNode(
                    TraversalPosition currentPos )
            {
                if ( currentPos.isStartNode() )
                {
                    return false;
                }
                Relationship rel = currentPos.lastRelationshipTraversed();
                return rel.isType( RoleRels.MEMBER_OF );
            }
        }, RoleRels.ROOT, Direction.INCOMING, RoleRels.PART_OF,
        Direction.INCOMING, RoleRels.MEMBER_OF, Direction.INCOMING );
```

```
Found: Ali at depth: 1
```

```
Found: Engin at depth: 1
Found: Burcu at depth: 1
Found: Can at depth: 1
Found: Demet at depth: 2
Found: Gul at depth: 2
Found: Fuat at depth: 2
Found: Hakan at depth: 2
Found: Irmak at depth: 2
Found: Jale at depth: 3
```

In Cypher, this looks like:

```
START refNode=node(16)
MATCH p=refNode<-[:ROOT]->parent<-[:PART_OF*0..]-group, group<-[:MEMBER_OF]-user
RETURN group.name, user.name, LENGTH(p)
ORDER BY LENGTH(p)
```

and results in the following output, listing even duplicate pathes to users, see e.g. user `Engin`.

group.name	user.name	LENGTH(p)
Admins	Ali	1
Users	Ali	1
Users	Engin	1
Users	Burcu	1
Users	Can	1
HelpDesk	Engin	2
HelpDesk	Demet	2
Managers	Gul	2
Managers	Fuat	2
Technicians	Hakan	2
Technicians	Irmak	2
ABCTechnicians	Jale	3
12 rows, 27 ms		

As seen above, querying even more complex scenarios can be done using comparatively short constructs in Java and other query mechanisms.

Full source code: RolesTest.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/test/java/org/neo4j/examples/RolesTest.java>

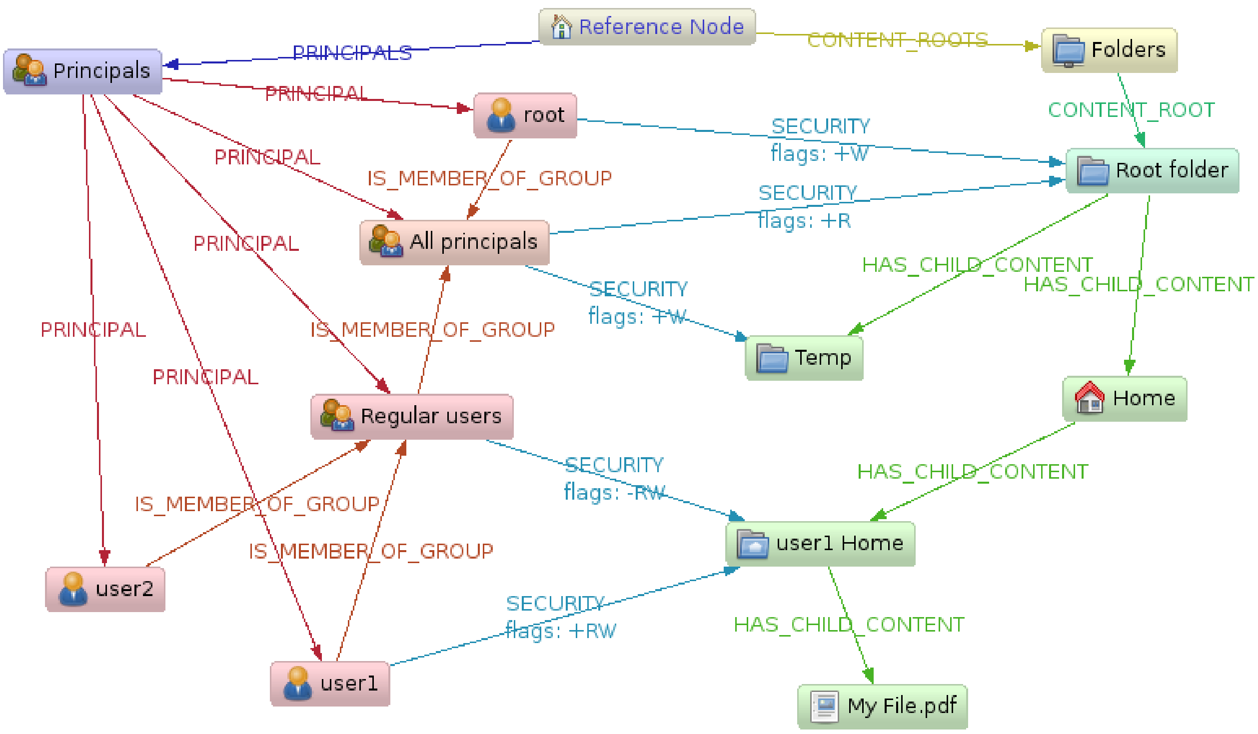
7.2. ACL structures in graphs

This example gives a generic overview of an approach to handling ACLs in graphs, and a simplified example with concrete queries.

7.2.1. Generic approach

In many scenarios, an application needs to handle security on some form of managed objects. This example describes one pattern to handle that through use a the graph structure and traversers that build a full permissions-structure for any managed object with exclude and include overriding possibilities. This results in a dynamic construction of Access Control Lists (ACLs) based on the position and context of the managed object.

The result is a complex security scheme that can easily be implemented in a graph structure, supporting permissions overriding, principal and content composition, and does not duplicate data anywhere.



Technique

As seen in the example graph layout, there are some key concepts in this domain model:

- The managed content (folders and files) that are connected by `HAS_CHILD_CONTENT` relationships
- The Principal subtree pointing out principals that can act as ACL members, pointed out by the `PRINCIPAL` relationships.
- The aggregation of principals into groups, connected by the `IS_MEMBER_OF` relationship. One principal (user or group) can be part of many groups at the same time.
- The `SECURITY` - relationships, connecting the content composite structure to the principal composite structure, containing a addition/removal modifier property ("`+RW`")

Constructing the ACL

The calculation of the effective permissions (e.g. Read, Write, Execute) for a principal for any given ACL-managed node (content) follows a number of rules that will be encoded into the permissions-traversal:

Top-down-Traversal

This approach will let you define a generic permission pattern on the root content, and then refine that for specific sub-content nodes and specific principals.

1. Start at the content node in question traverse upwards to the content root node to determine the path to it.
2. Start with a effective optimistic permissions list of "all permitted" (`111` in a bit encoded ReadWriteExecute case) or `000` if you like pessimistic security handling (everything is forbidden unless explicitly allowed).
3. Beginning from the topmost content node, look for any `SECURITY` relationships on it.
4. If found, look if the principal in question is part of the end-principal of the `SECURITY` relationship.
5. If yes, add the "+" permission modifiers to the existing permission pattern, revoke the "-" permission modifiers from the pattern.
6. If two principal nodes link to the same content node, first apply the more generic prinipals modifiers.
7. Repeat the security modifier search all the way down to the target content node, thus overriding more generic permissions with the set on nodes closer to the target node.

The same algorithm is applicable for the bottom-up approach, basically just traversing from the target content node upwards and applying the security modifiers dynamically as the traverser goes up.

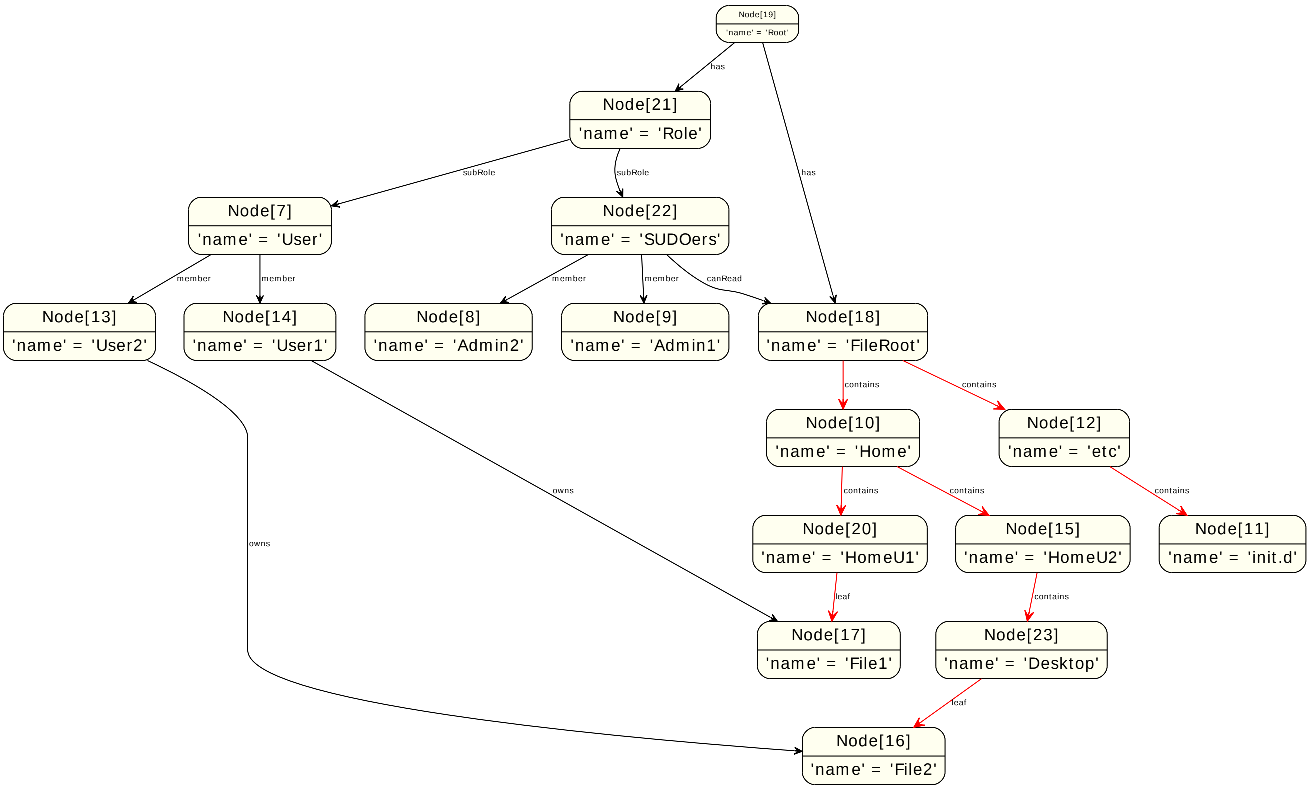
Example

Now, to get the resulting access rights for e.g. "`user 1`" on the "`My File.pdf`" in a Top-Down approach on the model in the graph above would go like:

1. Traveling upward, we start with "`Root folder`", and set the permissions to `11` initially (only considering Read, Write).
2. There are two `SECURITY` relationships to that folder. User 1 is contained in both of them, but "`root`" is more generic, so apply it first then "`All principals`" `+W` `+R` → `11`.
3. "`Home`" has no `SECURITY` instructions, continue.
4. "`user1 Home`" has `SECURITY`. First apply "`Regular Users`" (`-R` `-W`) → `00`, Then "`user 1`" (`+R` `+W`) → `11`.
5. The target node "`My File.pdf`" has no `SECURITY` modifiers on it, so the effective permissions for "`User 1`" on "`My File.pdf`" are `ReadWrite` → `11`.

7.2.2. Read-permission example

In this example, we are going to examine a tree structure of `directories` and `files`. Also, there are users that own files and roles that can be assigned to users. Roles can have permissions on directory or files structures (here we model only `canRead`, as opposed to full `rwx` Unix permissions) and be nested. A more thorough example of modeling ACL structures can be found at How to Build Role-Based Access Control in SQL <http://www.xaprb.com/blog/2006/08/16/how-to-build-role-based-access-control-in-sql/>.



Find all files in the directory structure

In order to find all files contained in this structure, we need a variable length query that follows all `contains` relationships and retrieves the nodes at the other end of the `leaf` relationships.

```
START root=node:node_auto_index(name = 'FileRoot')
MATCH root-[:contains*0..]->(parentDir)-[:leaf]->file
RETURN file
```

resulting in:

file
Node[11]{name->"File1"}
Node[10]{name->"File2"}
2 rows, 5 ms

What files are owned by whom?

If we introduce the concept of ownership on files, we then can ask for the owners of the files we find — connected via `owns` relationships to file nodes.

```
START root=node:node_auto_index(name = 'FileRoot')
MATCH root-[:contains*0..]->()-[:leaf]->file<-[:owns]-user
RETURN file, user
```

Returning the owners of all files below the `FileRoot` node.

file	user
Node[11]{name->"File1"}	Node[8]{name->"User1"}
2 rows, 5 ms	

file	user
Node[10]{name->"File2"}	Node[7]{name->"User2"}
2 rows, 5 ms	

Who has access to a File?

If we now want to check what users have read access to all Files, and define our ACL as

- The root directory has no access granted.
- Any user having a role that has been granted `canRead` access to one of the parent folders of a File has read access.

In order to find users that can read any part of the parent folder hierarchy above the files, Cypher provides optional variable length path.

```
START file=node:node_auto_index('name:File*')
MATCH file<-[:leaf]-()<-[:contains*0..]-dir<-[?:canRead]-role-[:member]->readUser
RETURN file.name, dir.name, role.name, readUser.name
```

This will return the `file`, and the directory where the user has the `canRead` permission along with the `user` and their `role`.

file.name	dir.name	role.name	readUser.name
File2	Desktop	<null>	<null>
File2	HomeU2	<null>	<null>
File2	Home	<null>	<null>
File2	FileRoot	SUDOers	Admin1
File2	FileRoot	SUDOers	Admin2
File1	HomeU1	<null>	<null>
File1	Home	<null>	<null>
File1	FileRoot	SUDOers	Admin1
File1	FileRoot	SUDOers	Admin2
9 rows, 48 ms			

The results listed above contain `null` values for optional path segments, which can be mitigated by either asking several queries or returning just the really needed values.

Full source code: AclExampleTest.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/test/java/org/neo4j/examples/AclExampleTest.java>

Chapter 8. Using the Neo4j REST API

8.1. How to use the REST API from Java

8.1.1. Creating a graph through the REST API from Java

The REST API uses HTTP and JSON, so that it can be used from many languages and platforms. Still, when geting started it's useful to see some patterns that can be re-used. In this brief overview, we'll show you how to create and manipulate a simple graph through the REST API and also how to query it.

For these examples, we've chosen the Jersey <http://jersey.java.net/> client components, which are easily downloaded <http://jersey.java.net/nonav/documentation/latest/user-guide.html#chapter_deps> via Maven.

8.1.2. Start the server

Before we can perform any actions on the server, we need to start it as per Section 17.1, "Server Installation".

```
WebResource resource = Client.create()
        .resource( SERVER_ROOT_URI );
ClientResponse response = resource.get( ClientResponse.class );

System.out.println( String.format( "GET on [%s], status code [%d]",
        SERVER_ROOT_URI, response.getStatus() ) );
response.close();
```

If the status of the response is `200 OK`, then we know the server is running fine and we can continue. If the code fails to conenct to the server, then please have a look at Chapter 17, *Neo4j Server*.

> **Note**
> If you get any other response than `200 OK` (particularly `4xx` or `5xx` responses) then please check your configuration and look in the log files in the *data/log* directory.

8.1.3. Creating a node

The REST API uses `POST` to create nodes. Encapsulating that in Java is straightforward using the Jersey client:

```
final String nodeEntryPointUri = SERVER_ROOT_URI + "node";
// http://localhost:7474/db/data/node

WebResource resource = Client.create()
        .resource( nodeEntryPointUri );
// POST {} to the node entry point URI
ClientResponse response = resource.accept( MediaType.APPLICATION_JSON )
        .type( MediaType.APPLICATION_JSON )
        .entity( "{}" )
        .post( ClientResponse.class );

final URI location = response.getLocation();
System.out.println( String.format(
        "POST to [%s], status code [%d], location header [%s]",
        nodeEntryPointUri, response.getStatus(), location.toString() ) );
response.close();

return location;
```

If the call completes successfully, under the covers it will have sent a HTTP request containing a JSON payload to the server. The server will then have created a new node in the database and responded with a `201 Created` response and a `Location` header with the URI of the newly created node.

In our example, we call this functionality twice to create two nodes in our database.

8.1.4. Adding properties

Once we have nodes in our datatabase, we can use them to store useful data. In this case, we're going to store information about music in our database. Let's start by looking at the code that we use to create nodes and add properties. Here we've added nodes to represent "Joe Strummer" and a band called "The Clash".

```
URI firstNode = createNode();
addProperty( firstNode, "name", "Joe Strummer" );
URI secondNode = createNode();
addProperty( secondNode, "band", "The Clash" );
```

Inside the `addProperty` method we determine the resource that represents properties for the node and decide on a name for that property. We then proceed to `PUT` the value of that property to the server.

```
String propertyUri = nodeUri.toString() + "/properties/" + propertyName;
// http://localhost:7474/db/data/node/{node_id}/properties/{property_name}

WebResource resource = Client.create()
        .resource( propertyUri );
ClientResponse response = resource.accept( MediaType.APPLICATION_JSON )
        .type( MediaType.APPLICATION_JSON )
        .entity( "\"" + propertyValue + "\"" )
        .put( ClientResponse.class );

System.out.println( String.format( "PUT to [%s], status code [%d]",
        propertyUri, response.getStatus() ) );
response.close();
```

If everything goes well, we'll get a `204 No Content` back indicating that the server processed the request but didn't echo back the property value.

8.1.5. Adding relationships

Now that we have nodes to represent Joe Strummer and The Clash, we can relate them. The REST API supports this through a `POST` of a relationship representation to the start node of the relationship. Correspondingly in Java we `POST` some JSON to the URI of our node that represents Joe Strummer, to establish a relationship between that node and the node representing The Clash.

```
URI relationshipUri = addRelationship( firstNode, secondNode, "singer",
        "{ \"from\" : \"1976\", \"until\" : \"1986\" }" );
```

Inside the `addRelationship` method, we determine the URI of the Joe Strummer node's relationships, and then `POST` a JSON description of our intended relationship. This description contains the destination node, a label for the relationship type, and any attributes for the relation as a JSON collection.

```
private static URI addRelationship( URI startNode, URI endNode,
        String relationshipType, String jsonAttributes )
        throws URISyntaxException
{
    URI fromUri = new URI( startNode.toString() + "/relationships" );
    String relationshipJson = generateJsonRelationship( endNode,
            relationshipType, jsonAttributes );

    WebResource resource = Client.create()
            .resource( fromUri );
    // POST JSON to the relationships URI
    ClientResponse response = resource.accept( MediaType.APPLICATION_JSON )
```

```
            .type( MediaType.APPLICATION_JSON )
            .entity( relationshipJson )
            .post( ClientResponse.class );

    final URI location = response.getLocation();
    System.out.println( String.format(
            "POST to [%s], status code [%d], location header [%s]",
            fromUri, response.getStatus(), location.toString() ) );

    response.close();
    return location;
}
```

If all goes well, we receive a `201 Created` status code and a `Location` header which contains a URI of the newly created relation.

8.1.6. Add properties to a relationship

Like nodes, relationships can have properties. Since we're big fans of both Joe Strummer and the Clash, we'll add a rating to the relationship so that others can see he's a 5-star singer with the band.

```
addMetadataToProperty( relationshipUri, "stars", "5" );
```

Inside the `addMetadataToProperty` method, we determine the URI of the properties of the relationship and `PUT` our new values (since it's `PUT` it will always overwrite existing values, so be careful).

```
private static void addMetadataToProperty( URI relationshipUri,
        String name, String value ) throws URISyntaxException
{
    URI propertyUri = new URI( relationshipUri.toString() + "/properties" );
    String entity = toJsonNameValuePairCollection( name, value );
    WebResource resource = Client.create()
            .resource( propertyUri );
    ClientResponse response = resource.accept( MediaType.APPLICATION_JSON )
            .type( MediaType.APPLICATION_JSON )
            .entity( entity )
            .put( ClientResponse.class );

    System.out.println( String.format(
            "PUT [%s] to [%s], status code [%d]", entity, propertyUri,
            response.getStatus() ) );
    response.close();
}
```

Assuming all goes well, we'll get a `200 OK` response back from the server (which we can check by calling `ClientResponse.getStatus()`) and we've now established a very small graph that we can query.

8.1.7. Querying graphs

As with the embedded version of the database, the Neo4j server uses graph traversals to look for data in graphs. Currently the Neo4j server expects a JSON payload describing the traversal to be `POST`-ed at the starting node for the traversal (though this is *likely to change* in time to a `GET`-based approach).

To start this process, we use a simple class that can turn itself into the equivalent JSON, ready for `POST`-ing to the server, and in this case we've hardcoded the traverser to look for all nodes with outgoing relationships with the type `"singer"`.

```
// TraversalDescription turns into JSON to send to the Server
TraversalDescription t = new TraversalDescription();
t.setOrder( TraversalDescription.DEPTH_FIRST );
t.setUniqueness( TraversalDescription.NODE );
```

```
t.setMaxDepth( 10 );
t.setReturnFilter( TraversalDescription.ALL );
t.setRelationships( new Relationship( "singer", Relationship.OUT ) );
```

Once we have defined the parameters of our traversal, we just need to transfer it. We do this by determining the URI of the traversers for the start node, and then `POST`-ing the JSON representation of the traverser to it.

```
URI traverserUri = new URI( startNode.toString() + "/traverse/node" );
WebResource resource = Client.create()
        .resource( traverserUri );
String jsonTraverserPayload = t.toJson();
ClientResponse response = resource.accept( MediaType.APPLICATION_JSON )
        .type( MediaType.APPLICATION_JSON )
        .entity( jsonTraverserPayload )
        .post( ClientResponse.class );

System.out.println( String.format(
        "POST [%s] to [%s], status code [%d], returned data: "
                + System.getProperty( "line.separator" ) + "%s",
        jsonTraverserPayload, traverserUri, response.getStatus(),
        response.getEntity( String.class ) ) );
response.close();
```

Once that request has completed, we get back our dataset of singers and the bands they belong to:


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/82/relationships/out",
  "data" : {
    "band" : "The Clash",
    "name" : "Joe Strummer"
  },
  "traverse" : "http://localhost:7474/db/data/node/82/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/82/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/82/properties/{key}",
  "all_relationships" : "http://localhost:7474/db/data/node/82/relationships/all",
  "self" : "http://localhost:7474/db/data/node/82",
  "properties" : "http://localhost:7474/db/data/node/82/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/82/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/82/relationships/in",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/82/relationships/in/{-list|&|types}",
  "create_relationship" : "http://localhost:7474/db/data/node/82/relationships"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/83/relationships/out",
  "data" : {
  },
  "traverse" : "http://localhost:7474/db/data/node/83/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/83/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/83/properties/{key}",
  "all_relationships" : "http://localhost:7474/db/data/node/83/relationships/all",
  "self" : "http://localhost:7474/db/data/node/83",
  "properties" : "http://localhost:7474/db/data/node/83/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/83/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/83/relationships/in",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/83/relationships/in/{-list|&|types}",
  "create_relationship" : "http://localhost:7474/db/data/node/83/relationships"
} ]
```


8.1.8. Phew, is that it?

That's a flavor of what we can do with the REST API. Naturally any of the HTTP idioms we provide on the server can be easily wrapped, including removing nodes and relationships through `DELETE`. Still if you've gotten this far, then switching `.post()` for `.delete()` in the Jersey client code should be straightforward.

8.1.9. What's next?

The HTTP API provides a good basis for implementers of client libraries, it's also great for HTTP and REST folks. In the future though we expect that idiomatic language bindings will appear to take advantage of the REST API while providing comfortable language-level constructs for developers to use, much as there are similar bindings for the embedded database. For a list of current Neo4j REST clients and embedded wrappers, see http://www.delicious.com/neo4j/drivers.

8.1.10. Appendix: the code

- CreateSimpleGraph.java <https://github.com/neo4j/community/blob/master/server-examples/src/main/java/org/neo4j/examples/server/CreateSimpleGraph.java>
- Relationship.java <https://github.com/neo4j/community/blob/master/server-examples/src/main/java/org/neo4j/examples/server/Relationship.java>
- TraversalDescription.java <https://github.com/neo4j/community/blob/master/server-examples/src/main/java/org/neo4j/examples/server/TraversalDescription.java>

Chapter 9. The Traversal Framework

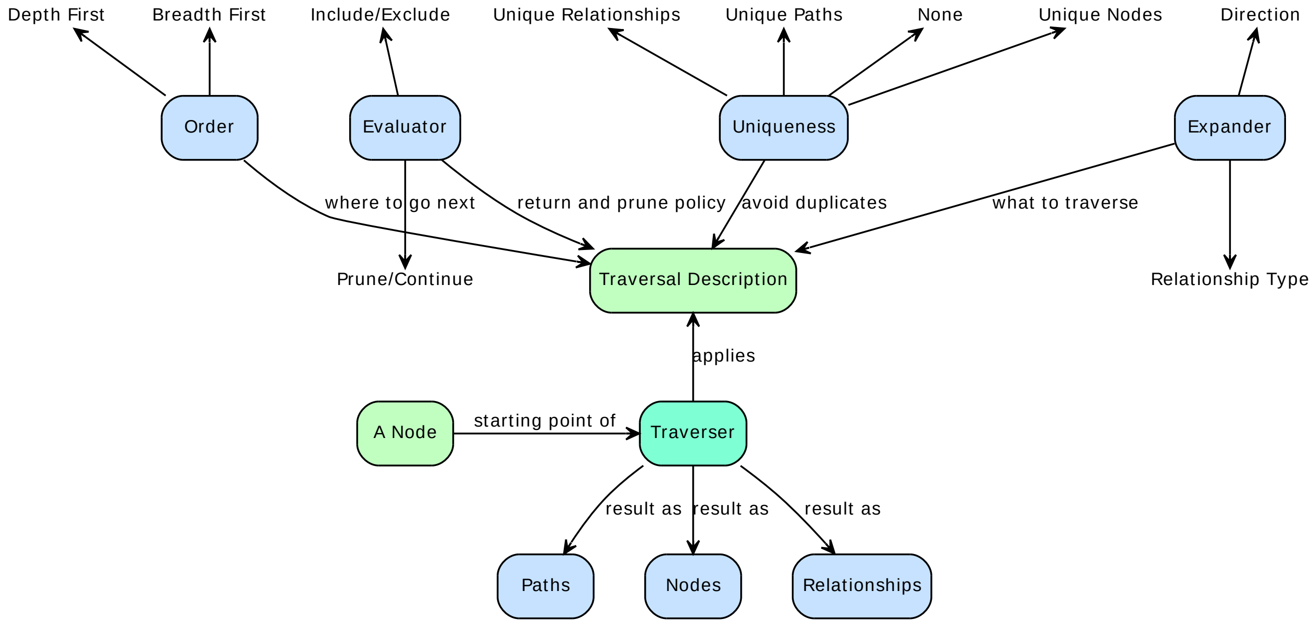
The Neo4j Traversal API <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/package-summary.html> is a callback based, lazily executed way of specifying desired movements through a graph in Java. Some traversal examples for both the new and the old traversal framework are collected under Section 4.4, “Traversal”.

Other options to traverse or query graphs in Neo4j are Cypher and Gremlin.

9.1. Main concepts

Here follows a short explanation of all different methods that can modify or add to a traversal description.

- *Expanders* — define what to traverse, typically in terms of relationships direction and type.
- *Order* — for example depth-first or breadth-first.
- *Uniqueness* — visit nodes (relationships, paths) only once.
- *Evaluator* — decide what to return and whether to stop or continue traversal beyond the current position.
- A *starting node* where the traversal will begin.



See Section 9.2, "Traversal Framework Java API" for more details.

9.2. Traversal Framework Java API

The traversal framework consists of a few main interfaces in addition to `Node` and `Relationship`: `TraversalDescription`, `Evaluator`, `Traverser` and `Uniqueness` are the main ones. The `Path` interface also has a special purpose in traversals, since it is used to represent a position in the graph when evaluating that position. Furthermore the `RelationshipExpander` and `Expander` interfaces are central to traversals, but users of the API rarely need to implement them. There are also a set of interfaces for advanced use, when explicit control over the traversal order is required: `BranchSelector`, `BranchOrderingPolicy` and `TraversalBranch`.

9.2.1. TraversalDescription

The `TraversalDescription` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/TraversalDescription.html> is the main interface used for defining and initializing traversals. It is not meant to be implemented by users of the traversal framework, but rather to be provided by the implementation of the traversal framework as a way for the user to describe traversals. `TraversalDescription` instances are immutable and its methods returns a new `TraversalDescription` that is modified compared to the object the method was invoked on with the arguments of the method.

Relationships

Adds a relationship type to the list of relationship types to traverse. By default that list is empty and it means that it will traverse *all relationships*, irregardless of type. If one or more relationships are added to this list *only the added* types will be traversed. There are two methods, one including direction <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/TraversalDescription.html#relationships> and another one excluding direction <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/TraversalDescription.html#relationships>, where the latter traverses relationships in both directions <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/Direction.html#BOTH>.

9.2.2. Evaluator

`Evaluator` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/Evaluator.html>s are used for deciding, at each position (represented as a `Path`): should the traversal continue, and/or should the node be included in the result. Given a `Path`, it asks for one of four actions for that branch of the traversal:

- `Evaluation.INCLUDE_AND_CONTINUE`: Include this node in the result and continue the traversal
- `Evaluation.INCLUDE_AND_PRUNE`: Include this node in the result, but don't continue the traversal
- `Evaluation.EXCLUDE_AND_CONTINUE`: Exclude this node from the result, but continue the traversal
- `Evaluation.EXCLUDE_AND_PRUNE`: Exclude this node from the result and don't continue the traversal

More than one evaluator can be added. Note that evaluators will be called for all positions the traverser encounters, even for the start node.

9.2.3. Traverser

The `Traverser` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/Traverser.html> object is the result of invoking `traverse()` <http://components.neo4j.org/neo4j-kernel/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/

TraversalDescription.html#traverse(org.neo4j.graphdb.Node)> of a TraversalDescription object. It represents a traversal positioned in the graph, and a specification of the format of the result. The actual traversal is performed lazily each time the `next()`-method of the iterator of the `Traverser` is invoked.

9.2.4. Uniqueness

Sets the rules for how positions can be revisited during a traversal as stated in `Uniqueness` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/Uniqueness.html>. Default if not set is `NODE_GLOBAL` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/Uniqueness.html#NODE_GLOBAL>.

A Uniqueness can be supplied to the TraversalDescription to dictate under what circumstances a traversal may revisit the same position in the graph. The various uniqueness levels that can be used in Neo4j are:

- `NONE` - any position in the graph may be revisited.
- `NODE_GLOBAL` uniqueness - no node in the entire graph may be visited more than once. This could potentially consume a lot of memory since it requires keeping an in-memory data structure remembering all the visited nodes.
- `RELATIONSHIP_GLOBAL` uniqueness - no relationship in the entire graph may be visited more than once. For the same reasons as NODE_GLOBAL uniqueness, this could use up a lot of memory. But since graphs typically have a larger number of relationships than nodes, the memory overhead of this uniqueness level could grow even quicker.
- `NODE_PATH` uniqueness - a node may not occur previously in the path reaching up to it.
- `RELATIONSHIP_PATH` uniqueness - a relationship may not occur previously in the path reaching up to it.
- `NODE_RECENT` uniqueness - Similar to NODE_GLOBAL uniqueness in that there is a global collection of visited nodes each position is checked against. This uniqueness level does however have a cap on how much memory it may consume in the form of a collection that only contains the most recently visited nodes. The size of this collection can be specified by providing a number as the second argument to the TraversalDescription.uniqueness()-method along with the uniqueness level.
- `RELATIONSHIP_RECENT` uniqueness - works like NODE_RECENT uniqueness, but with relationships instead of nodes.

Depth First / Breadth First

These are convenience methods for setting preorder depth-first <http://en.wikipedia.org/wiki/Depth-first_search>/ breadth-first <http://en.wikipedia.org/wiki/Breadth-first_search> `BranchSelector|ordering` policies. The same result can be achieved by calling the `order` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/TraversalDescription.html#order> method with ordering policies from the `Traversal` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/Traversal.html#preorderDepthFirst> `factory` <http://components.neo4j.org/neo4j-kernel/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/Traversal.html#preorderBreadthFirst>, or to write your own `BranchSelector/BranchOrderingPolicy` and pass in.

9.2.5. Order - How to move through branches?

A more generic version of depthFirst/breadthFirst methods in that it allows an arbitrary `BranchOrderingPolicy` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/BranchOrderingPolicy.html> to be injected into the description.

9.2.6. BranchSelector

A BranchSelector is used for selecting which branch of the traversal to attempt next. This is used for implementing traversal orderings. The traversal framework provides a few basic ordering implementations:

- `Traversal.preorderDepthFirst()` - Traversing depth first, visiting each node before visiting its child nodes.
- `Traversal.postorderDepthFirst()` - Traversing depth first, visiting each node after visiting its child nodes.
- `Traversal.preorderBreadthFirst()` - Traversing breadth first, visiting each node before visiting its child nodes.
- `Traversal.postorderBreadthFirst()` - Traversing breadth first, visiting each node after visiting its child nodes.

> **Note**
> Please note that breadth first traversals have a higher memory overhead than depth first traversals.

BranchSelectors carries state and hence needs to be uniquely instantiated for each traversal. Therefore it is supplied to the TraversalDescription through a BranchOrderingPolicy interface, which is a factory of BranchSelector instances.

A user of the Traversal framework rarely needs to implement his own BranchSelector or BranchOrderingPolicy, it is provided to let graph algorithm implementors provide their own traversal orders. The Neo4j Graph Algorithms package contains for example a BestFirst order BranchSelector/ BranchOrderingPolicy that is used in BestFirst search algorithms such as A* and Dijkstra.

BranchOrderingPolicy

A factory for creating BranchSelectors to decide in what order branches are returned (where a branch's position is represented as a `Path` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/Path.html> from the start node to the current node). Common policies are `depth-first` <http://components.neo4j.org/neo4j-kernel/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/TraversalDescription.html#depthFirst()> and `breadth-first` <http://components.neo4j.org/neo4j-kernel/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/TraversalDescription.html#breadthFirst()> and that's why there are convenience methods for those. For example, calling `TraversalDescription#depthFirst()` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/TraversalDescription.html#depthFirst()> is equivalent to:

```
description.order( Traversal.preorderDepthFirst() );
```

TraversalBranch

An object used by the BranchSelector to get more branches from a certain branch. In essence these are a composite of a Path and a RelationshipExpander that can be used to get new `TraversalBranch` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/TraversalBranch.html>es from the current one.

9.2.7. Path

A `Path` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/Path.html> is a general interface that is part of the Neo4j API. In the traversal API of Neo4j the use of Paths are twofold. Traversers can return their results in the form of the Paths of the visited positions in the graph that are marked for being returned. Path objects are also used in the evaluation of positions in the graph, for determining if the traversal should continue from a certain point or not, and whether a certain position should be included in the result set or not.

9.2.8. RelationshipExpander

The traversal framework use RelationshipExpanders to discover the relationships that should be followed from a particular node to further branches in the traversal.

9.2.9. Expander

A more generic version of relationships where a `RelationshipExpander` is injected, defining all relationships to be traversed for any given node. By default (and when using relationships) a default expander <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/Traversal.html#emptyExpander> is used, where any particular order of relationships isn't guaranteed. There's another implementation which guarantees that relationships are traversed in order of relationship type <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/OrderedByTypeExpander.html>, where types are iterated in the order they were added.

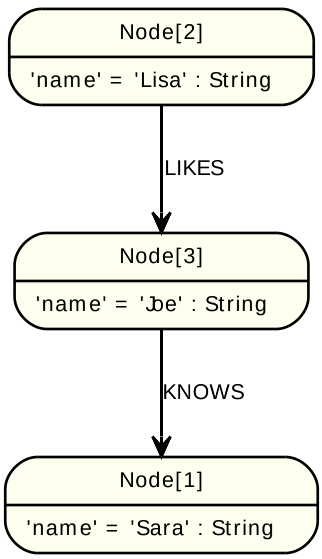
The `Expander` interface is an extension of the `RelationshipExpander` interface that makes it possible to build customized versions of an `Expander`. The implementation of `TraversalDescription` uses this to provide methods for defining which relationship types to traverse, this is the usual way a user of the API would define a `RelationshipExpander` - by building it internally in the `TraversalDescription`.

All the RelationshipExpanders provided by the Neo4j traversal framework also implement the Expander interface. For a user of the traversal API it is easier to implement the RelationshipExpander interface, since it only contains one method - the method for doing getting the relationships from a node, the methods that the Expander interface adds are just for building new Expanders.

9.2.10. How to use the Traversal framework

In contrary to Node#traverse <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/Node.html#traverse> a traversal description <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/TraversalDescription.html> is built (using a fluent interface) and such a description can spawn traversers <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/Traverser.html>.

Figure 9.1. Hello World Graph



With the definition of the `RelationshipTypes` as

```
private static enum Rels implements RelationshipType
{
    LIKES, KNOWS
}
```

The graph can be traversed with

```
for ( Path position : Traversal.description()
        .depthFirst()
        .relationships( Rels.KNOWS )
        .relationships( Rels.LIKES, Direction.INCOMING )
        .prune( Traversal.pruneAfterDepth( 5 ) )
        .traverse( joe ) )
{
  System.out.println( "Path from start node to current position is " + position );
}
```

Since `TraversalDescription` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/TraversalDescription.html>s are immutable it is also useful to create template descriptions which holds common settings shared by different traversals, for example:

```
final TraversalDescription FRIENDS_TRAVERSAL = Traversal.description()
        .relationships( Rels.KNOWS )
        .depthFirst()
        .uniqueness( Uniqueness.RELATIONSHIP_GLOBAL );

// Don't go further than depth 3
for ( Path position : FRIENDS_TRAVERSAL
          .prune( Traversal.pruneAfterDepth( 3 ) )
          .traverse( joe ) ) {}
// Don't go further than depth 4
for ( Path position : FRIENDS_TRAVERSAL
          .prune( Traversal.pruneAfterDepth( 4 ) )
          .traverse( joe ) ) {}
```

If you're not interested in the `Path` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/Path.html>s, but f.ex. the `Node` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/Node.html>s you can transform the traverser into an iterable of nodes <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/Traverser.html#nodes()> or relationships <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/traversal/Traverser.html#relationships()>:

```
for ( Node position : FRIENDS_TRAVERSAL.traverse( joe ).nodes() ) {}
```

The full source for this example is available at

TraversalTest.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/test/java/org/neo4j/examples/TraversalTest.java>

Part III. Reference

Chapter 10. Installation & Deployment

10.1. Deployment Scenarios

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

Neo4j deployment options

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

10.1.1. 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 17, *Neo4j Server*. For running the server and embedded installations in high availability mode, see Chapter 19, *High Availability*.

10.1.2. Embedded

Neo4j can be embedded directly in a server application by including the appropriate Java libraries. When programming, you can refer to the `GraphDatabaseService` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/GraphDatabaseService.html> API. To switch from a single instance to multiple highly available instances, simply switch from the concrete `EmbeddedGraphDatabase` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/EmbeddedGraphDatabase.html> to the `HighlyAvailableGraphDatabase` <http://components.neo4j.org/neo4j-enterprise/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/HighlyAvailableGraphDatabase.html>.

10.2. System Requirements

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

10.2.1. 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

10.2.2. Memory

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

Minimum
: 1GB

Recommended
: 4-8GB

10.2.3. Disk

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

Minimum
: SCSI, EIDE

Recommended
: SSD w/ SATA

10.2.4. Filesystem

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

Minimum
: ext3 (or similar)

Recommended
: ext4, ZFS

10.2.5. Software

Neo4j is Java-based.

Java
: 1.6+

Operating Systems
: Linux, Windows XP, Mac OS X

10.2.6. JDK Version

The Neo4j runtime is continuously tested with

- Oracle Java Runtime Environment JRE 1.6 <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

10.3. 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 17.1, “Server Installation”.

The following table outlines the available editions and their names for use with dependency management tools.

Tip
Follow the links in the table for details on dependency configuration with Apache Maven, Apache Buildr, Apache Ivy and Groovy Grape!

Neo4j editions

<table>
<tr><th>Edition</th><th>Dependency</th><th>Description</th><th>License</th></tr>
<tr><td>Community</td><td>org.neo4j:neo4j <http://search.maven.org/#search|gav|1|g%3A%22org.neo4j%22%20AND%20a%3A%22neo4j%22></td><td>a high performance, fully ACID transactional graph database</td><td>GPLv3</td></tr>
<tr><td>Advanced</td><td>org.neo4j:neo4j-advanced <http://search.maven.org/#search|gav|1|g%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-advanced%22></td><td>adding advanced monitoring</td><td>AGPLv3</td></tr>
<tr><td>Enterprise</td><td>org.neo4j:neo4j-enterprise <http://search.maven.org/#search|gav|1|g%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-enterprise%22></td><td>adding online backup and High Availability clustering</td><td>AGPLv3</td></tr>
</table>

Note
The listed dependencies do not contain the implementation, but pulls it in transitively.

For more information regarding licensing, see the Licensing Guide <http://neo4j.org/licensing-guide/>.

10.3.1. Embedded Installation

The latest release is always available from http://neo4j.org/download, included as part of the Neo4j download packages. 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]. Stable and milestone releases are available there.

For information on how to use Neo4j as a dependency with Maven and other dependency management tools, see the following table:

Note

The listed dependencies do not contain the implementation, but pulls it in transitively.

Maven dependency.

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

Where `${neo4j-version}` *is the intended version and the* `artifactId` *is one of* `neo4j`, `neo4j-advanced`, `neo4j-enterprise`.

10.3.2. Server Installation

OS X

Via homebrew <http://mxcl.github.com/homebrew/>, you can simply do

```
brew update
brew install neo4j
neo4j start
```

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

10.4. Upgrading

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.

For most upgrades, only small changes are required to the database store, and these changes proceed automatically when you start up the database using the newer version of Neo4J.

However, some upgrades require more significant changes to the database store. In these cases, Neo4j will refuse to start without explicit configuration to allow the upgrade.

The table below lists recent Neo4J versions, and the type of upgrade required.

Upgrade process for Neo4J version

From Version	To Version	Upgrade Type
1.3	1.4	Automatic
1.4	1.5	Explicit

10.4.1. Automatic 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

10.4.2. Explicit 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" in your `neo4j.properties` or embedded configuration
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

10.4.3. Upgrade 1.4 → 1.5

This upgrade includes a significant change to the layout of property store files, which reduces their size on disk, and improves IO performance. To achieve this layout change, the upgrade process takes some time to process the whole of the existing database. You should budget for several minutes per gigabyte of data as part of your upgrade planning.

In an HA environment these steps need to be performed:

1. shut down all the databases in the cluster
2. shut down the zoo keeper cluster and clear the `version-2` directories on all the zoo keeper instances
3. start the zoo keeper cluster again

4. remove the databases except the master and start the master database with 1.5, where migration will happen
5. start up the other databases so that they get a copy from the master

Warning

The upgrade process for this upgrade temporarily requires additional disk space, for the period while the upgrade is in progress. Before starting the upgrade to Neo4J 1.5, you should ensure that the machine performing the upgrade has free space equal to the current size of of the database on disk. You can find the current space occupied by the database by inspecting the store file directory (`data/graph.db` is the default location in Neo4J server). Once the upgrade is complete, this additional space is no longer required.

10.5. 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.

10.5.1. 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.

10.5.2. 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>
  <exclusions>
    <exclusion>
      <groupId>org.neo4j</groupId>
      <artifactId>neo4j-udc</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Where ${neo4j-version} is the Neo4j version in use.

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 11. 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.

11.1. Introduction

To gain good performance, these are the things to look into first:

- Make sure the JVM is not spending too much time performing garbage collection. Monitoring heap usage on an application that uses Neo4j can be a bit confusing since Neo4j will increase the size of caches if there is available memory and decrease if the heap is getting full. The goal is to have a large enough heap to make sure that heavy/peak load will not result in so called GC trashing (performance can drop as much as two orders of magnitude when GC trashing happens).
- Start the JVM with the -server flag and a good sized heap (see Section 11.4, “JVM Settings”). Having too large heap may also hurt performance so you may have to try some different heap sizes.
- Use the parallel/concurrent garbage collector (we found that `-XX:+UseConcMarkSweepGC` works well in most use-cases)

11.1.1. How to add configuration settings

When creating the embedded Neo4j instance it is possible to pass in parameters contained in a map where keys and values are strings.

```
Map<String, String> config = new HashMap<String, String>();
config.put( "neostore.nodestore.db.mapped_memory", "10M" );
config.put( "string_block_size", "60" );
config.put( "array_block_size", "300" );
EmbeddedGraphDatabase db = new EmbeddedGraphDatabase( "target/mydb", config  );
```

If no configuration is provided, the Database Kernel will try to determine suitable settings from the information available via the JVM settings and the underlying operating system.

The JVM is configured by passing command line flags when starting the JVM. The most important configuration parameters for Neo4j are the ones that control the memory and garbage collector, but some of the parameters for configuring the Just In Time compiler are also of interest.

This is an example of starting up your applications main class using 64-bit server VM mode and a heap space of 1GB:

```
java -d64 -server -Xmx1024m -cp /path/to/neo4j-kernel.jar:/path/to/jta.jar:/path/to/your-application.jar com.example.yourapp.MainCla
```

Looking at the example above you will also notice one of the most basic command line parameters: the one for specifying the classpath. The classpath is the path in which the JVM searches for your classes. It is usually a list of jar-files. Specifying the classpath is done by specifying the flag `-cp` (or `-classpath`) and then the value of the classpath. For Neo4j applications this should at least include the path to the Neo4j `neo4j-kernel.jar` and the Java Transaction API (`jta.jar`) as well as the path where the classes for your application are located.

Tip

On Linux, Unix and Mac OS X each element in the path list are separated by a colon symbol (`:`), on Windows the path elements are separated by a semicolon (`;`).

When using the Neo4j REST server, see Section 17.2, “Server Configuration” for how to add configuration settings for the database to the server.

11.2. Performance Guide

This is the Neo4j performance guide. It will attempt to guide you in how to use Neo4j to achieve maximum performance.

11.2.1. Try this first

The first thing is to make sure the JVM is running well and not spending to much time in garbage collection. Monitoring heap usage on an application that uses Neo4j can be a bit confusing since Neo4j will increase the size of caches if there is available memory and decrease if the heap is getting full. The goal is to have a large enough heap so heavy/peak load will not result in so called GC trashing (performance can drop as much as two orders of magnitude when this happens).

Start the JVM with `-server` flag and `-Xmx<good sized heap>` (f.ex. -Xmx512M for 512Mb memory or -Xmx3G for 3Gb memory). Having too large heap may also hurt performance so you may have to try some different heap sizes. Make sure parallel/concurrent garbage collector is running (`-XX:+UseConcMarkSweepGC` works well in most use-cases).

Finally make sure the OS has some memory to manage proper file system caches meaning if your server has 8GB of RAM don't use all of that RAM for heap (unless you turned off memory mapped buffers) but leave a good size of it to the OS. For more information on this see Chapter 11, *Configuration & Performance*.

For Linux specific tweaks, see Section 11.9, "Linux Performance Guide".

11.2.2. Neo4j primitives' lifecycle

Neo4j manages its primitives (nodes, relationships and properties) different depending on how you use Neo4j. For example if you never get a property from a certain node or relationship that node or relationship will not have its properties loaded into memory. Another example is a node that you never request any relationships from, that node will then not load any of its relationships into memory.

Nodes and relationships are cached using LRU caches. If you (for some strange reason) only work with nodes the relationship cache will become smaller and smaller while the node cache is allowed to grow (if needed). Working with many relationships and few nodes results in bigger relationship cache and smaller node cache.

The Neo4j API specification does not say anything about order regarding relationships so invoking

```
Node.getRelationships()
```

may return the relationships in a different order than the previous invocation. This allows us to make even heavier optimizations returning the relationships that are most commonly traversed.

All in all Neo4j has been designed to be very adaptive depending on how it is used. The (unachievable) overall goal is to be able to handle any incoming operation without having to go down and work with the file/disk I/O layer.

11.2.3. Configuring Neo4j

In Chapter 11, *Configuration & Performance* page there's information on how to configure Neo4j and the JVM. These settings have a lot impact on performance.

Disks, RAM and other tips

As always, as with any persistence solution, performance is very much depending on the persistence media used. Better disks equals better performance.

If you have multiple disks or persistence media available it may be a good idea to split the store files and transaction logs across those disks. Having the store files running on disks with low seek time can do wonders for non cached read operations. Today a typical mechanical drive has an average seek time of about 5ms, this can cause a query or traversal to be very slow when available RAM is too low or caches and memory mapped settings are badly configured. A new good SATA enabled SSD has an average seek time of <100 microseconds meaning those scenarios will execute at least 50 times faster.

To avoid hitting disk you need more RAM. On a standard mechanical drive you can handle graphs with a few tens of millions of primitives with 1-2GB of RAM. 4-8GB of RAM can handle graphs with hundreds of millions of primitives while you need a good server with 16-32GB to handle billions of primitives. However, if you invest in a good SSD you will be able to handle much larger graphs on less RAM.

Neo4j likes Java 1.6 JVMs and running in server mode so consider upgrading to that if you haven't yet (or at least give the -server flag). Use tools like `vmstat` or equivalent to gather info when your application is running. If you have high I/O waits and not that many blocks going out/in to disks when running write/read transactions its a sign that you need to tweak your Java heap, Neo4j cache and memory mapped settings (maybe even get more RAM or better disks).

Write performance

If you are experiencing poor write performance after writing some data (initially fast, then massive slowdown) it may be the operating system writing out dirty pages from the memory mapped regions of the store files. These regions do not need to be written out to maintain consistency so to achieve highest possible write speed that type of behavior should be avoided.

Another source of writes slow down can be the transaction size. Many small transactions result in a lot of I/O writes to disc and should be avoided. Too big transactions can result in OutOfMemory errors, since the uncommitted transaction data is held on the Java Heap in memory. On details about transaction management in Neo4j, please read the Chapter 13, *Transaction Management* guidelines.

The Neo4j kernel makes use of several store files and a logical log file to store the graph on disk. The store files contain the actual graph and the log contains mutating operations. All writes to the logical log are append-only and when a transaction is committed changes to the logical log will be forced (`fdatasync`) down to disk. The store files are however not flushed to disk and writes to them are not append-only either. They will be written to in a more or less random pattern (depending on graph layout) and writes will not be forced to disk until the log is rotated or the Neo4j kernel is shut down. It may be a good idea to increase the logical log target size for rotation or turn off log rotation if you experience problems with writes that can be linked to the actual rotation of the log. Here is some example code demonstrating how to change log rotation settings at runtime:

```
GraphDatabaseService graphDb; // ...

// get the XaDataSource for the native store
TxModule txModule = ((EmbeddedGraphDatabase) graphDb).getConfig().getTxModule();
XaDataSourceManager xaDsMgr = txModule.getXaDataSourceManager();
XaDataSource xaDs = xaDsMgr.getXaDataSource( "nioneodb" );

// turn off log rotation
xaDs.setAutoRotate( false );

// or to increase log target size to 100MB (default 10MB)
xaDs.setLogicalLogTargetSize( 100 * 1024 * 1024L );
```

Since random writes to memory mapped regions for the store files may happen it is very important that the data does not get written out to disk unless needed. Some operating systems have very aggressive settings regarding when to write out these dirty pages to disk. If the OS decides to start writing out dirty pages of these memory mapped regions, write access to disk will stop being sequential and become random. That hurts performance a lot, so to get maximum write performance when using Neo4j make sure the OS is configured not to write out any of the dirty pages caused by writes to the memory mapped regions of the store files. As an example, if the machine has 8GB of RAM and the total size of the store files is 4GB (fully memory mapped) the OS has to be configured to accept at least 50% dirty pages in virtual memory to make sure we do not get random disk writes.

Note: make sure to read the Section 11.9, “Linux Performance Guide” as well for more specific information.

Second level caching

While normally building applications and "always assume the graph is in memory", sometimes it is necessary to optimize certain performance critical sections. Neo4j adds a small overhead even if the node, relationship or property in question is cached when you compare to in memory data structures. If this becomes an issue use a profiler to find these hot spots and then add your own second-level caching. We believe second-level caching should be avoided to greatest extend possible since it will force you to take care of invalidation which sometimes can be hard. But when everything else fails you have to use it so here is an example of how it can be done.

We have some POJO that wrapps a node holding its state. In this particular POJO we’ve overridden the equals implementation.

```
public boolean equals( Object obj )
{
    return underlyingNode.getProperty( "some_property" ).equals( obj );
}

public int hashCode()
{
    return underlyingNode.getProperty( "some_property" ).hashCode();
}
```

This works fine in most scenarios but in this particular scenario many instances of that POJO is being worked with in nested loops adding/removing/getting/finding to collection classes. Profiling the applications will show that the equals implementation is being called many times and can be viewed as a hot spot. Adding second-level caching for the equals override will in this particular scenario increase performance.

```
private Object cachedProperty = null;

public boolean equals( Object obj )
{
   if ( cachedProperty == null )
   {
       cachedProperty = underlyingNode.getProperty( "some_property" );
   }
   return cachedProperty.equals( obj );
}

public int hashCode()
{
   if ( cachedPropety == null )
   {
       cachedProperty = underlyingNode.getProperty( "some_property" );
   }
   return cachedProperty.hashCode();
}
```

The problem now is that we need to invalidate the cached property whenever the `some_property` is changed (may not be a problem in this scenario since the state picked for equals and hash code computation often won’t change).

Tip

To sum up, avoid second-level caching if possible and only add it when you really need it.

11.3. Caches in Neo4j

For how to provide custom configuration to Neo4j, see Section 11.1, "Introduction".

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.

11.3.1. 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	`41 B`	Properties for nodes and relationships
stringstore	`128 B`	Values of string properties
arraystore	`128 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 8B 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 and arrays can be stored without using the string store or the array store respectively, see Section 11.6, "Compressed storage of short strings" and Section 11.7, "Compressed storage of short arrays".

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.

Important
Note that the block sizes can only be configured at store creation time.

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.
`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,

Parameter	Possible values	Effect
		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 8 bytes. This means that if the block size is 120, the size of the stored records will be 128 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., 8 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.

11.3.2. 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.

You can read about references and relevant JVM settings for Sun HotSpot here:

- Understanding soft/weak references <http://weblogs.java.net/blog/enicholas/archive/2006/05/understanding_w.html>
- How Hotspot Decides to Clear SoftReferences <http://jeremymanson.blogspot.com/2009/07/how-hotspot-decides-to-clear_07.html>
- HotSpot FAQ <http://java.sun.com/docs/hotspot/HotSpotFAQ.html#gc_softrefs>

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).*
	`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.*

Object	**Size**	**Comment**
	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).*

11.4. JVM Settings

There are two main memory parameters for the JVM, one controls the heap space and the other controls the stack space. The heap space parameter is the most important one for Neo4j, since this governs how many objects you can allocate. The stack space parameter governs the how deep the call stack of your application is allowed to get.

When it comes to heap space the general rule is: the larger heap space you have the better, but make sure the heap fits in the RAM memory of the computer. If the heap is paged out to disk performance will degrade rapidly. Having a heap that is much larger than what your application needs is not good either, since this means that the JVM will accumulate a lot of dead objects before the garbage collector is executed, this leads to long garbage collection pauses and undesired performance behavior.

Having a larger heap space will mean that Neo4j can handle larger transactions and more concurrent transactions. A large heap space will also make Neo4j run faster since it means Neo4j can fit a larger portion of the graph in its caches, meaning that the nodes and relationships your application uses frequently are always available quickly. The default heap size for a 32bit JVM is 64MB (and 30% larger for 64bit), which is too small for most real applications.

Neo4j works fine with the default stack space configuration, but if your application implements some recursive behavior it is a good idea to increment the stack size. Note that the stack size is shared for all threads, so if you application is running a lot of concurrent threads it is a good idea to increase the stack size.

- The heap size is set by specifying the `-Xmx???m` parameter to hotspot, where `???` is the heap size in megabytes. Default heap size is 64MB for 32bit JVMs, 30% larger (appr. 83MB) for 64bit JVMs.
- The stack size is set by specifying the `-Xss???m` parameter to hotspot, where `???` is the stack size in megabytes. Default stack size is 512kB for 32bit JVMs on Solaris, 320kB for 32bit JVMs on Linux (and Windows), and 1024kB for 64bit JVMs.

Most modern CPUs implement a Non-Uniform Memory Access (NUMA) architecture <http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access>, where different parts of the memory have different access speeds. Suns Hotspot JVM is able to allocate objects with awareness of the NUMA structure as of version 1.6.0 update 18. When enabled this can give up to 40% performance improvements. To enabled the NUMA awareness, specify the `-XX:+UseNUMA` parameter (works only when using the Parallel Scavenger garbage collector (default or `-XX:+UseParallelGC` not the concurrent mark and sweep one).

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.

11.4.1. 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.

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

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

Tip
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.

When having made sure that the heap size is well configured the second thing to tune in order to tune the garbage collector for your application is to specify the sizes of the different generations of the heap. The default settings are well tuned for "normal" applications, and work quite well for most applications, but if you have an application with either really high allocation rate, or a lot of long lived objects you might want to consider tuning the sizes of the heap generation. The ratio between the young and tenured generation of the heap is specified by using the `-XX:NewRatio=#` command line option (where `#` is replaced by a number). The default ratio is 1:12 for client mode JVM, and 1:8 for server mode JVM. You can also specify the size of the young generation explicitly using the `-Xmn` command line option, which works just like the `-Xmx` option that specifies the total heap space.

GC shortname	Generation	Command line parameter	Comment
Copy	Young	`-XX:+UseSerialGC`	The Copying collector
MarkSweepCompact	Tenured	`-XX:+UseSerialGC`	The Mark and Sweep Compactor
ConcurrentMarkSweep	Tenured	`-XX:+UseConcMarkSweepGC`	The Concurrent Mark and Sweep Compactor
ParNew	Young	`-XX:+UseParNewGC`	The parallel Young Generation Collector — can only

GC shortname	Generation	Command line parameter	Comment
			be used with the Concurrent mark and sweep compactor.
PS Scavenge	Young	`-XX:+UseParallelGC`	The parallel object scavenger
PS MarkSweep	Tenured	`-XX:+UseParallelGC`	The parallel mark and sweep collector

These are the default configurations on some platforms according to our non-exhaustive research:

JVM	-d32 -client	-d32 -server	-d64 -client	-d64 -server
Mac OS X Snow Leopard, 64-bit, Hotspot 1.6.0_17	`ParNew` and `ConcurrentMarkSweep`	`PS Scavenge` and `PS MarkSweep`	`ParNew` and `ConcurrentMarkSweep`	`PS Scavenge` and `PS MarkSweep`
Ubuntu, 32-bit, Hotspot 1.6.0_16	`Copy` and `MarkSweepCompact`	`Copy` and `MarkSweepCompact`	N/A	N/A

11.5. File system tuning for high IO

In order to support the high IO load of small transactions from a database, the underlying file system should be tuned. Symptoms for this are low CPU load with high iowait. In this case, there are a couple of tweaks possible on Linux systems:

- Disable access-time updates: `noatime,nodiratime` flags for disk mount command or in the */etc/fstab* for the database disk volume mount.
- Tune the IO scheduler for high disk IO on the database disk.

11.6. Compressed storage of short strings

Neo4j will try to classify your strings in a short string class and if it manages that it will treat it accordingly. In that case, it will be stored without indirection in the property store, inlining it instead in the property record, meaning that the dynamic string store will not be involved in storing that value, leading to reduced disk footprint. Additionally, when no string record is needed to store the property, it can be read and written in a single lookup, leading to performance improvements.

The various classes for short strings are:

- Numerical, consisting of digits 0..9 and the punctuation space, period, dash, plus, comma and apostrophe.
- Date, consisting of digits 0..9 and the punctuation space dash, colon, slash, plus and comma.
- Uppercase, consisting of uppercase letters A..Z, and the punctuation space, underscore, period, dash, colon and slash.
- Lowercase, like upper but with lowercase letters a..z instead of uppercase
- E-mail, consisting of lowercase letters a..z and the punctuation comma, underscore, period, dash, plus and the at sign (@)
- URI, consisting of lowercase letters a..z, digits 0..9 and most punctuation available.
- Alphanumerical, consisting of both upper and lowercase letters a..zA..z, digits 0..9 and punctuation space and underscore.
- Alphasymbolical, consisting of both upper and lowercase letters a..zA..Z and the punctuation space, underscore, period, dash, colon, slash, plus, comma, apostrophe, at sign, pipe and semicolon.
- European, consisting of most accented european characters and digits plus punctuation space, dash, underscore and period - like latin1 but with less punctuation
- Latin 1
- UTF-8

In addition to the string's contents, the number of characters also determines if the string can be inlined or not. Each class has its own character count limits, which are

- For Numerical and Date, 54
- For Uppercase, Lowercase and E-mail, 43
- For URI, Alphanumerical and Alphasymbolical, 36
- For European, 31
- For Latin1, 27
- For UTF-8, 14

That means that the largest inline-able string is 54 characters long and must be of the Numerical class and also that all Strings of size 14 or less will always be inlined.

Also note that the above limits are for the default 41 byte PropertyRecord layout - if that parameter is changed via editing the source and recompiling, the above have to be recalculated.

11.7. Compressed storage of short arrays

Neo4j will try to store your primitive arrays in a compressed way, so as to save disk space and possibly an I/O operation. To do that, it employs a "bit-shaving" algorithm that tries to reduce the number of bits required for storing the members of the array. In particular:

1. For each member of the array, it determines the position of leftmost set bit.
2. Determines the largest such position among all members of the array
3. It reduces all members to that number of bits
4. Stores those values, prefixed by a small header.

That means that when even a single negative value is included in the array then the natural size of the primitives will be used.

There is a possibility that the result can be inlined in the property record if:

- It is less than 24 bytes after compression
- It has less than 64 members

For example, an array long[] {0L, 1L, 2L, 4L} will be inlined, as the largest entry (4) will require 3 bits to store so the whole array will be stored in 4*3=12 bits. The array long[] {-1L, 1L, 2L, 4L} however will require the whole 64 bits for the -1 entry so it needs 64*4 = 32 bytes and it will end up in the dynamic store.

11.8. Memory mapped IO settings

Each file in the Neo4j store can use memory mapped I/O for reading/writing. Best performance is achieved if the full file can be memory mapped but if there isn't enough memory for that Neo4j will try and make the best use of the memory it gets (regions of the file that get accessed often will more likely be memory mapped).

Important

Neo4j makes heavy use of the `java.nio` package. Native I/O will result in memory being allocated outside the normal Java heap so that memory usage needs to be taken into consideration. Other processes running on the OS will impact the availability of such memory. Neo4j will require all of the heap memory of the JVM plus the memory to be used for memory mapping to be available as physical memory. Other processes may thus not use more than what is available after the configured memory allocation is made for Neo4j.

A well configured OS with large disk caches will help a lot once we get cache misses in the node and relationship caches. Therefore it is not a good idea to use all available memory as Java heap.

If you look into the directory of your Neo4j database, you will find its store files, all prefixed by `neostore`:

- `nodestore` stores information about nodes
- `relationshipstore` holds all the relationships
- `propertystore` stores information of properties and all simple properties such as primitive types (both for relationships and nodes)
- `propertystore strings` stores all string properties
- `propertystore arrays` stores all array properties

There are other files there as well, but they are normally not interesting in this context.

This is how the default memory mapping configuration looks:

```
neostore.nodestore.db.mapped_memory=25M
neostore.relationshipstore.db.mapped_memory=50M
neostore.propertystore.db.mapped_memory=90M
neostore.propertystore.db.strings.mapped_memory=130M
neostore.propertystore.db.arrays.mapped_memory=130M
```

11.8.1. Optimizing for traversal speed example

To tune the memory mapping settings start by investigating the size of the different store files found in the directory of your Neo4j database. Here is an example of some of the files and sizes in a Neo4j database:

```
14M neostore.nodestore.db
510M neostore.propertystore.db
1.2G neostore.propertystore.db.strings
304M neostore.relationshipstore.db
```

In this example the application is running on a machine with 4GB of RAM. We've reserved about 2GB for the OS and other programs. The Java heap is set to 1.5GB, that leaves about 500MB of RAM that can be used for memory mapping.

Tip

If traversal speed is the highest priority it is good to memory map as much as possible of the node- and relationship stores.

An example configuration on the example machine focusing on traversal speed would then look something like:

```
neostore.nodestore.db.mapped_memory=15M
neostore.relationshipstore.db.mapped_memory=285M
neostore.propertystore.db.mapped_memory=100M
neostore.propertystore.db.strings.mapped_memory=100M
neostore.propertystore.db.arrays.mapped_memory=0M
```

11.8.2. Batch insert example

The configuration should suit the data set you are about to inject using BatchInsert. Lets say we have a random-like graph with 10M nodes and 100M relationships. Each node (and maybe some relationships) have different properties of string and Java primitive types (but no arrays). The important thing with a random graph will be to give lots of memory to the relationship and node store:

```
neostore.nodestore.db.mapped_memory=90M
neostore.relationshipstore.db.mapped_memory=3G
neostore.propertystore.db.mapped_memory=50M
neostore.propertystore.db.strings.mapped_memory=100M
neostore.propertystore.db.arrays.mapped_memory=0M
```

The configuration above will fit the entire graph (with exception to properties) in memory.

A rough formula to calculate the memory needed for the nodes:

```
number_of_nodes * 9 bytes
```

and for relationships:

```
number_of_relationships * 33 bytes
```

Properties will typically only be injected once and never read so a few megabytes for the property store and string store is usually enough. If you have very large strings or arrays you may want to increase the amount of memory assigned to the string and array store files.

An important thing to remember is that the above configuration will need a Java heap of 3.3G+ since in batch inserter mode normal Java buffers that gets allocated on the heap will be used instead of memory mapped ones.

11.9. Linux Performance Guide

The key to achieve good performance on reads and writes is to have lots of RAM since disks are so slow. This guide will focus on achieving good write performance on a Linux kernel based operating system.

If you have not already read the information available in Chapter 11, *Configuration & Performance* do that now to get some basic knowledge on memory mapping and store files with Neo4j.

This section will guide you through how to set up a file system benchmark and use it to configure your system in a better way.

11.9.1. Setup

Create a large file with random data. The file should fit in RAM so if your machine has 4GB of RAM a 1-2GB file with random data will be enough. After the file has been created we will read the file sequentially a few times to make sure it is cached.

```
$ dd if=/dev/urandom of=store bs=1M count=1000
1000+0 records in
1000+0 records out
1048576000 bytes (1.0 GB) copied, 263.53 s, 4.0 MB/s
$
$ dd if=store of=/dev/null bs=100M
10+0 records in
10+0 records out
1048576000 bytes (1.0 GB) copied, 38.6809 s, 27.1 MB/s
$
$ dd if=store of=/dev/null bs=100M
10+0 records in
10+0 records out
1048576000 bytes (1.0 GB) copied, 1.52365 s, 688 MB/s
$ dd if=store of=/dev/null bs=100M
10+0 records in
10+0 records out
1048576000 bytes (1.0 GB) copied, 0.776044 s, 1.4 GB/s
```

If you have a standard hard drive in the machine you may know that it is not capable of transfer speeds as high as 1.4GB/s. What is measured is how fast we can read a file that is cached for us by the operating system.

Next we will use a small utility that simulates the Neo4j kernel behavior to benchmark write speed of the system.

```
$ git clone git@github.com:neo4j/tooling.git
...
$ cd tooling/write-test/
$ mvn compile
[INFO] Scanning for projects...
...
$ ./run
Usage: <large file> <log file> <[record size] [min tx size] [max tx size] [tx count] <[--nosync | --nowritelog | --nowritestore | --n
```

The utility will be given a store file (large file we just created) and a name of a log file. Then a record size in bytes, min tx size, max tx size and transaction count must be set. When started the utility will map the large store file entirely in memory and read (transaction size) records from it randomly and then write them sequentially to the log file. The log file will then force changes to disk and finally the records will be written back to the store file.

11.9.2. Running the benchmark

Lets try to benchmark 100 transactions of size 100-500 with a record size of 33 bytes (same record size used by the relationship store).

```
$ ./run store logfile 33 100 500 100
tx_count[100] records[30759] fdatasyncs[100] read[0.96802425 MB] wrote[1.9360485 MB]
Time was: 4.973
20.108585 tx/s, 6185.2 records/s, 20.108585 fdatasyncs/s, 199.32773 kB/s on reads, 398.65546 kB/s on writes
```

We see that we get about 6185 record updates/s and 20 transactions/s with the current transaction size. We can change the transaction size to be bigger, for example writing 10 transactions of size 1000-5000 records:

```
$ ./run store logfile 33 1000 5000 10
tx_count[10] records[24511] fdatasyncs[10] read[0.77139187 MB] wrote[1.5427837 MB]
Time was: 0.792
12.626263 tx/s, 30948.232 records/s, 12.626263 fdatasyncs/s, 997.35516 kB/s on reads, 1994.7103 kB/s on writes
```

With larger transaction we will do fewer of them per second but record throughput will increase. Lets see if it scales, 10 transactions in under 1s then 100 of them should execute in about 10s:

```
$ ./run store logfile 33 1000 5000 100
tx_count[100] records[308814] fdatasyncs[100] read[9.718763 MB] wrote[19.437527 MB]
Time was: 65.115
1.5357445 tx/s, 4742.594 records/s, 1.5357445 fdatasyncs/s, 152.83751 kB/s on reads, 305.67502 kB/s on writes
```

This is not very linear scaling. We modified a bit more than 10x records in total but the time jumped up almost 100x. Running the benchmark watching vmstat output will reveal that something is not as it should be:

```
$ vmstat 3
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 0  1  47660 298884 136036 2650324    0    0     0 10239 1167 2268  5  7 46 42
 0  1  47660 302728 136044 2646060    0    0     0  7389 1267 2627  6  7 47 40
 0  1  47660 302408 136044 2646024    0    0     0 11707 1861 2016  8  5 48 39
 0  2  47660 302472 136060 2646432    0    0     0 10011 1704 1878  4  7 49 40
 0  1  47660 303420 136068 2645788    0    0     0 13807 1406 1601  4  5 44 47
```

There are a lot of blocks going out to IO, way more than expected for the write speed we are seeing in the benchmark. Another observation that can be made is that the Linux kernel has spawned a process called "flush-x:x" (run top) that seems to be consuming a lot of resources.

The problem here is that the Linux kernel is trying to be smart and write out dirty pages from the virtual memory. As the benchmark will memory map a 1GB file and do random writes it is likely that this will result in 1/4 of the memory pages available on the system to be marked as dirty. The Neo4j kernel is not sending any system calls to the Linux kernel to write out these pages to disk however the Linux kernel decided to start doing so and it is a very bad decision. The result is that instead of doing sequential like writes down to disk (the logical log file) we are now doing random writes writing regions of the memory mapped file to disk.

It is possible to observe this behavior in more detail by looking at /proc/vmstat "nr_dirty" and "nr_writeback" values. By default the Linux kernel will start writing out pages at a very low ratio of dirty pages (10%).

```
$ sync
$ watch grep -A 1 dirty /proc/vmstat
...
nr_dirty 22
nr_writeback 0
```

The "sync" command will write out all data (that needs writing) from memory to disk. The second command will watch the "nr_dirty" and "nr_writeback" count from vmstat. Now start the benchmark again and observe the numbers:

```
nr_dirty 124947
nr_writeback 232
```

The "nr_dirty" pages will quickly start to rise and after a while the "nr_writeback" will also increase meaning the Linux kernel is scheduling a lot of pages to write out to disk.

11.9.3. Fixing the problem

As we have 4GB RAM on the machine and memory map a 1GB file that does not need its content written to disk (until we tell it to do so because of logical log rotation or Neo4j kernel shutdown) it should be possible to do endless random writes to that memory with high throughput. All we have to do is to tell the Linux kernel to stop trying to be smart. Edit the /etc/sysctl.conf (need root access) and add the following lines:

```
vm.dirty_background_ratio = 50
vm.dirty_ratio = 80
```

Then (as root) execute:

```
# sysctl -p
```

The "vm.dirty_background_ratio" tells at what ratio should the linux kernel start the background task of writing out dirty pages. We increased this from the default 10% to 50% and that should cover the 1GB memory mapped file. The "vm.dirty_ratio" tells at what ratio all IO writes become synchronous, meaning that we can not do IO calls without waiting for the underlying device to complete them (which is something you never want to happen).

Rerun the benchmark:

```
$ ./run store logfile 33 1000 5000 100
tx_count[100] records[265624] fdatasyncs[100] read[8.35952 MB] wrote[16.71904 MB]
Time was: 6.781
14.7470875 tx/s, 39171.805 records/s, 14.7470875 fdatasyncs/s, 1262.3726 kB/s on reads, 2524.745 kB/s on writes
```

Results are now more in line with what can be expected, 10x more records modified results in 10x longer execution time. The vmstat utility will not report any absurd amount of IO blocks going out (it reports the ones caused by the fdatasync to the logical log) and Linux kernel will not spawn a "flush-x:x" background process writing out dirty pages caused by writes to the memory mapped store file.

Chapter 12. Capabilities

12.1. Data Security

Some data may need to be protected from unauthorized access (e.g., theft, modification). Neo4j does not deal with data encryption explicitly, but supports all means built into the Java programming language and the JVM to protect data by encrypting it before storing.

Furthermore, data can be easily secured by running on an encrypted datastore at the file system level. Finally, data protection should be considered in the upper layers of the surrounding system in order to prevent problems with scraping, malicious data insertion, and other threats.

12.2. Data Integrity

In order to keep data consistent, there needs to be mechanisms and structures that guarantee the integrity of all stored data. In Neo4j, data integrity is maintained for the core graph engine together with other data sources - see below.

12.2.1. Core Graph Engine

In Neo4j, the whole data model is stored as a graph on disk and persisted as part of every committed transaction. In the storage layer, Relationships, Nodes, and Properties have direct pointers to each other. This maintains integrity without the need for data duplication between the different backend store files.

12.2.2. Different Data Sources

In a number of scenarios, the core graph engine is combined with other systems in order to achieve optimal performance for non-graph lookups. For example, Apache Lucene is frequently used as an additional index system for text queries that would otherwise be very processing-intensive in the graph layer.

To keep these external systems in synchronization with each other, Neo4j provides full Two Phase Commit transaction management, with rollback support over all data sources. Thus, failed index insertions into Lucene can be transparently rolled back in all data sources and thus keep data up-to-date.

12.3. Data Integration

Most enterprises rely primarily on relational databases to store their data, but this may cause performance limitations. In some of these cases, Neo4j can be used as an extension to supplement search/lookup for faster decision making. However, in any situation where multiple data repositories contain the same data, synchronization can be an issue.

In some applications, it is acceptable for the search platform to be slightly out of sync with the relational database. In others, tight data integrity (eg., between Neo4j and RDBMS) is necessary. Typically, this has to be addressed for data changing in real-time and for bulk data changes happening in the RDBMS.

A few strategies for synchronizing integrated data follows.

12.3.1. Event-based Synchronization

In this scenario, all data stores, both RDBMS and Neo4j, are fed with domain-specific events via an event bus. Thus, the data held in the different backends is not actually synchronized but rather replicated.

12.3.2. Periodic Synchronization

Another viable scenario is the periodic export of the latest changes in the RDBMS to Neo4j via some form of SQL query. This allows a small amount of latency in the synchronization, but has the advantage of using the RDBMS as the master for all data purposes. The same process can be applied with Neo4j as the master data source.

12.3.3. Periodic Full Export/Import of Data

Using the Batch Inserter tools for Neo4j, even large amounts of data can be imported into the database in very short times. Thus, a full export from the RDBMS and import into Neo4j becomes possible. If the propagation lag between the RDBMS and Neo4j is not a big issue, this is a very viable solution.

12.4. Availability and Reliability

Most mission-critical systems require the database subsystem to be accessible at all times. Neo4j ensures availability and reliability through a few different strategies.

12.4.1. Operational Availability

In order not to create a single point of failure, Neo4j supports different approaches which provide transparent fallback and/or recovery from failures.

Online backup (Cold spare)

In this approach, a single instance of the master database is used, with Online Backup enabled. In case of a failure, the backup files can be mounted onto a new Neo4j instance and reintegrated into the application.

Online Backup High Availability (Hot spare)

Here, a Neo4j "backup" instance listens to online transfers of changes from the master. In the event of a failure of the master, the backup is already running and can directly take over the load.

High Availability cluster

This approach uses a cluster of database instances, with one (read/write) master and a number of (read-only) slaves. Failing slaves can simply be restarted and brought back online. Alternatively, a new slave may be added by cloning an existing one. Should the master instance fail, a new master will be elected by the remaining cluster nodes.

12.4.2. Disaster Recovery/ Resiliency

In cases of a breakdown of major part of the IT infrastructure, there need to be mechanisms in place that enable the fast recovery and regrouping of the remaining services and servers. In Neo4j, there are different components that are suitable to be part of a disaster recovery strategy.

Prevention

- Online Backup High Availability to other locations outside the current data center.
- Online Backup to different file system locations: this is a simpler form of backup, applying changes directly to backup files; it is thus more suited for local backup scenarios.
- Neo4j High Availability cluster: a cluster of one write-master Neo4j server and a number of read-slaves, getting transaction logs from the master. Write-master failover is handled by quorum election among the read-slaves for a new master.

Detection

- SNMP and JMX monitoring can be used for the Neo4j database.

Correction

- Online Backup: A new Neo4j server can be started directly on the backed-up files and take over new requests.
- Neo4j High Availability cluster: A broken Neo4j read slave can be reinserted into the cluster, getting the latest updates from the master. Alternatively, a new server can be inserted by copying an existing server and applying the latest updates to it.

12.5. Capacity

12.5.1. File Sizes

Neo4j relies on Java's Non-blocking I/O subsystem for all file handling. Furthermore, while the storage file layout is optimized for interconnected data, Neo4j does not require raw devices. Thus, filesizes are only limited by the underlying operating system's capacity to handle large files. Physically, there is no built-in limit of the file handling capacity in Neo4j.

Neo4j tries to memory-map as much of the underlying store files as possible. If the available RAM is not sufficient to keep all data in RAM, Neo4j will use buffers in some cases, reallocating the memory-mapped high-performance I/O windows to the regions with the most I/O activity dynamically. Thus, ACID speed degrades gracefully as RAM becomes the limiting factor.

12.5.2. Read speed

Enterprises want to optimize the use of hardware to deliver the maximum business value from available resources. Neo4j's approach to reading data provides the best possible usage of all available hardware resources. Neo4j does not block or lock any read operations; thus, there is no danger for deadlocks in read operations and no need for read transactions. With a threaded read access to the database, queries can be run simultaneously on as many processors as may be available. This provides very good scale-up scenarios with bigger servers.

12.5.3. Write speed

Write speed is a consideration for many enterprise applications. However, there are two different scenarios:

1. sustained continuous operation and
2. bulk access (e.g., backup, initial or batch loading).

To support the disparate requirements of these scenarios, Neo4j supports two modes of writing to the storage layer.

In transactional, ACID-compliant normal operation, isolation level is maintained and read operations can occur at the same time as the writing process. At every commit, the data is persisted to disk and can be recovered to a consistent state upon system failures. This requires disk write access and a real flushing of data. Thus, the write speed of Neo4j on a single server in continuous mode is limited by the I/O capacity of the hardware. Consequently, the use of fast SSDs is highly recommended for production scenarios.

Neo4j has a Batch Inserter that operates directly on the store files. This mode does not provide transactional security, so it can only be used when there is a single write thread. Because data is written sequentially, and never flushed to the logical logs, huge performance boosts are achieved. The Batch Inserter is optimized for non-transactional bulk import of large amounts of data.

12.5.4. Data size

In Neo4j, data size is mainly limited by the address space of the primary keys for Nodes, Relationships, and Properties. Currently, the address space is as follows:

- 2^35 (~ 34 billion) nodes
- 2^35 (~ 34 billion) relationships
- 2^36 (~ 68 billion) properties

Chapter 13. 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.

13.1. 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.

13.2. 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.

13.3. 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.

13.4. 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.

13.5. 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 14. 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_1_0/index.html> being the default backend included with Neo4j.

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

14.1. Introduction

Indexing operations are part of the Neo4j index API <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/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/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/Node.html> or relationships <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/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.

14.2. 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 indexes 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 indexes and one relationship index with default configuration. See Section 14.8, "Relationship indexes" for more information specific to relationship indexes.

See Section 14.10, "Configuration and fulltext indexes" for how to create *fulltext* indexes.

You can also check if an index exists like this:

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

14.3. Delete

Indexes 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/1.5-SNAPSHOT/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.

14.4. 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 indexes:

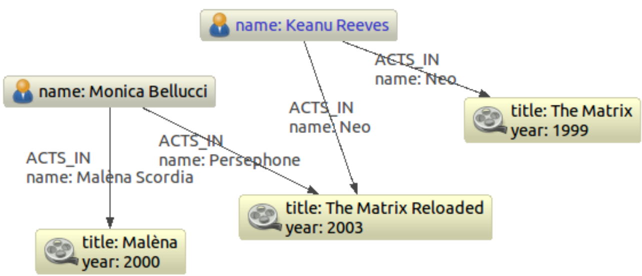
```
// Actors
Node reeves = graphDb.createNode();
actors.add( reeves, "name", "Keanu Reeves" );
Node bellucci = graphDb.createNode();
actors.add( bellucci, "name", "Monica Bellucci" );
// multiple values for a field
actors.add( bellucci, "name", "La Bellucci" );
// Movies
Node theMatrix = graphDb.createNode();
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:



14.5. Remove

Removing <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/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" );
```

14.6. 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" ) );
```

14.7. Search

An index can be searched in two ways, get <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/index/Index.html#get%28java.lang.String,%20java.lang.Object%29> and query <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/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_1_0/queryparsersyntax.html> can be used directly with the default indexing backend.

14.7.1. 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/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/index/IndexHits.html> is an `Iterable` with some additional useful methods. For example getSingle() <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/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/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/index/IndexHits.html#close%28%29> must be called explicitly.

14.7.2. 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 14.11.3, "Querying with Lucene Query objects" for how to do that.

14.8. Relationship indexes

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/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/index/RelationshipIndex.html> interface which extends Index<Relationship> <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/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.

14.9. Scores

The `IndexHits` interface exposes scoring <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/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 14.11.2, “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() );
}
```

14.10. Configuration and fulltext indexes

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( IndexManager.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.

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_1_0/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`.
`analyzer`	the full class name of an Analyzer <http://lucene.apache.org/java/3_1_0/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.

14.11. Extra features for Lucene indexes

14.11.1. 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/1.5-SNAPSHOT/apidocs/org/neo4j/index/lucene/ValueContext.html> class, like this:

```
movies.add( theMatrix, "year-numeric", new ValueContext( 1999 ).indexNumeric() );
movies.add( theMatrixReloaded, "year-numeric", new ValueContext( 2003 ).indexNumeric() );
movies.add( malena, "year-numeric",  new ValueContext( 2000 ).indexNumeric() );

int from = 1997;
int to = 1999;
hits = movies.query( QueryContext.numericRange( "year-numeric", from, to ) );
```

Note

The same type must be used for indexing and querying. That is, you can't index a value as a Long and then query the index using an Integer.

By giving `null` as from/to argument, an open ended query is created. In the following example we are doing that, and have added sorting to the query as well:

```
hits = movies.query(
        QueryContext.numericRange( "year-numeric", from, null )
          .sortNumeric( "year-numeric", false ) );
```

From/to in the ranges defaults to be *inclusive*, but you can change this behavior by using two extra parameters:

```
movies.add( theMatrix, "score", new ValueContext( 8.7 ).indexNumeric() );
movies.add( theMatrixReloaded, "score", new ValueContext( 7.1 ).indexNumeric() );
movies.add( malena, "score", new ValueContext( 7.4 ).indexNumeric() );

// include 8.0, exclude 9.0
hits = movies.query( QueryContext.numericRange( "score", 8.0, 9.0, true, false ) );
```

14.11.2. 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/1.5-SNAPSHOT/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)
}
```

14.11.3. 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_1_0/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.

14.11.4. 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.

14.11.5. 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/1.5-SNAPSHOT/apidocs/org/neo4j/index/lucene/QueryContext.html> class:

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

14.11.6. Caching

If your index lookups becomes a performance bottle neck, caching can be enabled for certain keys in certain indexes (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.

14.12. 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/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/index/BatchInserterIndex.html> which are provided via BatchInserterIndexProvider <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/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 the changes visible for reading, use this sparsely, requires IO!
actors.flush();

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

The configuration parameters are the same as mentioned in Section 14.10, “Configuration and fulltext indexes”.

14.12.1. Best practices

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

- Try to avoid flushing <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/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/1.5-SNAPSHOT/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).

Note

Changes to the index are available for reading first after they are flushed to disk. Thus, for optimal performance, read and lookup operations should be kept to a minimum during batchinsertion since they involve IO and impact speed negatively.

14.13. Automatic Indexing

Neo4j provides a single index for nodes and one for relationships in each database that automatically follow property values as they are added, deleted and changed on database primitives. This functionality is called *auto indexing* and is controlled both from the database configuration Map and through its own API.

Caution

This is an experimental feature. Expect changes in the API and do not rely on it for production data handling.

14.13.1. Configuration

By default Auto Indexing is off for both Nodes and Relationships. To enable it on database startup set the configuration options `Config.NODE_AUTO_INDEXING` and `Config.RELATIONSHIP_AUTO_INDEXING` to the string "`true`".

If you just enable auto indexing as above, then still *no* property will be auto indexed. To define which property names you want the auto indexer to monitor as a configuration parameter, set the `Config.{NODE,RELATIONSHIP}_KEYS_INDEXABLE` option to a String that is a comma separated concatenation of the property names you want auto indexed.

```
/*
 * Creating the configuration, adding nodeProp1 and nodeProp2 as
 * auto indexed properties for Nodes and relProp1 and relProp2 as
 * auto indexed properties for Relationships. Only those will be
 * indexed. We also have to enable auto indexing for both these
 * primitives explicitly.
 */
Map<String, String> config = new HashMap<String, String>();
config.put( Config.NODE_KEYS_INDEXABLE, "nodeProp1, nodeProp2" );
config.put( Config.RELATIONSHIP_KEYS_INDEXABLE, "relProp1, relProp2" );
config.put( Config.NODE_AUTO_INDEXING, "true" );
config.put( Config.RELATIONSHIP_AUTO_INDEXING, "true" );

EmbeddedGraphDatabase graphDb = new EmbeddedGraphDatabase(
        getStoreDir( "testConfig" ), config );

Transaction tx = graphDb.beginTx();
Node node1 = null, node2 = null;
Relationship rel = null;
try
{
    // Create the primitives
    node1 = graphDb.createNode();
    node2 = graphDb.createNode();
    rel = node1.createRelationshipTo( node2,
            DynamicRelationshipType.withName( "DYNAMIC" ) );

    // Add indexable and non-indexable properties
    node1.setProperty( "nodeProp1", "nodeProp1Value" );
    node2.setProperty( "nodeProp2", "nodeProp2Value" );
    node1.setProperty( "nonIndexed", "nodeProp2NonIndexedValue" );
    rel.setProperty( "relProp1", "relProp1Value" );
    rel.setProperty( "relPropNonIndexed", "relPropValueNonIndexed" );

    // Make things persistent
    tx.success();
}
catch ( Exception e )
```

```
{
    tx.failure();
}
finally
{
    tx.finish();
}
```

14.13.2. Search

The usefulness of the auto indexing functionality comes of course from the ability to actually query the index and retrieve results. To that end, you can acquire a `ReadableIndex` object from the `AutoIndexer` that exposes all the query and get methods of a full `Index` <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/index/Index.html> with exactly the same functionality. Continuing from the previous example, accessing the index is done like this:

```
// Get the Node auto index
ReadableIndex<Node> autoNodeIndex = graphDb.index().getNodeAutoIndexer().getAutoIndex();
// node1 and node2 both had auto indexed properties, get them
assertEquals( node1,
        autoNodeIndex.get( "nodeProp1", "nodeProp1Value" ).getSingle() );
assertEquals( node2,
        autoNodeIndex.get( "nodeProp2", "nodeProp2Value" ).getSingle() );
// node2 also had a property that should be ignored.
assertFalse( autoNodeIndex.get( "nonIndexed",
        "nodeProp2NonIndexedValue" ).hasNext() );

// Get the relationship auto index
ReadableIndex<Relationship> autoRelIndex = graphDb.index().getRelationshipAutoIndexer().getAutoIndex();
// One property was set for auto indexing
assertEquals( rel,
        autoRelIndex.get( "relProp1", "relProp1Value" ).getSingle() );
// The rest should be ignored
assertFalse( autoRelIndex.get( "relPropNonIndexed",
        "relPropValueNonIndexed" ).hasNext() );
```

14.13.3. Runtime Configuration

The same options that are available during database creation via the configuration can also be set during runtime via the `AutoIndexer` API.

Gaining access to the `AutoIndexer` API and adding two `Node` and one +Relationship properties to auto index is done like so:

```
// Start without any configuration
EmbeddedGraphDatabase graphDb = new EmbeddedGraphDatabase(
        getStoreDir( "testAPI" ) );

// Get the Node AutoIndexer, set nodeProp1 and nodeProp2 as auto
// indexed.
AutoIndexer<Node> nodeAutoIndexer = graphDb.index().getNodeAutoIndexer();
nodeAutoIndexer.startAutoIndexingProperty( "nodeProp1" );
nodeAutoIndexer.startAutoIndexingProperty( "nodeProp2" );

// Get the Relationship AutoIndexer
AutoIndexer<Relationship> relAutoIndexer = graphDb.index().getRelationshipAutoIndexer();
relAutoIndexer.startAutoIndexingProperty( "relProp1" );

// None of the AutoIndexers are enabled so far. Do that now
nodeAutoIndexer.setEnabled( true );
relAutoIndexer.setEnabled( true );
```

Parameters to the AutoIndexers passed through the Configuration and settings made through the API are cumulative. So you can set some beforehand known settings, do runtime checks to augment the initial configuration and then enable the desired auto indexers - the final configuration is the same regardless of the method used to reach it.

14.13.4. Updating the Automatic Index

Updates to the auto indexed properties happen of course automatically as you update them. Removal of properties from the auto index happens for two reasons. One is that you actually removed the property. The other is that you stopped autoindexing on a property. When the latter happens, any primitive you touch and it has that property, it is removed from the auto index, regardless of any operations on the property. When you start or stop auto indexing on a property, no auto update operation happens currently. If you need to change the set of auto indexed properties and have them re-indexed, you currently have to do this by hand. An example will illustrate the above better:

```
/*
 * Creating the configuration
 */
Map<String, String> config = new HashMap<String, String>();
config.put( Config.NODE_KEYS_INDEXABLE, "nodeProp1, nodeProp2" );
config.put( Config.NODE_AUTO_INDEXING, "true" );

EmbeddedGraphDatabase graphDb = new EmbeddedGraphDatabase(
        getStoreDir( "mutations" ), config );

Transaction tx = graphDb.beginTx();
Node node1 = null, node2 = null, node3 = null, node4 = null;
try
{
    // Create the primitives
    node1 = graphDb.createNode();
    node2 = graphDb.createNode();
    node3 = graphDb.createNode();
    node4 = graphDb.createNode();

    // Add indexable and non-indexable properties
    node1.setProperty( "nodeProp1", "nodeProp1Value" );
    node2.setProperty( "nodeProp2", "nodeProp2Value" );
    node3.setProperty( "nodeProp1", "nodeProp3Value" );
    node4.setProperty( "nodeProp2", "nodeProp4Value" );

    // Make things persistent
    tx.success();
}
catch ( Exception e )
{
    tx.failure();
}
finally
{
    tx.finish();
}

/*
 *  Here both nodes are indexed. To demonstrate removal, we stop
 *  autoindexing nodeProp1.
 */
AutoIndexer<Node> nodeAutoIndexer = graphDb.index().getNodeAutoIndexer();
nodeAutoIndexer.stopAutoIndexingProperty( "nodeProp1" );

tx = graphDb.beginTx();
```

```
try
{
    /*
     * nodeProp1 is no longer auto indexed. It will be
     * removed regardless. Note that node3 will remain.
     */
    node1.setProperty( "nodeProp1", "nodeProp1Value2" );
    /*
     * node2 will be auto updated
     */
    node2.setProperty( "nodeProp2", "nodeProp2Value2" );
    /*
     * remove node4 property nodeProp2 from index.
     */
    node4.removeProperty( "nodeProp2" );
    // Make things persistent
    tx.success();
}
catch ( Exception e )
{
    tx.failure();
}
finally
{
    tx.finish();
}

// Verify
ReadableIndex<Node> nodeAutoIndex = nodeAutoIndexer.getAutoIndex();
// node1 is completely gone
assertFalse( nodeAutoIndex.get( "nodeProp1", "nodeProp1Value" ).hasNext() );
assertFalse( nodeAutoIndex.get( "nodeProp1", "nodeProp1Value2" ).hasNext() );
// node2 is updated
assertFalse( nodeAutoIndex.get( "nodeProp2", "nodeProp2Value" ).hasNext() );
assertEquals( node2,
        nodeAutoIndex.get( "nodeProp2", "nodeProp2Value2" ).getSingle() );
/*
 * node3 is still there, despite its nodeProp1 property not being monitored
 * any more because it was not touched, in contrast with node1.
 */
assertEquals( node3,
        nodeAutoIndex.get( "nodeProp1", "nodeProp3Value" ).getSingle() );
// Finally, node4 is removed because the property was removed.
assertFalse( nodeAutoIndex.get( "nodeProp2", "nodeProp4Value" ).hasNext() );
```

Caution

If you start the database with auto indexing enabled but different auto indexed properties than the last run, then already auto-indexed entities will be deleted as you work with them. Make sure that the monitored set is what you want before enabling the functionality.

Chapter 15. 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*.

15.1. 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/1.5-SNAPSHOT/apidocs/org/neo4j/graphalgo/package-summary.html>
- Download <http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-graph-algo%22>
- Source code <https://github.com/neo4j/community/tree/master/graph-algo>

For information on how to use neo4j-graph-algo as a dependency with Maven and other dependency management tools, see `org.neo4j:neo4j-graph-algo` <http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-graph-algo%22> Note that it should be used with the same version of `org.neo4j:neo4j-kernel` <http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-kernel%22>. 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` <http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j%22%20AND%20a%3A%22neo4j%22> 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/1.5-SNAPSHOT/apidocs/org/neo4j/graphalgo/GraphAlgoFactory.html>.

For examples, see Section 4.6, “Graph Algorithm examples” (embedded database) and Section 18.11, “Built-in Graph Algorithms” (REST API).

Chapter 16. Cypher Query Language

A new query language, code-named “Cypher”, has been added to Neo4j. It allows for expressive and efficient querying of the graph store without having to write traversers in code. Cypher is still growing and maturing, and that means that there probably will be breaking syntax changes. It also means that it has not undergone the same rigorous performance testing as the other components.

Cypher is designed to be a humane query language, suitable for both developers and (importantly, we think) operations professionals who want to make ad-hoc queries on the database. Its constructs are based on English prose and neat iconography, which helps to make it (somewhat) self-explanatory.

Cypher is inspired by a number of different approaches and builds upon established practices for expressive querying. Most of the keywords like `WHERE` and `ORDER BY` are inspired by SQL <http://en.wikipedia.org/wiki/SQL>. Pattern matching borrows expression approaches from SPARQL <http://en.wikipedia.org/wiki/SPARQL>. Regular expression matching is implemented using the Scala programming language <http://www.scala-lang.org/>.

Cypher is a declarative language. It focuses on the clarity of expressing *what* to retrieve from a graph, not *how* to do it, in contrast to imperative languages like Java, and scripting languages like Gremlin <http://gremlin.tinkerpop.com> (supported via the Section 18.14, “Gremlin Plugin”) and the JRuby Neo4j bindings <http://neo4j.rubyforge.org/>. This makes the concern of how to optimize queries in implementation detail not exposed to the user.

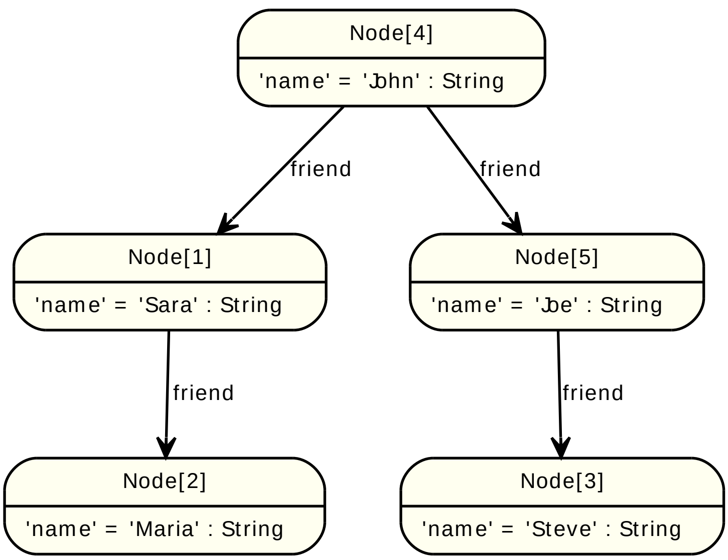
The query language is comprised of several distinct parts.

- `START`: Starting points in the graph, obtained by element IDs or via index lookups
- `MATCH`: The graph pattern to match, bound to the starting points in `START`
- `WHERE`: Filtering criteria
- `RETURN`: What to return

Let’s see three of them in action:

Imagine an example graph like

Figure 16.1. Example Graph



For example, here is a query which finds a user called John in an index and then traverses the graph looking for friends of Johns friends (though not his direct friends) before returning both John and any friends-of-friends that are found.

```
START john=node:node_auto_index(name = 'John')
MATCH john-[:friend]->()-[:friend]->fof
RETURN john, fof
```

Resulting in

john	fof
Node[4]{name->"John"}	Node[2]{name->"Maria"}
Node[4]{name->"John"}	Node[3]{name->"Steve"}
2 rows, 2 ms	

Next up we will add filtering to set all four parts in motion:

In this next example, we take a list of users (by node ID) and traverse the graph looking for those other users that have an outgoing `friend` relationship, returning only those followed users who have a `name` property starting with `S`.

```
START user=node(5,4,1,2,3)
MATCH user-[:friend]->follower
WHERE follower.name =~ /S.*/
RETURN user, follower.name
```

Resulting in

user	follower.name
Node[5]{name->"Joe"}	Steve
Node[4]{name->"John"}	Sara
2 rows, 1 ms	

In Java, using the query language looks something like this:

```
ExecutionEngine engine = new ExecutionEngine( db );
ExecutionResult result = engine.execute( "start n=node(0) where 1=1 return n" );

assertThat( result.columns(), hasItem( "n" ) );
Iterator<Node> n_column = result.columnAs( "n" );
assertThat( asIterable( n_column ), hasItem( db.getNodeById( 0 ) ) );
assertThat( result.toString(), containsString( "Node[0]" ) );
```

16.1. Parameters

Cypher supports querying with parameters. This allows developers to not to have to do string building to create a query, and it also makes caching of execution plans much easier for Cypher.

Parameters can be used for literals in the `WHERE` clause, for the index key and index value in the `START` clause, index queries, and finally for node/relationship ids.

Accepted names for parameter are letters and number, and any combination of these.

Here follows a few examples of how you can use parameters from Java.

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "id", 0 );
ExecutionResult result = engine.execute( "start n=node({id}) return n.name", params );
```

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "id", 0 );
ExecutionResult result = engine.execute( "start n=node({id}) return n.name", params );
Map<String, Object> params = new HashMap<String, Object>();
params.put( "node", andreasNode );
ExecutionResult result = engine.execute( "start n=node({node}) return n.name", params );
```

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "id", Arrays.asList( 0, 1, 2 ) );
ExecutionResult result = engine.execute( "start n=node({id}) return n.name", params );
```

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "name", "Johan" );
ExecutionResult result = engine.execute( "start n=node(0,1,2) where n.name = {name} return n", params );
```

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "key", "name" );
params.put( "value", "Michaela" );
ExecutionResult result = engine.execute( "start n=node:people({key} = {value}) return n", params );
```

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "query", "name:Andreas" );
ExecutionResult result = engine.execute( "start n=node:people({query}) return n", params );
```

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "s", 1 );
params.put( "l", 1 );
ExecutionResult result = engine.execute( "start n=node(0,1,2) return n.name skip {s} limit {l}", params );
```

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "regex", ".*h.*" );
ExecutionResult result = engine.execute( "start n=node(0,1,2) where n.name =~ {regex} return n.name", params );
```

16.2. Identifiers

When you reference parts of the pattern, you do so by naming them.

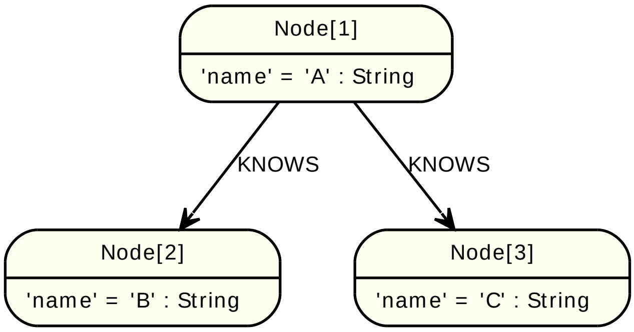
Identifiers can be lower or upper case, and may contain underscore. If other characters are needed, you can use the ` sign. The same rules apply to property names.

```
"start n=node(0) return n.NOT_EXISTING"
```

16.3. Start

Every query describes a pattern, and in that pattern one can have multiple bound points. A bound point is a relationship or a node that form the starting points for a pattern match. You can either bind points by id, or by index lookups.

Graph



16.3.1. Node by id

Binding a node as a start point is done with the node(*) function .

Query

```
START n=node(1)
RETURN n
```

The reference node is returned

Result

n
Node[1]{name->"A"}
1 rows, 0 ms

16.3.2. Relationship by id

Binding a relationship as a start point is done with the relationship() **function, which can also be abbreviated rel**().

Query

```
START r=relationship(0)
RETURN r
```

The relationshop with id 0 is returned

Result

r
:KNOWS[0] {}
1 rows, 0 ms

16.3.3. Multiple nodes by id

Multiple nodes are selected by listing them separated by commas.

Query

```
START n=node(1, 2, 3)
RETURN n
```

The nodes listed in the START statement.

Result

n
Node[1]{name->"A"}
Node[2]{name->"B"}
Node[3]{name->"C"}
3 rows, 0 ms

16.3.4. Node by index lookup

If the start point can be found by index lookups, it can be done like this: node:index-name(key = "value"). In this example, there exists a node index named *nodes*.

Query

```
START n=node:nodes(name = "A")
RETURN n
```

The node indexed with name "A" is returned

Result

n
Node[1]{name->"A"}
1 rows, 0 ms

16.3.5. Relationship by index lookup

If the start point can be found by index lookups, it can be done like this: relationship:index-name(key = "value"].

Query

```
START r=relationship:rels(property = "some_value")
RETURN r
```

The relationship indexed with property "some_value" is returned

Result

r
:KNOWS[0] {property->"some_value"}
1 rows, 0 ms

16.3.6. Node by index query

If the start point can be found by index more complex lucene queries: node:index-name("query").This allows you to write more advanced index queries

Query

```
START n=node:nodes("name:A")
RETURN n
```

The node indexed with name "A" is returned

Result

n
Node[1]{name->"A"}
1 rows, 1 ms

16.3.7. Multiple start points

Sometimes you want to bind multiple start points. Just list them separated by commas.

Query

```
START a=node(1), b=node(2)
RETURN a,b
```

Both the A and the B node are returned

Result

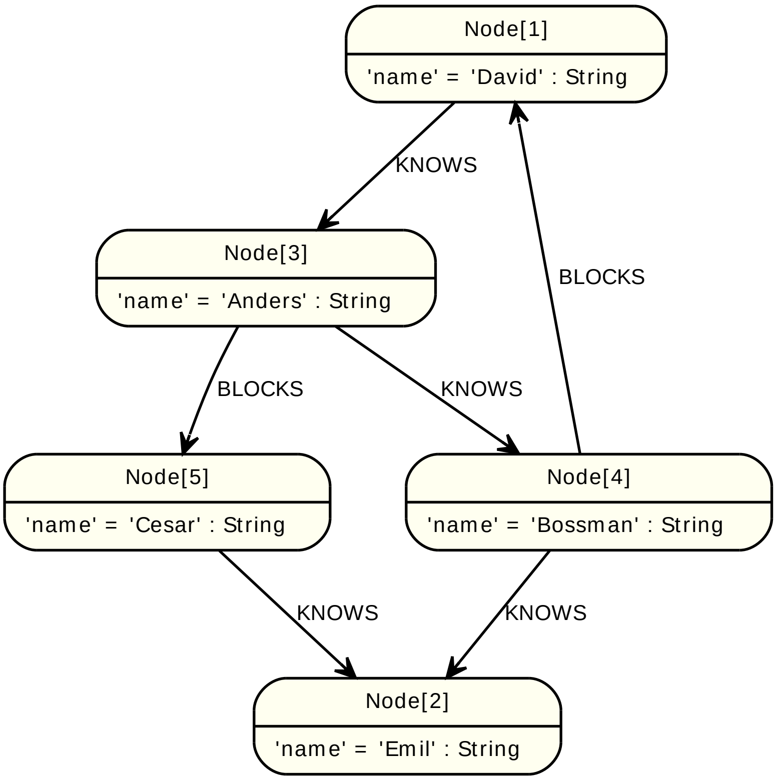
a	b
Node[1]{name->"A"}	Node[2]{name->"B"}
1 rows, 0 ms	

16.4. Match

In the match part of a query, the pattern is described. The description of the pattern is made up of one or more paths, separated by commas.

All parts of the pattern must be directly or indirectly bound to a start point.

Graph



16.4.1. Related nodes

The symbol -- means related to, without regard to type or direction.

Query

```
START n=node(3)
MATCH (n)--(x)
RETURN x
```

All nodes related to A are returned

Result

x
Node[4]{name->"Bossman"}
Node[1]{name->"David"}
Node[5]{name->"Cesar"}
3 rows, 0 ms

16.4.2. Outgoing relationships

When the direction of a relationship is interesting, it is shown by using --> or <--, like this:

Query

```
START n=node(3)
MATCH (n)-->(x)
RETURN x
```

All nodes that A has outgoing relationships to.

Result

x
Node[4]{name->"Bossman"}
Node[5]{name->"Cesar"}
2 rows, 1 ms

16.4.3. Directed relationships and identifier

If an identifier is needed, either for filtering on properties of the relationship, or to return the relationship, this is how you introduce the identifier.

Query

```
START n=node(3)
MATCH (n)-[r]->()
RETURN r
```

All outgoing relationships from node A.

Result

r
:KNOWS[0] {}
:BLOCKS[1] {}
2 rows, 1 ms

16.4.4. Match by relationship type

When you know the relationship type you want to match on, you can specify it by using a colon.

Query

```
START n=node(3)
MATCH (n)-[:BLOCKS]->(x)
RETURN x
```

All nodes that are BLOCKed by A.

Result

x
Node[5]{name->"Cesar"}
1 rows, 1 ms

16.4.5. Match by relationship type and use an identifier

If you both want to introduce an identifier to hold the relationship, and specify the relationship type you want, just add them both, like this.

Query

```
START n=node(3)
```

```
MATCH (n)-[r:BLOCKS]->()
RETURN r
```

All `BLOCKS` relationship going out from A.

Result

r
:BLOCKS[1] {}
1 rows, 0 ms

16.4.6. Relationship types with uncommon characters

Sometime your database will have types with non-letter characters, or with spaces in them. Use ` to escape these.

Query

```
START n=node(3)
MATCH (n)-[r:`TYPE WITH SPACE IN IT`]->()
RETURN r
```

This returns a relationship of a type with spaces in it.

Result

r
:TYPE WITH SPACE IN IT[6] {}
1 rows, 0 ms

16.4.7. Multiple relationships

Relationships can be expressed by using multiple statements in the form of `()--()`, or they can be strung together, like this:

Query

```
START a=node(3)
MATCH (a)-[:KNOWS]->(b)-[:KNOWS]->(c)
RETURN a,b,c
```

The three nodes in the path.

Result

a	b	c
Node[3]{name->"Anders"}	Node[4]{name->"Bossman"}	Node[2]{name->"Emil"}
1 rows, 1 ms		

16.4.8. Variable length relationships

Nodes that are variable number of relationship→node hops can be found using `-[:TYPE*minHops..maxHops]->`.

Query

```
START a=node(3), x=node(2, 4)
MATCH a-[:KNOWS*1..3]->x
RETURN a,x
```

Returns the start and end point, if there is a path between 1 and 3 relationships away

Result

a	x
Node[3]{name->"Anders"}	Node[2]{name->"Emil"}
Node[3]{name->"Anders"}	Node[4]{name->"Bossman"}
2 rows, 1 ms	

16.4.9. Zero length paths

When using variable length paths that have the lower bound zero, it means that two identifiers can point to the same node. If the distance between two nodes is zero, they are, by definition, the same node.

Query

```
START a=node(3)
MATCH p1=a-[:KNOWS*0..1]->b, p2=b-[:BLOCKS*0..1]->c
RETURN a,b,c, length(p1), length(p2)
```

This query will return four paths, some of them with length zero.

Result

a	b	c	LENGTH(p1)	LENGTH(p2)
Node[3]{name->"Anders"}	Node[3]{name->"Anders"}	Node[3]{name->"Anders"}	0	0
Node[3]{name->"Anders"}	Node[3]{name->"Anders"}	Node[5]{name->"Cesar"}	0	1
Node[3]{name->"Anders"}	Node[4]{name->"Bossman"}	Node[4]{name->"Bossman"}	1	0
Node[3]{name->"Anders"}	Node[4]{name->"Bossman"}	Node[1]{name->"David"}	1	1
4 rows, 3 ms				

16.4.10. Optional relationship

If a relationship is optional, it can be marked with a question mark. This similar to how a SQL outer join works, if the relationship is there, it is returned. If it's not, `null` is returned in it's place. Remember that anything hanging of an optional relation, is in turn optional, unless it is connected with a bound node some other path.

Query

```
START a=node(2)
MATCH a-[?]->x
RETURN a,x
```

A node, and `null`, since the node has no relationships.

Result

a	x
Node[2]{name->"Emil"}	<null>
1 rows, 0 ms	

16.4.11. Optional typed and named relationship

Just as with a normal relationship, you can decide which identifier it goes into, and what relationship type you need.

Query

```
START a=node(3)
MATCH a-[r?:LOVES]->()
RETURN a,r
```

A node, and `null`, since the node has no relationships.

Result

a	r
Node[3]{name->"Anders"}	<null>
1 rows, 0 ms	

16.4.12. Properties on optional elements

Returning a property from an optional element that is `null` will also return `null`.

Query

```
START a=node(2)
MATCH a-[?]->x
RETURN x, x.name
```

The element x (null in this query), and null as it's name.

Result

x	x.name
<null>	<null>
1 rows, 1 ms	

16.4.13. Complex matching

Using Cypher, you can also express more complex patterns to match on, like a diamond shape pattern.

Query

```
START a=node(3)
MATCH (a)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:BLOCKS]-(d)-[:KNOWS]-(c)
RETURN a,b,c,d
```

The four nodes in the path.

Result

a	b	c	d
Node[3]{name->"Anders"}	Node[4]{name->"Bossman"}	Node[2]{name->"Emil"}	Node[5]{name->"Cesar"}
1 rows, 2 ms			

16.4.14. Shortest path

Finding the shortest path between two nodes is as easy as using the shortestPath-function, like this.

Query

```
START d=node(1), e=node(2)
MATCH p = shortestPath( d-[*..15]->e )
RETURN p
```

This means: find the shortest path between two nodes, as long as the path is max 15 relationships long. Inside of the parenthesis you write a single link of a path - the starting node, the connecting relationship and the end node. Characteristics describing the relationship like relationship type, max hops and direction are all used when finding the shortest path.

Result

p
(1)--[KNOWS,2]-->(3)--[KNOWS,0]-->(4)--[KNOWS,3]-->(2)
1 rows, 0 ms

16.4.15. Named path

If you want to return or filter on a path in your pattern graph, you can a introduce a named path.

Query

```
START a=node(3)
MATCH p = a-->b
RETURN p
```

The two paths starting from the first node.

Result

p
(3)--[KNOWS,0]-->(4)
(3)--[BLOCKS,1]-->(5)
2 rows, 1 ms

16.4.16. Matching on a bound relationship

When your pattern contains a bound relationship, and that relationship pattern doesn specify direction, Cypher will try to match the relationship where the connected nodes switch sides.

Query

```
START r=rel(0)
MATCH a-[r]-b
RETURN a,b
```

This returns the two connected nodes, once as the start node, and once as the end node

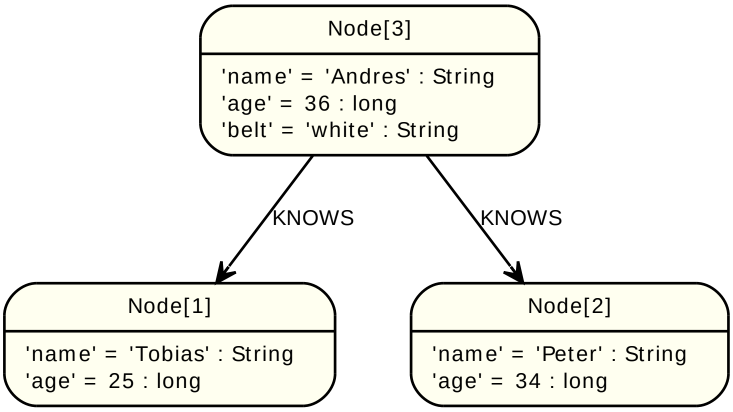
Result

a	**b**
Node[3]{name->"Anders"}	Node[4]{name->"Bossman"}
Node[4]{name->"Bossman"}	Node[3]{name->"Anders"}
2 rows, 0 ms	

16.5. Where

If you need filtering apart from the pattern of the data that you are looking for, you can add clauses in the where part of the query.

Graph



16.5.1. Boolean operations

You can use the expected boolean operators `AND` and `OR`, and also the boolean function `NOT()`.

Query

```
START n=node(3, 1)
WHERE (n.age < 30 and n.name = "Tobias") or not(n.name = "Tobias")
RETURN n
```

The node.

Result

n
Node[3]{name->"Andres",age->36,belt->"white"}
Node[1]{name->"Tobias",age->25}
2 rows, 1 ms

16.5.2. Filter on node property

To filter on a property, write your clause after the `WHERE` keyword.

Query

```
START n=node(3, 1)
WHERE n.age < 30
RETURN n
```

The node.

Result

n
Node[1]{name->"Tobias",age->25}
1 rows, 1 ms

16.5.3. Regular expressions

You can match on regular expressions by using `=~ /regexp/`, like this:

Query

```
START n=node(3, 1)
WHERE n.name =~ /Tob.*/
RETURN n
```

The node named Tobias.

Result

n
Node[1]{name->"Tobias",age->25}
1 rows, 0 ms

16.5.4. Filtering on relationship type

You can put the exact relationship type in the `MATCH` pattern, but sometimes you want to be able to do more advanced filtering on the type. You can use the special property `TYPE` to compare the type with something else. In this example, the query does a regular expression comparison with the name of the relationship type.

Query

```
START n=node(3)
MATCH (n)-[r]->()
WHERE type(r) =~ /K.*/
RETURN r
```

The relationship that has a type whose name starts with K.

Result

r
:KNOWS[0] {}
:KNOWS[1] {}
2 rows, 0 ms

16.5.5. Property exists

To only include nodes/relationships that have a property, just write out the identifier and the property you expect it to have.

Query

```
START n=node(3, 1)
WHERE n.belt
RETURN n
```

The node named Andres.

Result

n
Node[3]{name->"Andres",age->36,belt->"white"}
1 rows, 0 ms

16.5.6. Compare if property exists

If you want to compare a property on a graph element, but only if it exists, use the nullable property syntax. It is the property with the dot notation, followed by a question mark

Query

```
START n=node(3, 1)
WHERE n.belt? = 'white'
RETURN n
```

All nodes, even those without the belt property

Result

n
Node[3]{name->"Andres",age->36,belt->"white"}
Node[1]{name->"Tobias",age->25}
2 rows, 0 ms

16.5.7. Filter on null values

Sometimes you might want to test if a value or an identifier is null. This is done just like SQL does it, with IS NULL. Also like SQL, the negative is IS NOT NULL, althought NOT(IS NULL x) also works.

Query

```
START a=node(1), b=node(3, 2)
MATCH a<-[r?]-b
WHERE r is null
RETURN b
```

Nodes that Tobias is not connected to

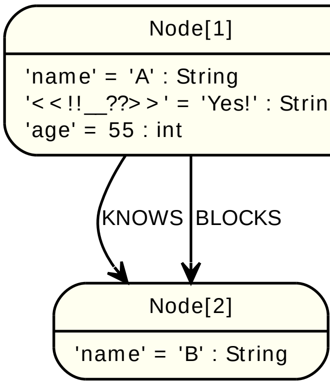
Result

b
Node[2]{name->"Peter",age->34}
1 rows, 1 ms

16.6. Return

In the return part of your query, you define which parts of the pattern you are interested in. It can be nodes, relationships, or properties on these.

Graph



16.6.1. Return nodes

To return a node, list it in the return statemenet.

Query

```
START n=node(2)
RETURN n
```

The node.

Result

n
Node[2]{name->"B"}
1 rows, 0 ms

16.6.2. Return relationships

To return a relationship, just include it in the return list.

Query

```
START n=node(1)
MATCH (n)-[r:KNOWS]->(c)
RETURN r
```

The relationship.

Result

r
:KNOWS[0] {}
1 rows, 1 ms

16.6.3. Return property

To return a property, use the dot separator, like this:

Query

```
START n=node(1)
RETURN n.name
```

The the value of the property *name*.

Result

n.name
A
1 rows, 0 ms

16.6.4. Identifier with uncommon characters

To introduce a placeholder that is made up of characters that are outside of the english alphabet, you can use the ` to enclose the identifier, like this:

Query

```
START `This isn't a common identifier`=node(1)
RETURN `This isn't a common identifier`.`<<!!__??>>`
```

The node indexed with name "A" is returned

Result

This isn't a common identifier.<<!!__??>>
Yes!
1 rows, 0 ms

16.6.5. Optional properties

If a property might or might not be there, you can select it optionally by adding a questionmark to the identifier, like this:

Query

```
START n=node(1, 2)
RETURN n.age?
```

The age when the node has that property, or `null` if the property is not there.

Result

n.age
55
<null>
2 rows, 0 ms

16.6.6. Unique results

`DISTINCT` retrieves only unique rows depending on the columns that have been selected to output.

Query

```
START a=node(1)
MATCH (a)-->(b)
```

```
RETURN distinct b
```

The node named B, but only once.

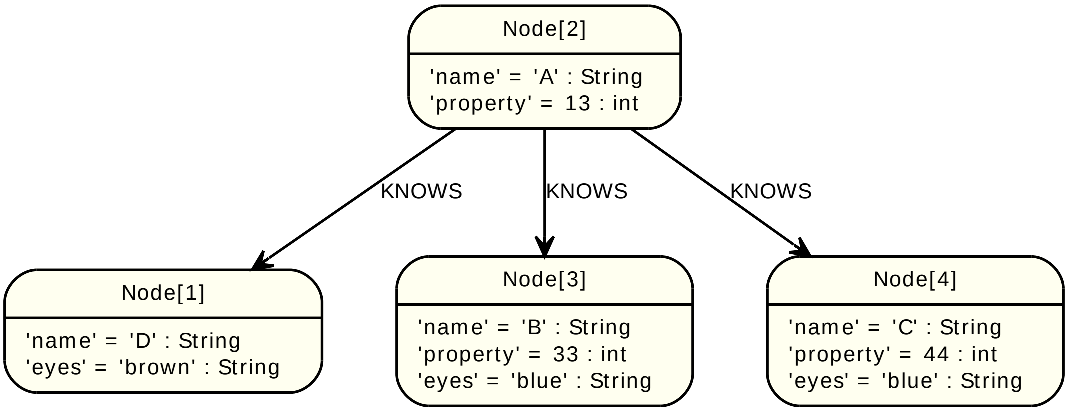
Result

b
Node[2]{name->"B"}
1 rows, 1 ms

16.7. Aggregation

To calculate aggregated data, Cypher offers aggregation, much like SQL's `GROUP BY`. If any aggregation functions are found in the `RETURN` statement, all the columns without aggregating functions are used as the grouping key.

Graph



16.7.1. COUNT

`COUNT` is used to count the number of rows. `COUNT` can be used in two forms - `COUNT(*)` which just counts the number of matching rows, and `COUNT(<identifier>)`, which counts the number of non-null values in `<identifier>`.

16.7.2. Count nodes

To count the number of nodes, for example the number of nodes connected to one node, you can use `count(*)`.

Query

```
START n=node(2)
MATCH (n)-->(x)
RETURN n, count(*)
```

The start node and the count of related nodes.

Result

n	count(*)
Node[2]{name->"A",property->13}	3
1 rows, 1 ms	

16.7.3. Group Count Relationship Types

To count the groups of relationship types, return the types and count them with `count(*)`.

Query

```
START n=node(2)
MATCH (n)-[r]->()
RETURN type(r), count(*)
```

The relationship types and their group count.

Result

TYPE(r)	count(*)
KNOWS	3
1 rows, 1 ms	

16.7.4. Count entities

Instead of counting the number of results with `count(*)`, it might be more expressive to include the name of the identifier you care about.

Query

```
START n=node(2)
MATCH (n)-->(x)
RETURN count(x)
```

The number of connected nodes from the start node.

Result

count(x)
3
1 rows, 1 ms

16.7.5. Count non null values

You can count the non-null values by using `count(<identifier>)`.

Query

```
START n=node(2,3,4,1)
RETURN count(n.property?)
```

The count of related nodes.

Result

count(n.property)
3
1 rows, 1 ms

16.7.6. SUM

The `SUM` aggregation function simply sums all the numeric values it encounters. Null values are silently dropped. This is an example of how you can use `SUM`.

Query

```
START n=node(2,3,4)
RETURN sum(n.property)
```

The sum of all the values in the property *property*.

Result

sum(n.property)
90
1 rows, 0 ms

16.7.7. AVG

`AVG` calculates the average of a numeric column.

Query

```
START n=node(2,3,4)
RETURN avg(n.property)
```

The average of all the values in the property *property*.

Result

avg(n.property)
30.0
1 rows, 0 ms

16.7.8. MAX

`MAX` find the largets value in a numeric column.

Query

```
START n=node(2,3,4)
RETURN max(n.property)
```

The largest of all the values in the property *property*.

Result

max(n.property)
44
1 rows, 0 ms

16.7.9. MIN

`MIN` takes a numeric property as input, and returns the smallest value in that column.

Query

```
START n=node(2,3,4)
RETURN min(n.property)
```

The smallest of all the values in the property *property*.

Result

min(n.property)
13
1 rows, 0 ms

16.7.10. COLLECT

`COLLECT` collects all the values into a list.

Query

```
START n=node(2,3,4)
RETURN collect(n.property)
```

Returns a single row, with all the values collected.

Result

collect(n.property)
`List(13, 33, 44)`
1 rows, 0 ms

16.7.11. DISTINCT

All aggregation functions also take `DISTINCT` modifier, which removes duplicates from the values. So, to count the number of unique eye colours from nodes related to a, this query can be used:

Query

```
START a=node(2)
MATCH a-->b
RETURN count(distinct b.eyes)
```

Returns the number of eye colours.

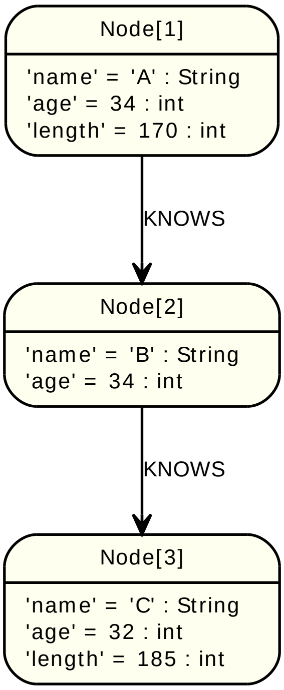
Result

count(distinct b.eyes)
`2`
1 rows, 1 ms

16.8. Order by

To sort the output, use the `ORDER BY` clause. Note that you can not sort on nodes or relationships, just on properties on these.

Graph



16.8.1. Order nodes by property

`ORDER BY` is used to sort the output

Query

```
START n=node(3,1,2)
RETURN n
ORDER BY n.name
```

The nodes, sorted by their name.

Result

n
Node[1]{name->"A",age->34,length->170}
Node[2]{name->"B",age->34}
Node[3]{name->"C",age->32,length->185}
3 rows, 0 ms

16.8.2. Order nodes by multiple properties

You can order by multiple properties by stating each identifier in the `ORDER BY` statement. Cypher will sort the result by the first identifier listed, and for equals values, go to the next property in the order by, and so on.

Query

```
START n=node(3,1,2)
RETURN n
ORDER BY n.age, n.name
```

The nodes, sorted first by their age, and then by their name.

Result

n
Node[3]{name->"C",age->32,length->185}
Node[1]{name->"A",age->34,length->170}
Node[2]{name->"B",age->34}
3 rows, 0 ms

16.8.3. Order nodes in descending order

By adding `DESC[ENDING]` after the identifier to sort on, the sort will be done in reverse order.

Query

```
START n=node(3,1,2)
RETURN n
ORDER BY n.name DESC
```

The nodes, sorted by their name reversely.

Result

n
Node[3]{name->"C",age->32,length->185}
Node[2]{name->"B",age->34}
Node[1]{name->"A",age->34,length->170}
3 rows, 0 ms

16.8.4. Ordering null

When sorting the result set, `null` will always come at the end of the result set for ascending sorting, and first when doing descending sort.

Query

```
START n=node(3,1,2)
RETURN n.length?, n
ORDER BY n.length?
```

The nodes sorted by the length property, with a node without that property last.

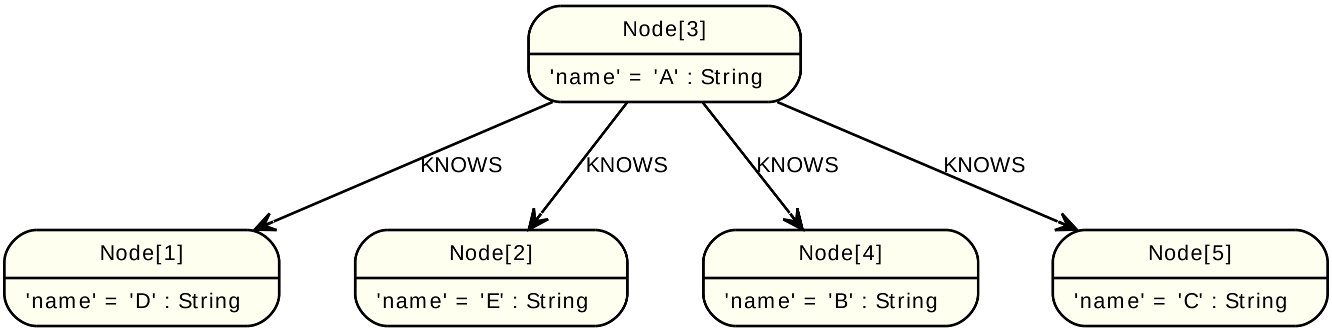
Result

n.length	n
170	Node[1]{name->"A",age->34,length->170}
185	Node[3]{name->"C",age->32,length->185}
<null>	Node[2]{name->"B",age->34}
3 rows, 0 ms	

16.9. Skip

`SKIP` enables the return of only subsets of the total result. By using `SKIP`, the result set will trimmed from the top. Please note that no guarantees are made on the order of the result unless the query specifies the `ORDER BY` clause.

Graph



16.9.1. Skip first three

To return a subset of the result, starting from third result, use this syntax:

Query

```
START n=node(3, 4, 5, 1, 2)
RETURN n
ORDER BY n.name
SKIP 3
```

The first three nodes are skipped, and only the last two are returned.

Result

n
Node[1]{name->"D"}
Node[2]{name->"E"}
2 rows, 1 ms

16.9.2. Return middle two

To return a subset of the result, starting from somewhere in the middle, use this syntax:

Query

```
START n=node(3, 4, 5, 1, 2)
RETURN n
ORDER BY n.name
SKIP 1
LIMIT 2
```

Two nodes from the middle are returned

Result

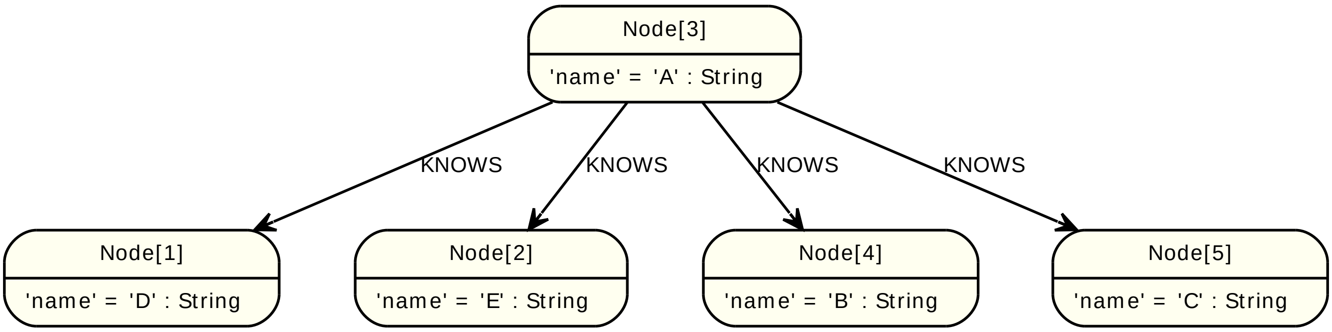
n
Node[4]{name->"B"}
2 rows, 0 ms

n
`Node[5]{name->"C"}`
2 rows, 0 ms

16.10. Limit

`LIMIT` enables the return of only subsets of the total result.

Graph



16.10.1. Return first part

To return a subset of the result, starting from the top, use this syntax:

Query

```
START n=node(3, 4, 5, 1, 2)
RETURN n
LIMIT 3
```

The top three items are returned

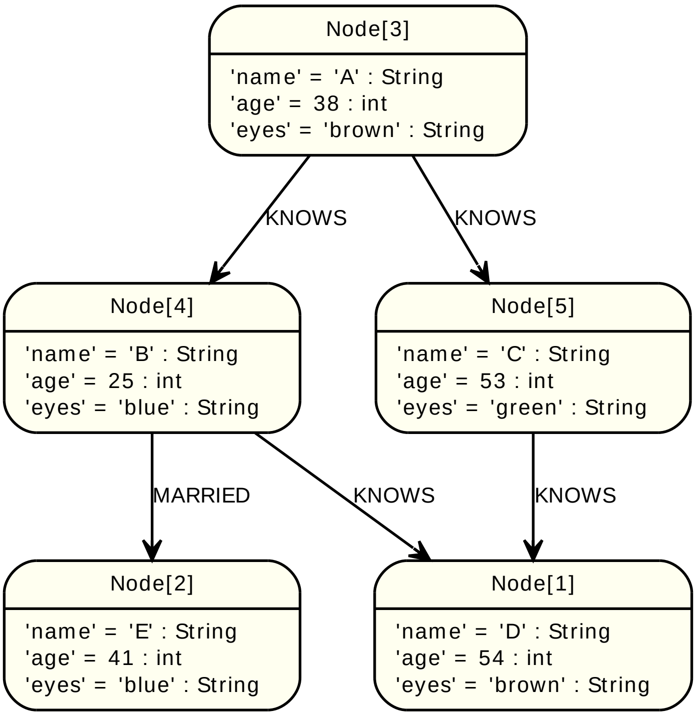
Result

n
Node[3]{name->"A"}
Node[4]{name->"B"}
Node[5]{name->"C"}
3 rows, 0 ms

16.11. Functions

Here is a list of the functions in Cypher, seperated into three different sections: Predicates, Scalar functions and Aggregated functions

Graph



16.11.1. Predicates

Predicates are boolean functions that return true or false for a given set of input. They are most commonly used to filter out subgraphs in the `WHERE` part of a query.

16.11.2. ALL

Tests the predicate closure to see if all items in the iterable match.

Syntax: `ALL(identifier in iterable : predicate)`

Arguments:

- *iterable:* An array property, or an iterable symbol, or an iterable function.
- *symbol:* The closure will have a symbol introduced in it's context. Here you decide which symbol to use. If you leave the symbol out, the default symbol _ (underscore) will be used.
- *predicate:* A predicate that is tested against all items in iterable

Query

```
START a=node(3), b=node(1)
MATCH p=a-[*1..3]->b
WHERE all(x in nodes(p) : x.age > 30)
RETURN p
```

All nodes in the path.

Result

p
(3)--[KNOWS,1]-->(5)--[KNOWS,3]-->(1)
1 rows, 2 ms

16.11.3. ANY

Tests the predicate closure to see if at least one item in the iterable match.

Syntax: `ANY(identifier in iterable : predicate)`

Arguments:

- *iterable:* An array property, or an iterable symbol, or an iterable function.
- *symbol:* The closure will have a symbol introduced in it's context. Here you decide which symbol to use. If you leave the symbol out, the default symbol _ (underscore) will be used.
- *predicate:* A predicate that is tested against all items in iterable

Query

```
START a=node(3)
MATCH p=a-[*1..3]->b
WHERE any(x in nodes(p) : x.eyes = "blue")
RETURN p
```

All nodes in the path.

Result

p
(3)--[KNOWS,0]-->(4)
(3)--[KNOWS,0]-->(4)--[KNOWS,2]-->(1)
(3)--[KNOWS,0]-->(4)--[MARRIED,4]-->(2)
3 rows, 2 ms

16.11.4. NONE

Tests the predicate closure to see if no items in the iterable match. If even one matches, the function returns false.

Syntax: `NONE(identifier in iterable : predicate)`

Arguments:

- *iterable:* An array property, or an iterable symbol, or an iterable function.
- *symbol:* The closure will have a symbol introduced in it's context. Here you decide which symbol to use. If you leave the symbol out, the default symbol _ (underscore) will be used.
- *predicate:* A predicate that is tested against all items in iterable

Query

```
START n=node(3)
MATCH p=n-[*1..3]->b
WHERE NONE(x in nodes(p) : x.age = 25)
RETURN p
```

All nodes in the path.

Result

p
(3)--[KNOWS,1]-->(5)
(3)--[KNOWS,1]-->(5)--[KNOWS,3]-->(1)
2 rows, 3 ms

16.11.5. SINGLE

Returns true if the closure predicate matches exactly one of the items in the iterable.

Syntax: `SINGLE(identifier in iterable : predicate)`

Arguments:

- *iterable:* An array property, or an iterable symbol, or an iterable function.
- *symbol:* The closure will have a symbol introduced in it's context. Here you decide which symbol to use. If you leave the symbol out, the default symbol _ (underscore) will be used.
- *predicate:* A predicate that is tested against all items in iterable

Query

```
START n=node(3)
MATCH p=n-->b
WHERE SINGLE(var in nodes(p) : var.eyes = "blue")
RETURN p
```

All nodes in the path.

Result

p
(3)--[KNOWS,0]-->(4)
1 rows, 1 ms

16.11.6. Scalar functions

Scalar functions return a single value.

16.11.7. LENGTH

To return or filter on the length of a path, use the special property LENGTH

Syntax: `LENGTH( iterable )`

Arguments:

- *iterable:* An iterable, value or function call

Query

```
START a=node(3)
MATCH p=a-->b-->c
RETURN length(p)
```

The length of the path p.

Result

LENGTH(p)
2
2
2
3 rows, 2 ms

16.11.8. TYPE

Returns a string representation of the relationship type.

Syntax: `TYPE( relationship )`

Arguments:

- *relationship:* A relationship

Query

```
START n=node(3)
MATCH (n)-[r]->()
RETURN type(r)
```

The relationship type of r.

Result

TYPE(r)
KNOWS
KNOWS
2 rows, 0 ms

16.11.9. ID

Returns the id of the relationship or node

Syntax: `ID( property-container )`

Arguments:

- *property-container:* A node or a relationship

Query

```
START a=node(3, 4, 5)
RETURN ID(a)
```

The node id for three nodes.

Result

ID(a)
3
4
3 rows, 0 ms

ID(a)
5
3 rows, 0 ms

16.11.10. Iterable functions

Iterable functions return an iterable of things - nodes in a path, and so on.

16.11.11. NODES

Returns all nodes in a path

Syntax: `NODES( path )`

Arguments:

- *path:* A path

Query

```
START a=node(3), c=node(2)
MATCH p=a-->b-->c
RETURN NODES(p)
```

All the nodes in the path p.

Result

NODES(p)
List(Node[3], Node[4], Node[2])
1 rows, 2 ms

16.11.12. RELATIONSHIPS

Returns all relationships in a path

Syntax: `RELATIONSHIPS( path )`

Arguments:

- *path:* A path

Query

```
START a=node(3), c=node(2)
MATCH p=a-->b-->c
RETURN RELATIONSHIPS(p)
```

All the nodes in the path p.

Result

RELATIONSHIPS(p)
List(Relationship[0], Relationship[4])
1 rows, 1 ms

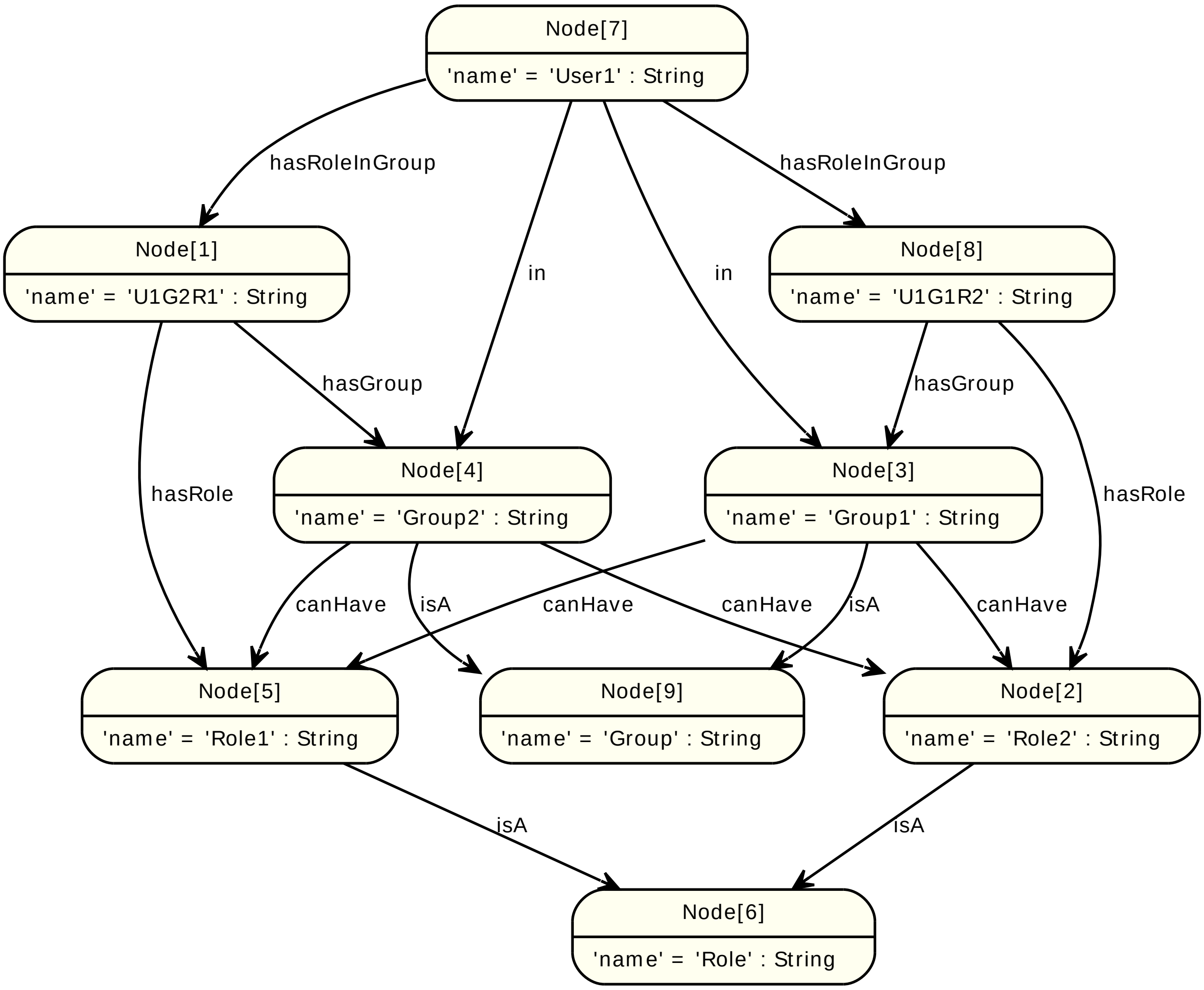
16.12. Cypher Cookbook

The following cookbook aims to provide a few snippets, examples and use-cases and their query-solutions in Cypher.

16.12.1. Hyperedges and Cypher

Imagine a user being part of different groups. A group can have different roles, and a user can be part of different groups. He also can have different roles in different groups apart from the membership. The association of a User, a Group and a Role can be referred to as a *HyperEdge*. However, it can be easily modeled in a property graph as a node that captures this n-ary relationship, as depicted below in the `U1G2R1` node.

Graph



Find Groups

To find out in what roles a user is for a particular groups (here *Group2*), the following Cypher Query can traverse this HyperEdge node and provide answers.

Query

```
START n=node:node_auto_index(name = "User1")
MATCH n-[:hasRoleInGroup]->hyperEdge-[:hasGroup]->group, hyperEdge-[:hasRole]->role
WHERE group.name = "Group2"
RETURN role.name
```

The role of `User1`:

Result

role.name
Role1
1 rows, 2 ms

Find all groups and roles for a user

Here, find all groups and the roles a user has, sorted by the roles names.

Query

```
START n=node:node_auto_index(name = "User1")
MATCH n-[:hasRoleInGroup]->hyperEdge-[:hasGroup]->group, hyperEdge-[:hasRole]->role
RETURN role.name, group.name
ORDER BY role.name ASC
```

The groups and roles of `User1`

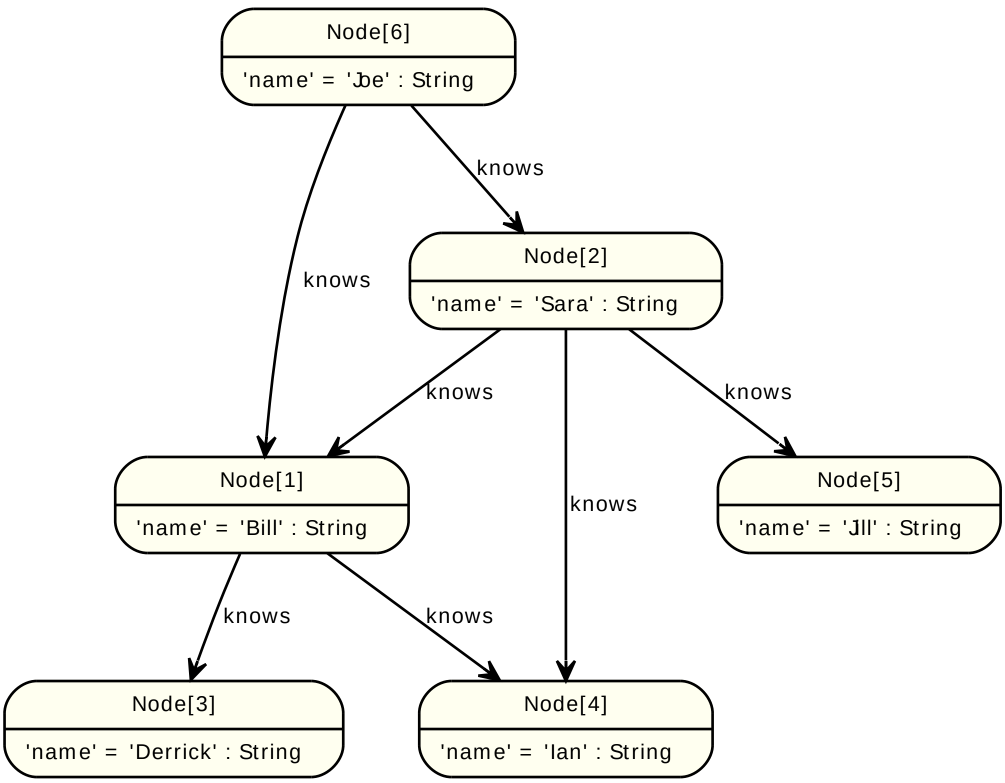
Result

role.name	group.name
Role1	Group2
Role2	Group1
2 rows, 2 ms	

16.12.2. Basic Friend finding based on social neighborhood

Imagine an example graph like

Graph



Simple Friend Finder

To find out the friends of Joes friends that are not already his friends, Cypher looks like:

Query

```
START joe=node:node_auto_index(name = "Joe")
MATCH joe-[:knows]->friend-[:knows]->friend_of_friend, joe-[r?:knows]->friend_of_friend
WHERE r IS NULL
RETURN friend_of_friend.name, COUNT(*)
ORDER BY COUNT(*) DESC, friend_of_friend.name
```

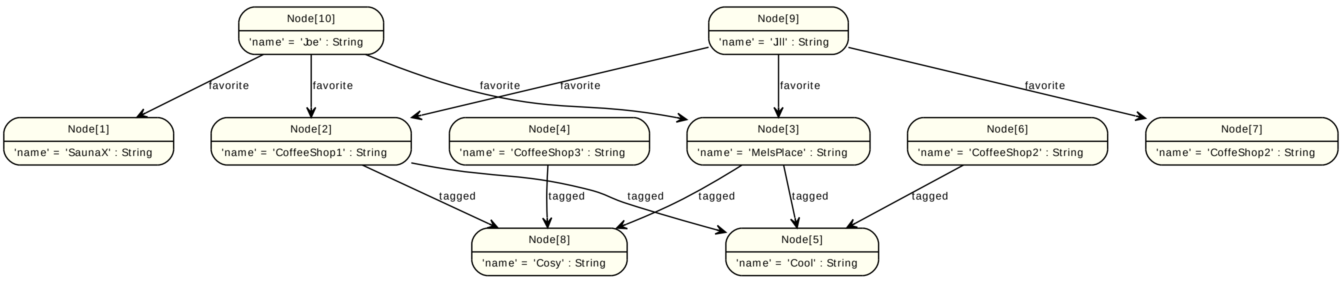
The list of Friends-of-friends order by the number of connections to them, secondly by their name.

Result

friend_of_friend.name	count(*)
Ian	2
Derrick	1
Jill	1
3 rows, 10 ms	

16.12.3. Co-favorited places

Graph



Co-Favorited Places - Users Who Like x Also Like y

Find places that people also like who favorite this place:

- Determine who has favorited place x.
- What else have they favorited that is not place x.

Query

```
START place=node:node_auto_index(name = "CoffeeShop1")
MATCH place<-[:favorite]-person-[:favorite]->stuff
RETURN stuff.name, count(*)
ORDER BY count(*) DESC, stuff.name
```

The list of places that are favorited by people that favorited the start place.

Result

stuff.name	count(*)
MelsPlace	2
CoffeShop2	1
3 rows, 2 ms	

stuff.name	count(*)
SaunaX	1
3 rows, 2 ms	

Co-Tagged Places - Places Related through Tags

Find places that are tagged with the same tags:

- Determine the tags for place x.
- What else is tagged the same as x that is not x.

Query

```
START place=node:node_auto_index(name = "CoffeeShop1")
MATCH place-[:tagged]->tag<-[:tagged]-otherPlace

RETURN otherPlace.name, collect(tag.name)
ORDER By otherPlace.name DESC
```

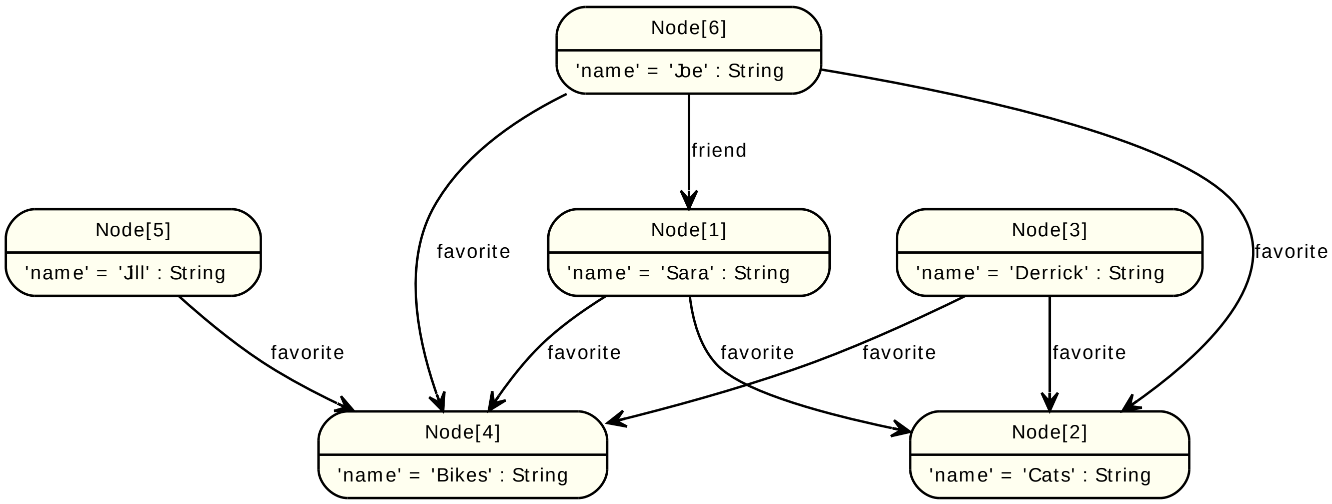
The list of possible friends ranked by them liking similar stuff that are not yet friends.

Result

otherPlace.name	collect(tag.name)
MelsPlace	List(Cool, Cosy)
CoffeeShop3	List(Cosy)
CoffeeShop2	List(Cool)
3 rows, 3 ms	

16.12.4. Find people based on similar favorites

Graph



Find people based on similar favorites

To find out the possible new friends based on them liking similar things as the asking person:

Query

```
START me=node:node_auto_index(name = "Joe")
MATCH me-[:favorite]->stuff<-[:favorite]-person, me-[r?:friend]-person
```

```
WHERE r IS NULL
RETURN person.name, count(stuff)
ORDER BY count(stuff) DESC
```

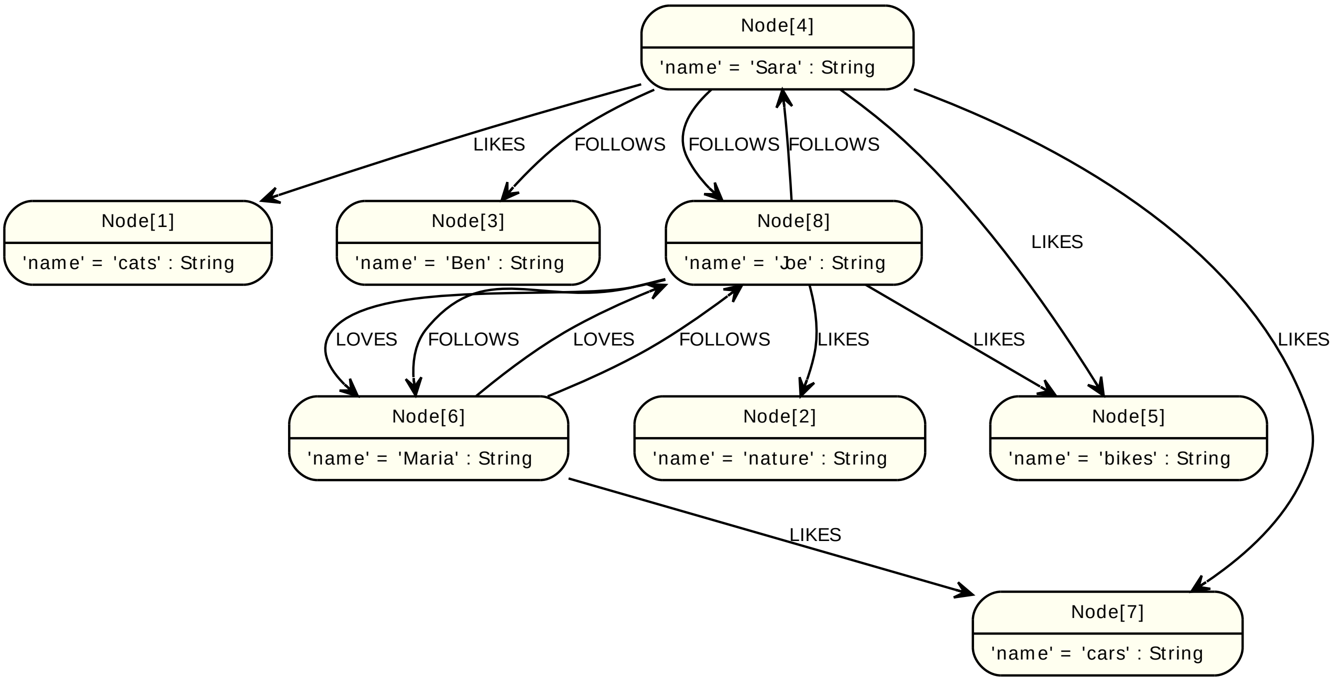
The list of possible friends ranked by them liking similar stuff that are not yet friends.

Result

person.name	count(stuff)
Derrick	2
Jill	1
2 rows, 10 ms	

16.12.5. Multirelational social graph

Graph



Who FOLLOWS or LOVES me back

This example shows a multi-relational * network between persons and things they like. * A multi-relational graph is a graph with more than * one kind of relationship between nodes.

Query

```
START me=node:node_auto_index(name = 'Joe')
MATCH me-[r1]->other-[r2]->me
WHERE type(r1)=type(r2) AND type(r1) =~ /FOLLOWS|LOVES/
RETURN other.name, type(r1)
```

People that `FOLLOWS` or `LOVES` `Joe` back.

Result

other.name	TYPE(r1)
Sara	FOLLOWS
3 rows, 5 ms	

other.name	TYPE(r1)
Maria	FOLLOWS
Maria	LOVES
3 rows, 5 ms	

Chapter 17. Neo4j Server

17.1. 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

17.1.1. 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+`

17.1.2. 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 neo4j-service status`
   should indicate that the server is not running
4. `service neo4j-service start`
   will start the server

   During installation you will be given the option to select the user Neo4j will run as. You will be asked to supply a username (defaulting to `neo4j`) and if that user is not present on the system it will be created as a system account and the `$NEO4J_HOME/data` directory will be `chown`'ed to that user.

   You are encouraged to create a dedicated user for running the service and for that reason it is suggested that you unpack the distribution package under `/opt` or your site specific optional packages directory.

   Finally, note that if you chose to create a new user account, on uninstall you will be prompted to remove it from the system.

17.1.3. Mac OSX Service

Neo4j can be installed as a Mac launchd job:

1. `cd $NEO4J_HOME`
2. `./bin/neo4j install`
3. `launchctl list | grep neo`
   should reveal the launchd "org.neo4j.server.7474" job for running the Neo4j Server
4. `./bin/neo4j status`
   should indicate that the server is running

17.1.4. 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
```

Server 2 (port 7475):

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

17.1.5. High Availability Mode

For information on High Availability, please refer to Chapter 19, *High Availability*.

17.2. 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*
- Configuraion of the deamonizing wrapper are found in *conf/neo4j-wrapper.properties*

17.2.1. Important server configurations parameters

The main configuration file for the server can be found at *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 like this:

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

Note

On Windows systems, absolute locations including drive letters need to read *"c:/data/db"*.

Specify the HTTP server port supporting data, administrative, and UI access:

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

Specify the client accept pattern for the webserver (default is `127.0.0.1`, localhost only):

```
#allow any client to connect
org.neo4j.server.webserver.address=0.0.0.0
```

For securing the Neo4j Server, see also Section 22.1, "Securing access to the Neo4j Server"

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. This should be a relative path.

```
org.neo4j.server.webadmin.data.uri=/db/data/
```

Setting the management URI for the administration API that the Webadmin tool uses. This should be a relative path.

```
org.neo4j.server.webadmin.management.uri=/db/manage
```

Force the server to use IPv4 network addresses, in *conf/neo4j-wrapper.conf* under the section *Java Additional Parameters* add a new paramter:

```
wrapper.java.additional.3=-Djava.net.preferIPv4Stack=true
```

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.

17.2.2. Neo4j Database performance configuration

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 17.5, "Server Performance Tuning". 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.

17.2.3. Logging configuration

The logging framework in use by the Neo4j server is `java.util.logging` `<http://download.oracle.com/javase/6/docs/technotes/guides/logging/overview.html>` and is configured in *conf/logging.properties*.

By default it is setup to print `INFO` level messages both on screen and in a rolling file in *data/log*. 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. On the other hand, if you want to completely silence the console output, set:

```
java.util.logging.ConsoleHandler.level=OFF
```

By default log files are rotated at approximately 10Mb and named consecutively neo4j.<id>.<rotation sequence #>.log To change the naming scheme, rotation frequency and backlog size modify

```
java.util.logging.FileHandler.pattern
java.util.logging.FileHandler.limit
java.util.logging.FileHandler.count
```

respectively to your needs. Details are available at the Javadoc for `java.util.logging.FileHandler` `<http://download.oracle.com/javase/6/docs/api/java/util/logging/FileHandler.html>`.

Apart from log statements originating from the Neo4j server, other libraries report their messages through various frameworks.

Zookeeper is hardwired to use the log4j logging framework. The bundled *conf/log4j.properties* applies for this use only and uses a rolling appender and outputs logs by default to the *data/log* directory.

17.2.4. Other configuration options

Enabling logging from the garbage collector

To get garbage collection logging output you have to pass the corresponding option to the server JVM executable by setting in *conf/neo4j-wrapper.conf* the value

```
wrapper.java.additional.3=-Xloggc:data/log/neo4j-gc.log
```

This line is already present and needs uncommenting. Note also that logging is not directed to console ; You will find the logging statements in *data/log/ne4j-gc.log* or whatever directory you set at the option.

17.3. 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 the *conf/neo4j-wrapper.properties* file.

In order to specify the parameters, add a line for the additional Java arguments like this:

```
# 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-Djava.compiler=NONE-Xru
```

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

17.4. Using the server (including web administration) with an embedded database

Even if you are using the Neo4j Java API directly, for instance via `EmbeddedGraphDatabase` or `HighlyAvailableGraphDatabase`, you can still use the features the server provides.

17.4.1. Getting the libraries

From the Neo4j Server installation

To run the server all the libraries you need are in the *system/lib/* directory of the download package <http://neo4j.org/download/>. For further instructions, see Section 4.1, "Include Neo4j in your project". The only difference to the embedded setup is that *system/lib/* should be added as well, not only the *lib/* directory.

Via Maven

For users of dependency management, an example for Apache Maven <http://maven.apache.org> follows. Note that the web resources are in a different artifact.

Maven pom.xml snippet.

```
<dependencies>
  <dependency>
    <groupId>org.neo4j.app</groupId>
    <artifactId>neo4j-server</artifactId>
    <version>${neo4j-version}</version>
  </dependency>
  <dependency>
    <groupId>org.neo4j.app</groupId>
    <artifactId>neo4j-server</artifactId>
    <classifier>static-web</classifier>
    <version>${neo4j-version}</version>
  </dependency>
</dependencies>
<repositories>
  <repository>
    <id>neo4j-release-repository</id>
    <name>Neo4j Maven 2 release repository</name>
    <url>http://m2.neo4j.org/releases</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

Where `${neo4j-version}` *is the intended version.*

17.4.2. Starting the Server from Java

The Neo4j server exposes a class called WrappingNeoServerBootstrapper <http://components.neo4j.org/neo4j-server/1.5-SNAPSHOT/apidocs/org/neo4j/server/WrappingNeoServerBootstrapper.html>, which is capable of starting a Neo4j server in the same process as your application. It uses an AbstractGraphDatabase <http://components.neo4j.org/neo4j-kernel/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/AbstractGraphDatabase.html> instance that you provide.

This gives your application, among other things, the REST API, statistics gathering and the web administration interface that comes with the server.

Usage example.

```
AbstractGraphDatabase graphdb = getGraphDb();
WrappingNeoServerBootstrapper srv;
srv = new WrappingNeoServerBootstrapper( graphdb );
srv.start();
// The server is now running
// until we stop it:
srv.stop();
```

Once you have the server up and running, see Chapter 24, *Web Administration* and Chapter 18, *REST API* for how to use it!

17.4.3. Providing custom configuration

You can modify the server settings programmatically and, within reason, the same settings are available to you here as those outlined in Section 17.2, "Server Configuration".

The settings that are not available (or rather, that are ignored) are those that concern the underlying database, such as database location and database configuration path.

Custom configuration example.

```
AbstractGraphDatabase graphdb = getGraphDb();
EmbeddedServerConfigurator config;
config = new EmbeddedServerConfigurator( graphdb );
config.configuration().setProperty(
        Configurator.WEBSERVER_PORT_PROPERTY_KEY, 7575 );

WrappingNeoServerBootstrapper srv;
srv = new WrappingNeoServerBootstrapper( graphdb, config );
srv.start();
```

17.5. Server Performance Tuning

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 11.3, "Caches in Neo4j" for more information.

17.5.1. 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.

17.5.2. 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:

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 11.4, "JVM Settings".

Chapter 18. REST API

The Neo4j REST API is designed with discoverability in mind, so that you can start with a GET on the Section 18.1, "Service root" and from there discover URIs to perform other requests. The examples below uses URIs in the examples; they are subject to change in the future, so for future-proofness *discover URIs where possible*, instead of relying on the current layout. The default representation is json <http://www.json.org/>, both for responses and for data sent with POST/PUT requests.

Below follows a listing of ways to interact with the REST API. You can also see a (at runtime) generated description of the API be pointing your browser to the (exact URI may vary) http://localhost:7474/db/data/application.wadl

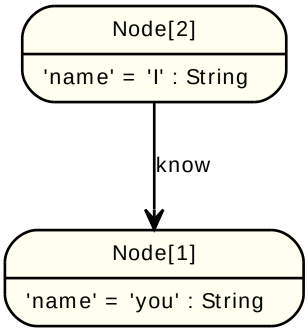
To interact with the JSON interface you must explicitly set the request header `Accept:application/json` for those requests that responds with data. You should also set the header `Content-Type:application/json` if your request sends data, for example when you're creating a relationship. The examples include the relevant request and response headers.

18.1. Service root

18.1.1. Get service root

The service root is your starting point to discover the REST API. It contains the basic starting points for the databse, and some version and extension information. The `reference_node` entry will only be present if there is a reference node set and exists in the database.

Figure 18.1. Final Graph



Example request

- GET `http://localhost:7474/db/data/`
- Accept: `application/json`

Example response

- 200: OK
- Content-Type: `application/json`

```
{
  "relationship_index" : "http://localhost:7474/db/data/index/relationship",
  "node" : "http://localhost:7474/db/data/node",
  "relationship_types" : "http://localhost:7474/db/data/relationship/types",
  "neo4j_version" : "1.5-1-gc02e5d2-dirty",
  "batch" : "http://localhost:7474/db/data/batch",
  "extensions_info" : "http://localhost:7474/db/data/ext",
  "node_index" : "http://localhost:7474/db/data/index/node",
  "reference_node" : "http://localhost:7474/db/data/node/2",
  "extensions" : {
  }
}
```

18.2. Nodes

18.2.1. Create Node

Example request

- POST http://localhost:7474/db/data/node
- Accept: application/json

Example response

- 201: Created
- Content-Type: application/json
- Location: http://localhost:7474/db/data/node/1


```
{
  "outgoing_relationships" : "http://localhost:7474/db/data/node/1/relationships/out",
  "data" : {
  },
  "traverse" : "http://localhost:7474/db/data/node/1/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/1/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/1",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/1/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/1/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/1/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/1/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/1/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/in/{-list|&|types}"
}
```


18.2.2. Create Node with properties

Example request

- POST http://localhost:7474/db/data/node
- Accept: application/json
- Content-Type: application/json

```
{"foo" : "bar"}
```

Example response

- 201: Created
- Content-Length: 1096
- Content-Type: application/json
- Location: http://localhost:7474/db/data/node/2


```
{
  "outgoing_relationships" : "http://localhost:7474/db/data/node/2/relationships/out",
  "data" : {
    "foo" : "bar"
  },
  "traverse" : "http://localhost:7474/db/data/node/2/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/2/relationships/all/{-list|&|types}",
```

```
  "property" : "http://localhost:7474/db/data/node/2/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/2",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/2/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/2/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/2/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/2/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/2/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/2/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/2/relationships/in/{-list|&|types}"
}
```


18.2.3. Get node

Note that the response contains URI/templates for the available operations for getting properties and relationships.

Example request

- GET `http://localhost:7474/db/data/node/1`
- Accept: `application/json`

Example response

- `200`: OK
- Content-Type: `application/json`


```
{
  "outgoing_relationships" : "http://localhost:7474/db/data/node/1/relationships/out",
  "data" : {
  },
  "traverse" : "http://localhost:7474/db/data/node/1/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/1/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/1",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/1/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/1/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/1/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/1/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/1/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/in/{-list|&|types}"
}
```


18.2.4. Get non-existent node

Example request

- GET `http://localhost:7474/db/data/node/600000`
- Accept: `application/json`

Example response

- `404`: Not Found
- Content-Type: `application/json`


```
{
```

```
  "message" : "Cannot find node with id [600000] in database.",
  "exception" : "org.neo4j.server.rest.web.NodeNotFoundException: Cannot find node with id [600000] in database.",
  "stacktrace" : [ "org.neo4j.server.rest.web.DatabaseActions.node(DatabaseActions.java:101)", "org.neo4j.server.rest.web.DatabaseAct
}
```


18.2.5. Delete node

Example request

- DELETE http://localhost:7474/db/data/node/9
- Accept: application/json

Example response

- 204: No Content

18.2.6. Nodes with relationships can not be deleted

The relationships on a node has to be deleted before the node can be deleted.

Example request

- DELETE http://localhost:7474/db/data/node/10
- Accept: application/json

Example response

- 409: Conflict
- Content-Type: application/json


```
{
  "message" : "The node with id 10 cannot be deleted. Check that the node is orphaned before deletion.",
  "exception" : "org.neo4j.server.rest.web.OperationFailureException: The node with id 10 cannot be deleted. Check that the node is o
  "stacktrace" : [ "org.neo4j.server.rest.web.DatabaseActions.deleteNode(DatabaseActions.java:226)", "org.neo4j.server.rest.web.Rest
}
```

18.3. Relationships

Relationships are a first class citizen in the Neo4j REST API. They can be accessed either stand-alone or through the nodes they are attached to.

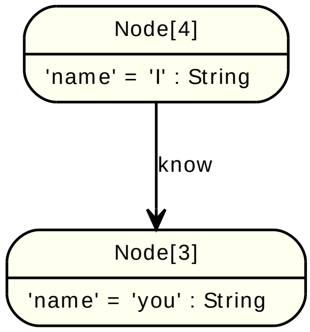
The general pattern to get relationships from a node is:

```
GET http://localhost:7474/db/data/node/123/relationships/{dir}/{-list|&|types}
```

Where `dir` is one of `all`, `in`, `out` and `types` is an ampersand-separated list of types. See the examples below for more information.

18.3.1. Get Relationship by ID

Figure 18.2. Final Graph



Example request

- `GET http://localhost:7474/db/data/relationship/1`
- `Accept: application/json`

Example response

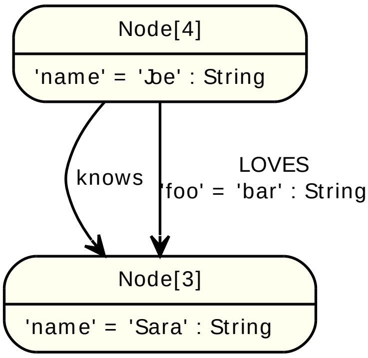
- `200: OK`
- `Content-Type: application/json`


```
{
  "start" : "http://localhost:7474/db/data/node/4",
  "data" : {
  },
  "self" : "http://localhost:7474/db/data/relationship/1",
  "property" : "http://localhost:7474/db/data/relationship/1/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/1/properties",
  "type" : "know",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/3"
}
```


18.3.2. Create relationship

Documentation not available

Figure 18.3. Final Graph



Example request

- POST http://localhost:7474/db/data/node/4/relationships
- Accept: application/json
- Content-Type: application/json

```
{"to" : "http://localhost:7474/db/data/node/3", "type" : "LOVES", "data" : {"foo" : "bar"}}
```

Example response

- 201: Created
- Content-Type: application/json
- Location: http://localhost:7474/db/data/relationship/3

```
{
  "start" : "http://localhost:7474/db/data/node/4",
  "data" : {
    "foo" : "bar"
  },
  "self" : "http://localhost:7474/db/data/relationship/3",
  "property" : "http://localhost:7474/db/data/relationship/3/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/3/properties",
  "type" : "LOVES",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/3"
}
```

18.3.3. Create a relationship with properties

Figure 18.4. Starting Graph

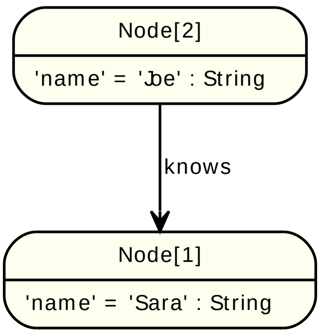
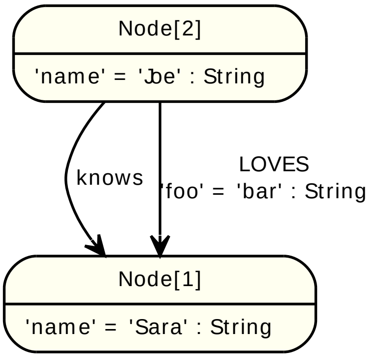


Figure 18.5. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/2/relationships`
- Accept: `application/json`
- Content-Type: `application/json`

```
{"to" : "http://localhost:7474/db/data/node/1", "type" : "LOVES", "data" : {"foo" : "bar"}}
```

Example response

- `201: Created`
- Content-Type: `application/json`
- Location: `http://localhost:7474/db/data/relationship/1`

```
{
  "start" : "http://localhost:7474/db/data/node/2",
  "data" : {
    "foo" : "bar"
  },
  "self" : "http://localhost:7474/db/data/relationship/1",
  "property" : "http://localhost:7474/db/data/relationship/1/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/1/properties",
  "type" : "LOVES",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/1"
}
```

18.3.4. Delete relationship

Figure 18.6. Starting Graph

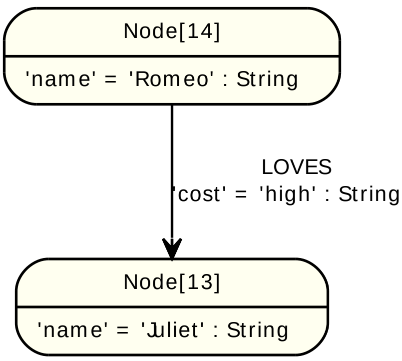
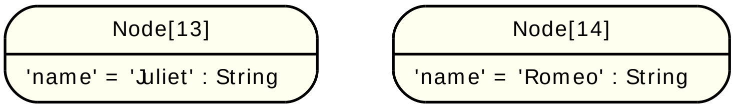


Figure 18.7. Final Graph



Example request

- DELETE `http://localhost:7474/db/data/relationship/6`
- Accept: `application/json`

Example response

- 204: No Content

18.3.5. Get all relationships

Example request

- GET `http://localhost:7474/db/data/node/6/relationships/all`
- Accept: `application/json`

Example response

- 200: OK
- Content-Type: `application/json`


```
[ {
  "start" : "http://localhost:7474/db/data/node/6",
  "data" : {
  },
  "self" : "http://localhost:7474/db/data/relationship/3",
  "property" : "http://localhost:7474/db/data/relationship/3/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/3/properties",
  "type" : "LIKES",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/7"
}, {
  "start" : "http://localhost:7474/db/data/node/8",
  "data" : {
  },
  "self" : "http://localhost:7474/db/data/relationship/4",
  "property" : "http://localhost:7474/db/data/relationship/4/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/4/properties",
  "type" : "LIKES",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/6"
}, {
  "start" : "http://localhost:7474/db/data/node/6",
  "data" : {
  },
  "self" : "http://localhost:7474/db/data/relationship/5",
  "property" : "http://localhost:7474/db/data/relationship/5/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/5/properties",
  "type" : "HATES",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/9"
```

```
} ]
```


18.3.6. Get incoming relationships

Example request

- GET http://localhost:7474/db/data/node/11/relationships/in
- Accept: application/json

Example response

- 200: OK
- Content-Type: application/json


```
[ {
  "start" : "http://localhost:7474/db/data/node/13",
  "data" : {
  },
  "self" : "http://localhost:7474/db/data/relationship/7",
  "property" : "http://localhost:7474/db/data/relationship/7/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/7/properties",
  "type" : "LIKES",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/11"
} ]
```


18.3.7. Get outgoing relationships

Example request

- GET http://localhost:7474/db/data/node/16/relationships/out
- Accept: application/json

Example response

- 200: OK
- Content-Type: application/json


```
[ {
  "start" : "http://localhost:7474/db/data/node/16",
  "data" : {
  },
  "self" : "http://localhost:7474/db/data/relationship/9",
  "property" : "http://localhost:7474/db/data/relationship/9/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/9/properties",
  "type" : "LIKES",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/17"
}, {
  "start" : "http://localhost:7474/db/data/node/16",
  "data" : {
  },
  "self" : "http://localhost:7474/db/data/relationship/11",
  "property" : "http://localhost:7474/db/data/relationship/11/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/11/properties",
  "type" : "HATES",
  "extensions" : {
  },
```

```
  "end" : "http://localhost:7474/db/data/node/19"
} ]
```


18.3.8. Get typed relationships

Note that the "&" needs to be escaped for example when using cURL <http://curl.haxx.se/> from the terminal.

Example request

- GET http://localhost:7474/db/data/node/21/relationships/all/LIKES&HATES
- Accept: application/json

Example response

- 200: OK
- Content-Type: application/json


```
[ {
  "start" : "http://localhost:7474/db/data/node/21",
  "data" : {
  },
  "self" : "http://localhost:7474/db/data/relationship/12",
  "property" : "http://localhost:7474/db/data/relationship/12/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/12/properties",
  "type" : "LIKES",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/22"
}, {
  "start" : "http://localhost:7474/db/data/node/23",
  "data" : {
  },
  "self" : "http://localhost:7474/db/data/relationship/13",
  "property" : "http://localhost:7474/db/data/relationship/13/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/13/properties",
  "type" : "LIKES",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/21"
}, {
  "start" : "http://localhost:7474/db/data/node/21",
  "data" : {
  },
  "self" : "http://localhost:7474/db/data/relationship/14",
  "property" : "http://localhost:7474/db/data/relationship/14/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/14/properties",
  "type" : "HATES",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/24"
} ]
```


18.3.9. Get relationships on a node without relationships

Example request

- GET http://localhost:7474/db/data/node/40/relationships/all
- Accept: application/json

Example response

- `200`: `OK`
- `Content-Type`: `application/json`


```
[ ]
```

18.4. Relationship types

18.4.1. Get relationship types

Example request

- `GET` `http://localhost:7474/db/data/relationship/types`
- `Accept:` `application/json`

Example response

- `200:` `OK`
- `Content-Type:` `application/json`

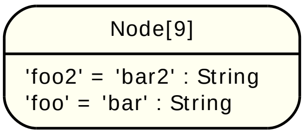
```
["foo","bar"]
```

18.5. Node properties

18.5.1. Set property on node

Setting different properties will retain the existing ones for this node. Note that a single value are submitted not as a map but just as a value (which is valid JSON) like in the example below.

Figure 18.8. Final Graph



Example request

- PUT `http://localhost:7474/db/data/node/9/properties/foo`
- Accept: `application/json`
- Content-Type: `application/json`

```
"bar"
```

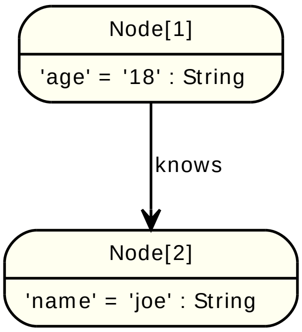
Example response

- 204: No Content

18.5.2. Update node properties

This will replace all existing properties on the node with the new set of attributes.

Figure 18.9. Final Graph



Example request

- PUT `http://localhost:7474/db/data/node/1/properties`
- Accept: `application/json`
- Content-Type: `application/json`

```
{
  "age" : "18"
}
```

Example response

- 204: No Content

18.5.3. Get properties for node

Example request

- GET `http://localhost:7474/db/data/node/5/properties`
- Accept: `application/json`

Example response

- 200: OK
- Content-Type: `application/json`

```
{
  "foo" : "bar"
}
```

18.5.4. Get properties for node (empty result)

If there are no properties, there will be an HTTP 204 response.

Example request

- GET `http://localhost:7474/db/data/node/1/properties`
- Accept: `application/json`

Example response

- 204: No Content

18.5.5. Property values can not be null

This example shows the response you get when trying to set a property to null.

Example request

- POST `http://localhost:7474/db/data/node`
- Accept: `application/json`
- Content-Type: `application/json`

```
{"foo":null}
```

Example response

- 400: Bad Request
- Content-Type: `application/json`

```
{
  "message" : "Could not set property \"foo\", unsupported type: null",
  "exception" : "org.neo4j.server.rest.web.PropertyValueException: Could not set property \"foo\", unsupported type: null",
  "stacktrace" : [ "org.neo4j.server.rest.web.DatabaseActions.set(DatabaseActions.java:133)", "org.neo4j.server.rest.web.DatabaseActi
}
```

18.5.6. Property values can not be nested

Nesting properties is not supported. You could for example store the nested json as a string instead.

Figure 18.10. Final Graph

Example request

- POST http://localhost:7474/db/data/node/
- Accept: application/json
- Content-Type: application/json

```
{"foo" : {"bar" : "baz"}}
```

Example response

- 400: Bad Request
- Content-Type: application/json


```
{
  "message" : "Could not set property \"foo\", unsupported type: {bar=baz}",
  "exception" : "org.neo4j.server.rest.web.PropertyValueException: Could not set property \"foo\", unsupported type: {bar=baz}",
  "stacktrace" : [ "org.neo4j.server.rest.web.DatabaseActions.set(DatabaseActions.java:133)", "org.neo4j.server.rest.web.DatabaseAct
}
```


18.5.7. Delete all properties from node

Example request

- DELETE http://localhost:7474/db/data/node/2/properties
- Accept: application/json

Example response

- 204: No Content

18.6. Relationship properties

18.6.1. Update relationship properties

Example request

- PUT `http://localhost:7474/db/data/relationship/0/properties`
- `Accept: application/json`
- `Content-Type: application/json`

```
{
  "jim" : "tobias"
}
```

Example response

- `204: No Content`

inlcude::remove-properties-from-a-relationship.txt[]

18.6.2. Remove property from a relationship

Documentation not available

Figure 18.11. Starting Graph

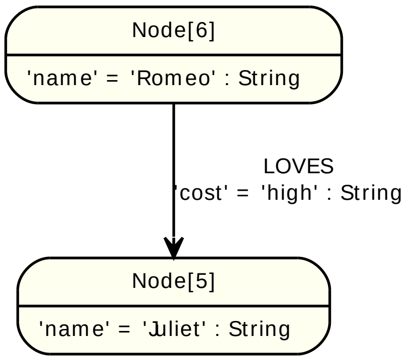
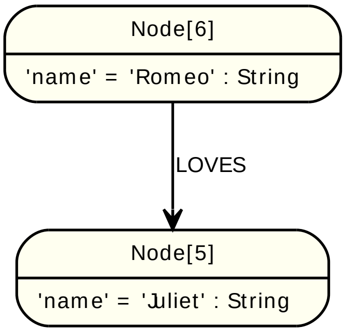


Figure 18.12. Final Graph



Example request

- DELETE `http://localhost:7474/db/data/relationship/2/properties/cost`
- `Accept: application/json`

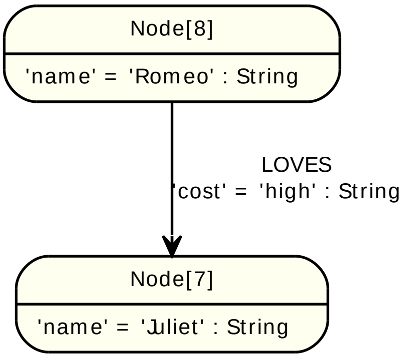
Example response

- 204: No Content

18.6.3. Remove non-existent property from a relationship

Documentation not available

Figure 18.13. Final Graph



Example request

- DELETE http://localhost:7474/db/data/relationship/3/properties/non-existent
- Accept: application/json

Example response

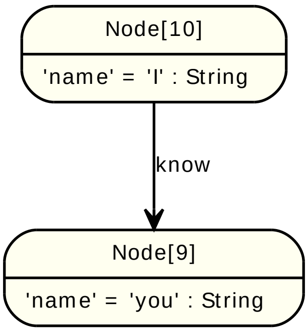
- 404: Not Found
- Content-Type: application/json


```
{
  "message" : "Relationship[3] does not have a property \"non-existent\"",
  "exception" : "org.neo4j.server.rest.web.NoSuchPropertyException: Relationship[3] does not have a property \"non-existent\"",
  "stacktrace" : [ "org.neo4j.server.rest.web.DatabaseActions.removeRelationshipProperty(DatabaseActions.java:649)", "org.neo4j.serve
}
```


18.6.4. Remove properties from a non-existing relationship

Documentation not available

Figure 18.14. Final Graph



Example request

- DELETE http://localhost:7474/db/data/relationship/1234/properties
- Accept: application/json

Example response

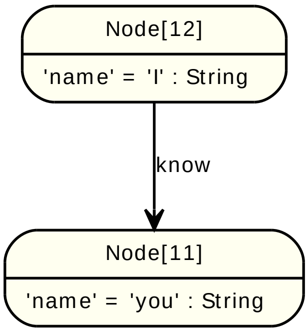
- `404:` Not Found
- `Content-Type:` application/json

```
{
  "exception" : "org.neo4j.server.rest.web.RelationshipNotFoundException",
  "stacktrace" : [ "org.neo4j.server.rest.web.DatabaseActions.relationship(DatabaseActions.java:115)", "org.neo4j.server.rest.web.Da
}
```

18.6.5. Remove property from a non-existing relationship

Documentation not available

Figure 18.15. Final Graph



Example request

- DELETE `http://localhost:7474/db/data/relationship/1234/properties/cost`
- `Accept:` application/json

Example response

- `404:` Not Found
- `Content-Type:` application/json

```
{
  "exception" : "org.neo4j.server.rest.web.RelationshipNotFoundException",
  "stacktrace" : [ "org.neo4j.server.rest.web.DatabaseActions.relationship(DatabaseActions.java:115)", "org.neo4j.server.rest.web.Da
}
```

18.7. Indexes

An index can contain either nodes or relationships.

Note
To create an index with default configuration, simply start using it by adding nodes/relationships to it. It will then be automatically created for you.

What default configuration means depends on how you have configured your database. If you haven't changed any indexing configuration, it means the indexes will be using a Lucene-based backend.

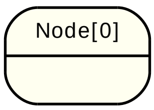
All the examples below show you how to do operations on node indexes, but all of them are just as applicable to relationship indexes. Simple change the "node" part of the URL to "relationship".

If you want to customize the index settings, see Section 18.7.2, "Create node index with configuration".

18.7.1. Create node index

Note
Instead of creating the index this way, you can simply start to use it, and it will be created automatically.

Figure 18.16. Final Graph



Example request

- `POST http://localhost:7474/db/data/index/node/`
- `Accept: application/json`
- `Content-Type: application/json`

```
{
  "name" : "favorites"
}
```

Example response

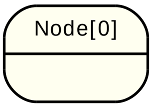
- `201: Created`
- `Content-Type: application/json`
- `Location: http://localhost:7474/db/data/index/node/favorites/`

```
{
  "template" : "http://localhost:7474/db/data/index/node/favorites/{key}/{value}"
}
```

18.7.2. Create node index with configuration

This request is only necessary if you want to customize the index settings. If you are happy with the defaults, you can just start indexing nodes/relationships, as non-existent indexes will automatically be created as you do. See Section 14.10, "Configuration and fulltext indexes" for more information on index configuration.

Figure 18.17. Final Graph



Example request

- POST http://localhost:7474/db/data/index/node/
- Accept: application/json
- Content-Type: application/json

```
{"name":"fulltext", "config":{"type":"fulltext","provider":"lucene"}}
```

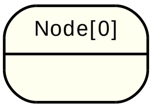
Example response

- 201: Created
- Content-Type: application/json
- Location: http://localhost:7474/db/data/index/node/fulltext/

```
{
  "template" : "http://localhost:7474/db/data/index/node/fulltext/{key}/{value}",
  "provider" : "lucene",
  "type" : "fulltext"
}
```

18.7.3. Delete node index

Figure 18.18. Final Graph



Example request

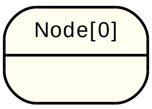
- DELETE http://localhost:7474/db/data/index/node/kvnode
- Accept: application/json

Example response

- 204: No Content

18.7.4. List node indexes

Figure 18.19. Final Graph



Example request

- GET http://localhost:7474/db/data/index/node/
- Accept: application/json

Example response

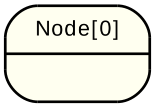
- 200: OK
- Content-Type: application/json


```
{
  "favorites" : {
    "template" : "http://localhost:7474/db/data/index/node/favorites/{key}/{value}",
    "provider" : "lucene",
    "type" : "exact"
  }
}
```


18.7.5. List node indexes (empty result)

This is an example covering the case where no node index exists.

Figure 18.20. Final Graph



Example request

- GET http://localhost:7474/db/data/index/node/
- Accept: application/json

Example response

- 204: No Content

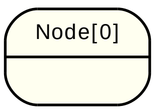
18.7.6. Add node to index

Associates a node with the given key/value pair in the given index.

Note
Spaces in the URI have to be escaped.

Caution
This does **not** overwrite previous entries. If you index the same key/value/item combination twice, two index entries are created. To do update-type operations, you need to delete the old entry before adding a new one.

Figure 18.21. Final Graph



Example request

- POST http://localhost:7474/db/data/index/node/favorites
- Accept: application/json
- Content-Type: application/json

```
{
  "value" : "some value",
  "uri" : "http://localhost:7474/db/data/node/0",
  "key" : "some-key"
}
```

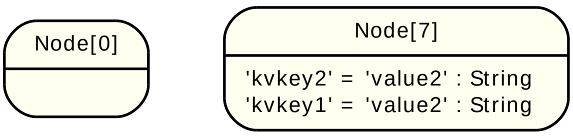

Example response

- 201: Created
- Content-Type: application/json
- Location: http://localhost:7474/db/data/index/node/favorites/some-key/some%20value/0


```
{
  "indexed" : "http://localhost:7474/db/data/index/node/favorites/some-key/some%20value/0",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/0/relationships/out",
  "data" : {
  },
  "traverse" : "http://localhost:7474/db/data/node/0/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/0/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/0/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/0",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/0/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/0/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/0/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/0/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/0/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/0/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/0/relationships/in/{-list|&|types}"
}
```


18.7.7. Remove all entries with a given node from an index

Figure 18.22. Final Graph



Example request

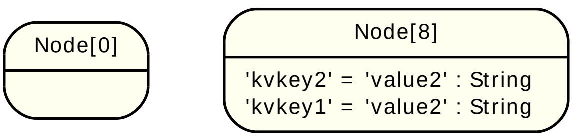
- DELETE http://localhost:7474/db/data/index/node/kvnode/7
- Accept: application/json

Example response

- 204: No Content

18.7.8. Remove all entries with a given node and key from an index

Figure 18.23. Final Graph



Example request

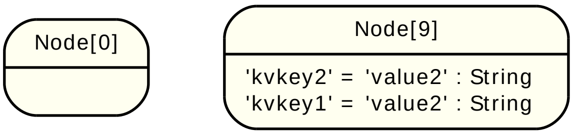
- DELETE http://localhost:7474/db/data/index/node/kvnode/kvkey2/8
- Accept: application/json

Example response

- 204: No Content

18.7.9. Remove all entries with a given node, key and value from an index

Figure 18.24. Final Graph



Example request

- DELETE http://localhost:7474/db/data/index/node/kvnode/kvkey1/value1/9
- Accept: application/json

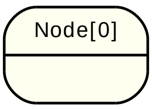
Example response

- 204: No Content

18.7.10. Find node by exact match

Note
Spaces in the URI have to be escaped.

Figure 18.25. Final Graph



Example request

- GET http://localhost:7474/db/data/index/node/favorites/key/the%2520value
- Accept: application/json

Example response

- 200: OK
- Content-Type: application/json


```
[ {
  "indexed" : "http://localhost:7474/db/data/index/node/favorites/key/the%2520value/0",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/0/relationships/out",
  "data" : {
  },
  "traverse" : "http://localhost:7474/db/data/node/0/traverse/{returnType}",
```

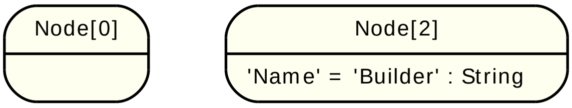
```
  "all_typed_relationships" : "http://localhost:7474/db/data/node/0/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/0/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/0",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/0/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/0/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/0/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/0/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/0/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/0/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/0/relationships/in/{-list|&|types}"
} ]
```


18.7.11. Find node by query

The query language used here depends on what type of index you are querying. The default index type is Lucene, in which case you should use the Lucene query language here. Below and Example of a fuzzy search over multiple keys.

See: http://lucene.apache.org/java/3_1_0/queryparsersyntax.html

Figure 18.26. Final Graph



Example request

- GET `http://localhost:7474/db/data/index/node/bobTheIndex?query=Name:Build~0.1%20AND%20Gender:Male`
- `Accept: application/json`

Example response

- `200: OK`
- `Content-Type: application/json`


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/2/relationships/out",
  "data" : {
    "Name" : "Builder"
  },
  "traverse" : "http://localhost:7474/db/data/node/2/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/2/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/2/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/2",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/2/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/2/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/2/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/2/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/2/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/2/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/2/relationships/in/{-list|&|types}"
} ]
```

18.8. Auto-Indexes

18.8.1. Find node by exact match from an automatic index

Automatic index nodes can be found via exact lookups with normal Index REST syntax.

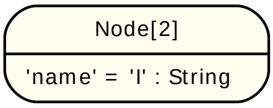


Figure 18.27. Final Graph

Example request

- GET `http://localhost:7474/db/data/index/auto/node/name/I`
- Accept: `application/json`

Example response

- 200: OK
- Content-Type: `application/json`


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/2/relationships/out",
  "data" : {
    "name" : "I"
  },
  "traverse" : "http://localhost:7474/db/data/node/2/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/2/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/2/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/2",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/2/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/2/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/2/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/2/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/2/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/2/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/2/relationships/in/{-list|&|types}"
} ]
```


18.8.2. Find node by query from an automatic index

See Find node by query for the actual query syntax.

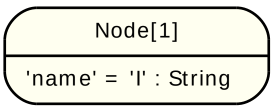


Figure 18.28. Final Graph

Example request

- GET `http://localhost:7474/db/data/index/auto/node/?query=name:I`
- Accept: `application/json`

Example response

- `200`: OK
- `Content-Type`: application/json


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/1/relationships/out",
  "data" : {
    "name" : "I"
  },
  "traverse" : "http://localhost:7474/db/data/node/1/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/1/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/1",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/1/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/1/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/1/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/1/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/1/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/in/{-list|&|types}"
} ]
```

18.9. Configurable Auto-Indexing

Out of the box auto-indexing supports exact matches since they are created with the default configuration (http://docs.neo4j.org/chunked/snapshot/indexing-create.html) the first time you access them. However it is possible to intervene in the lifecycle of the server before any auto indexes are created to change their configuration.

This approach **cannot** be used on databases that already have auto-indexes established. To change the auto-index configuration existing indexes would have to be deleted first, so be careful!

Caution

This technique works, but it is not particularly pleasant. Future versions of Neo4j may remove this loophole in favour of a better structured feature for managing auto-indexing configurations.

Auto-indexing must be enabled through configuration before we can create or configure them. Firstly ensure that you've added some config like this into your server's `neo4j.properties` file:

```
node_auto_indexing=true
relationship_auto_indexing=true
node_keys_indexable=name,phone
relationship_keys_indexable=since
```

The `node_auto_indexing` and `relationship_auto_indexing` turn auto-indexing on for nodes and relationships respectively. The `node_keys_indexable` key allows you to specify a comma-separated list of node property keys to be indexed. The `relationship_keys_indexable` does the same for relationship property keys.

Next start the server as usual by invoking the start script as described in Section 17.1, "Server Installation".

Next we have to pre-empt the creation of an auto-index, by telling the server to create an apparently manual index which has the same name as the node (or relationship) auto-index. For example, in this case we'll create a node auto index whose name is `node_auto_index`, like so:

18.9.1. Create an auto index for nodes with specific configuration

Example request

- `POST http://localhost:7474/db/data/index/node/`
- `Accept: application/json`
- `Content-Type: application/json`

```
{"name":"node_auto_index", "config":{"type":"fulltext","provider":"lucene"}}
```

Example response

- `201: Created`
- `Content-Type: application/json`
- `Location: http://localhost:7474/db/data/index/node/node_auto_index/`

```
{
  "template" : "http://localhost:7474/db/data/index/node/node_auto_index/{key}/{value}",
  "provider" : "lucene",
  "type" : "fulltext"
}
```

If you require configured auto-indexes for relationships, the approach is similar:

18.9.2. Create an auto index for relationships with specific configuration

Example request

- `POST http://localhost:7474/db/data/index/relationship/`
- `Accept: application/json`
- `Content-Type: application/json`

```
{"name":"relationship_auto_index", "config":{"type":"fulltext","provider":"lucene"}}
```

Example response

- `201: Created`
- `Content-Type: application/json`
- `Location: http://localhost:7474/db/data/index/relationship/relationship_auto_index/`

```
{
  "template" : "http://localhost:7474/db/data/index/relationship/relationship_auto_index/{key}/{value}",
  "provider" : "lucene",
  "type" : "fulltext"
}
```

In case you're curious how this works, on the server side it triggers the creation of an index which happens to have the same name as the auto index that the database would create for itself. Now when we interact with the database, the index thinks the index is already is created so the state machine skips over that step and just gets on with normal day-to-day auto-indexing.

Caution

You have to do this early in your server lifecycle, before any normal auto indexes are created.

18.10. Traversals

Traversals are performed from a start node. The traversal is controlled by the URI and the body sent with the request.

returnType
: The kind of objects in the response is determined by *traverse/{returnType}* in the URL. `returnType` can have one of these values:

- `node`
- `relationship`
- `path` - contains full representations of start and end node, the rest are URIs
- `fullpath` - contains full representations of all nodes and relationships

To decide how the graph should be traversed you can use these parameters in the request body:

order
: Decides in which order to visit nodes. Possible values:

- `breadth_first` - see Breadth-first search <http://en.wikipedia.org/wiki/Breadth-first_search>
- `depth_first` - see Depth-first search <http://en.wikipedia.org/wiki/Depth-first_search>

relationships
: Decides which relationship types and directions should be followed. The direction can be one of:

- `all`
- `in`
- `out`

uniqueness
: Decides how uniqueness should be calculated. For details on different uniqueness values see the Java API on Uniqueness <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/kernel/Uniqueness.html>. Possible values:

- `node_global`
- `none`
- `relationship_global`
- `node_path`
- `relationship_path`

prune_evaluator
: Decides whether the traverser should continue down that path or if it should be pruned so that the traverser won't continue down that path. You can write your own prune evaluator as (see Section 18.10.1, "Traversal using a return filter" or use the `built-in none` prune evaluator.

return_filter
: Decides whether the current position should be included in the result. You can provide your own code for this (see Section 18.10.1, "Traversal using a return filter"), or use one of the built-in filters:

- `all`
- `all_but_start_node`

max_depth
: Is a short-hand way of specifying a prune evaluator which prunes after a certain depth. If not specified a max depth of 1 is used and if a `prune_evaluator` is specified instead of a `max_depth`, no max depth limit is set.

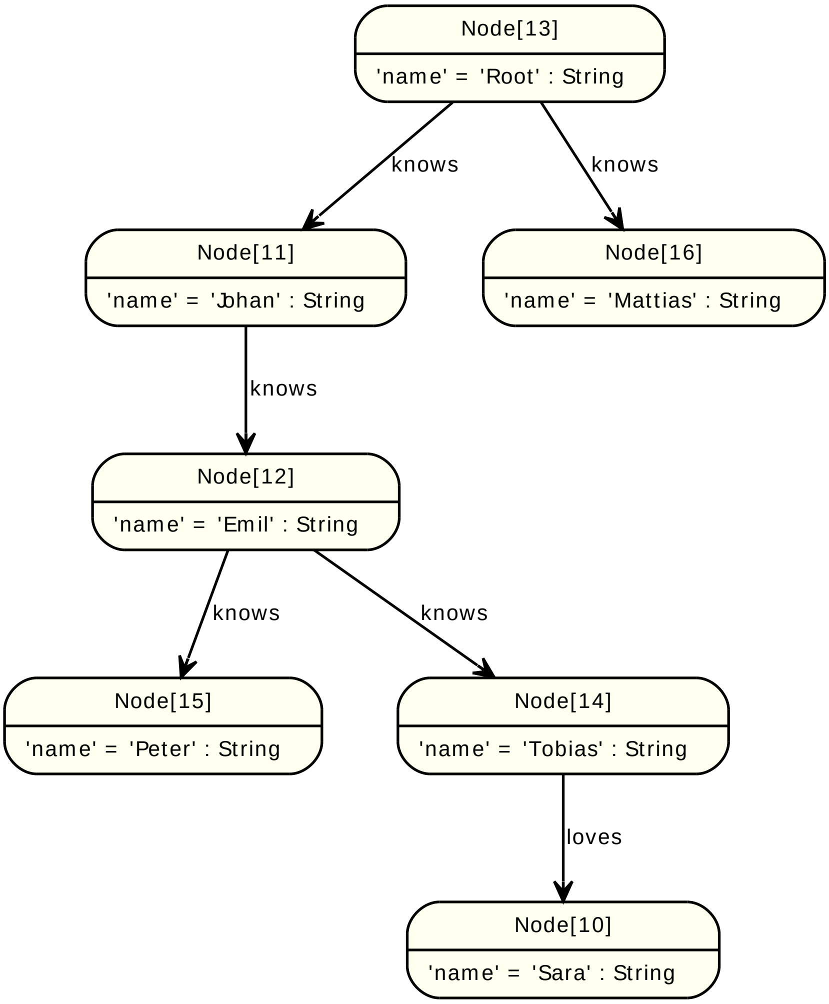
The `position` object in the body of the `return_filter` and `prune_evaluator` is a Path `<http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/org/neo4j/graphdb/Path.html>` object representing the path from the start node to the current traversal position.

Out of the box, the REST API supports JavaScript code in filters and evaluators. The script body will be executed in a Java context which has access to the full Neo4j Java API <http://components.neo4j.org/neo4j/1.5-SNAPSHOT/apidocs/>. See the examples for the exact syntax of the request.

18.10.1. Traversal using a return filter

In this example, the `none` prune evaluator is used and a return filter is supplied in order to return all names containing "t". The result is to be returned as nodes and the max depth is set to 3.

Figure 18.29. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/13/traverse/node`
- Accept: `application/json`
- Content-Type: `application/json`

```
{
  "order" : "breadth_first",
  "return_filter" : {
    "body" : "position.endNode().getProperty('name').toLowerCase().contains('t')",
    "language" : "javascript"
```

```
  },
  "prune_evaluator" : {
    "body" : "position.length() > 10",
    "language" : "javascript"
  },
  "uniqueness" : "node_global",
  "relationships" : [ {
    "direction" : "all",
    "type" : "knows"
  }, {
    "direction" : "all",
    "type" : "loves"
  } ],
  "max_depth" : 3
}
```


Example response

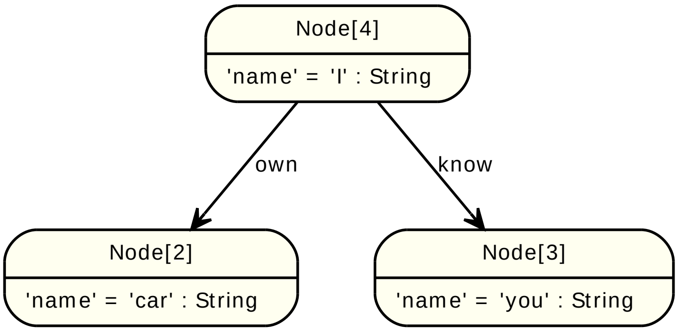
- `200:` OK
- `Content-Type:` application/json


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/13/relationships/out",
  "data" : {
    "name" : "Root"
  },
  "traverse" : "http://localhost:7474/db/data/node/13/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/13/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/13/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/13",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/13/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/13/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/13/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/13/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/13/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/13/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/13/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/16/relationships/out",
  "data" : {
    "name" : "Mattias"
  },
  "traverse" : "http://localhost:7474/db/data/node/16/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/16/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/16/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/16",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/16/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/16/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/16/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/16/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/16/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/16/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/16/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/15/relationships/out",
  "data" : {
    "name" : "Peter"
  },
  "traverse" : "http://localhost:7474/db/data/node/15/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/15/relationships/all/{-list|&|types}",
```

```
  "property" : "http://localhost:7474/db/data/node/15/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/15",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/15/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/15/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/15/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/15/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/15/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/15/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/15/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/14/relationships/out",
  "data" : {
    "name" : "Tobias"
  },
  "traverse" : "http://localhost:7474/db/data/node/14/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/14/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/14/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/14",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/14/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/14/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/14/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/14/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/14/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/14/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/14/relationships/in/{-list|&|types}"
} ]
```


18.10.2. Return relationships from a traversal

Figure 18.30. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/4/traverse/relationship`
- `Accept: application/json`
- `Content-Type: application/json`


```
{"order":"breadth_first","uniqueness":"none","return_filter":{"language":"builtin","name":"all"}}
```


Example response

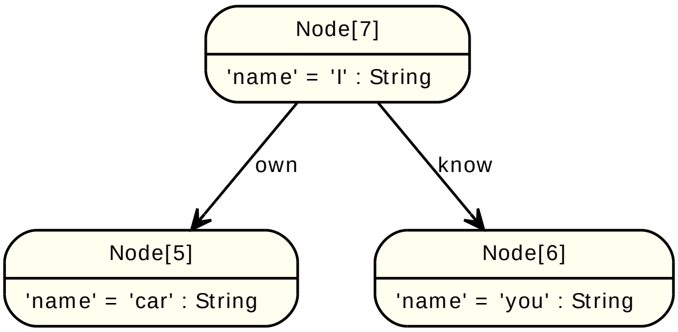
- `200: OK`
- `Content-Type: application/json`


```
[ {
  "start" : "http://localhost:7474/db/data/node/4",
```

```
  "data" : {
  },
  "self" : "http://localhost:7474/db/data/relationship/0",
  "property" : "http://localhost:7474/db/data/relationship/0/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/0/properties",
  "type" : "know",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/3"
}, {
  "start" : "http://localhost:7474/db/data/node/4",
  "data" : {
  },
  "self" : "http://localhost:7474/db/data/relationship/1",
  "property" : "http://localhost:7474/db/data/relationship/1/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/1/properties",
  "type" : "own",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/2"
} ]
```

18.10.3. Return paths from a traversal

Figure 18.31. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/7/traverse/path`
- `Accept: application/json`
- `Content-Type: application/json`

```
{"order":"breadth_first","uniqueness":"none","return_filter":{"language":"builtin","name":"all"}}
```

Example response

- `200: OK`
- `Content-Type: application/json`

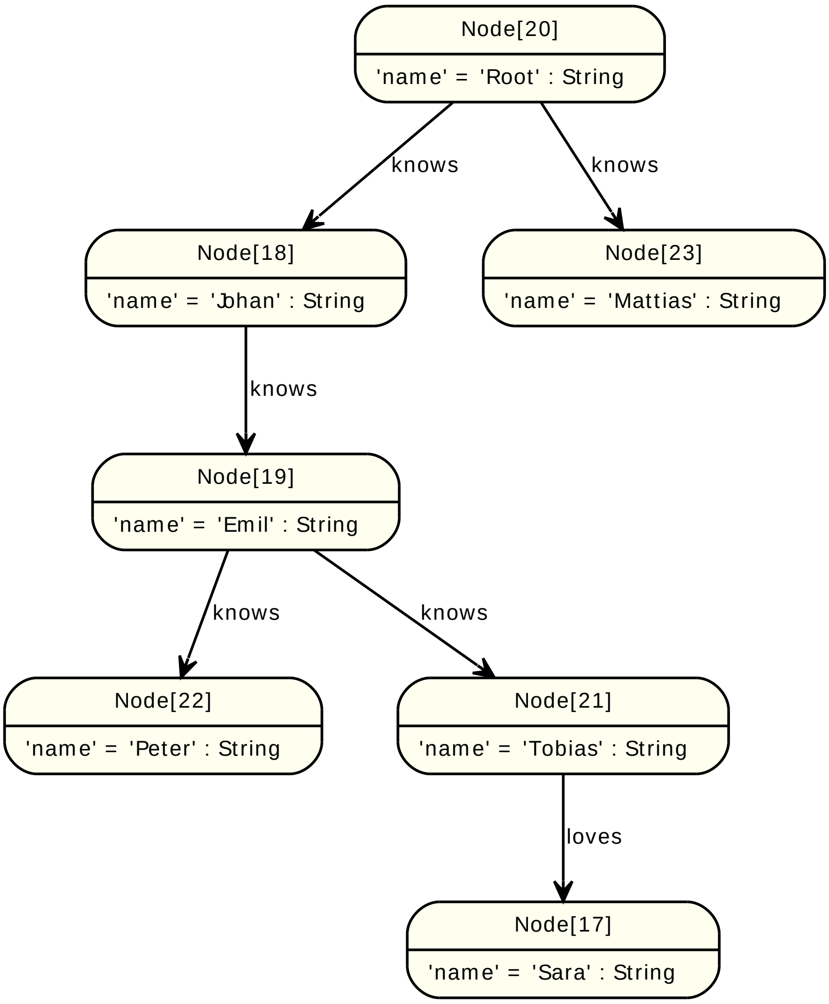
```
[ {
  "start" : "http://localhost:7474/db/data/node/7",
  "nodes" : [ "http://localhost:7474/db/data/node/7" ],
  "length" : 0,
  "relationships" : [ ],
  "end" : "http://localhost:7474/db/data/node/7"
}, {
  "start" : "http://localhost:7474/db/data/node/7",
  "nodes" : [ "http://localhost:7474/db/data/node/7", "http://localhost:7474/db/data/node/6" ],
  "length" : 1,
  "relationships" : [ "http://localhost:7474/db/data/relationship/2" ],
```

```
  "end" : "http://localhost:7474/db/data/node/6"
}, {
  "start" : "http://localhost:7474/db/data/node/7",
  "nodes" : [ "http://localhost:7474/db/data/node/7", "http://localhost:7474/db/data/node/5" ],
  "length" : 1,
  "relationships" : [ "http://localhost:7474/db/data/relationship/3" ],
  "end" : "http://localhost:7474/db/data/node/5"
} ]
```


18.10.4. Traversal returning nodes below a certain depth

Here, all nodes at a traversal depth below 3 are returned.

Figure 18.32. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/20/traverse/node`
- Accept: `application/json`
- Content-Type: `application/json`

```
{
  "return_filter" : {
    "body" : "position.length()<3;",
    "language" : "javascript"
  },
  "prune_evaluator" : {
    "name" : "none",
```

```
    "language" : "builtin"
  }
}
```

Example response

- `200:` OK
- `Content-Type:` application/json


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/20/relationships/out",
  "data" : {
    "name" : "Root"
  },
  "traverse" : "http://localhost:7474/db/data/node/20/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/20/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/20/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/20",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/20/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/20/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/20/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/20/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/20/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/20/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/20/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/23/relationships/out",
  "data" : {
    "name" : "Mattias"
  },
  "traverse" : "http://localhost:7474/db/data/node/23/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/23/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/23/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/23",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/23/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/23/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/23/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/23/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/23/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/23/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/23/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/18/relationships/out",
  "data" : {
    "name" : "Johan"
  },
  "traverse" : "http://localhost:7474/db/data/node/18/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/18/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/18/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/18",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/18/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/18/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/18/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/18/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/18/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/18/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/18/relationships/in/{-list|&|types}"
}, {
```

```
  "outgoing_relationships" : "http://localhost:7474/db/data/node/19/relationships/out",
  "data" : {
    "name" : "Emil"
  },
  "traverse" : "http://localhost:7474/db/data/node/19/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/19/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/19/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/19",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/19/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/19/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/19/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/19/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/19/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/19/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/19/relationships/in/{-list|&|types}"
} ]
```


18.10.5. Creating a paged traverser

Paged traversers are created by `POST`-ing a traversal description to the link identified by the `paged_traverser` key in a node representation. When creating a paged traverser, the same options apply as for a regular traverser, meaning that `node`, `path`, or `fullpath`, can be targeted.

Example request

- `POST http://localhost:7474/db/data/node/34/paged/traverse/node`
- `Accept: application/json`
- `Content-Type: application/json`

```
{
  "prune_evaluator":{
    "language":"builtin",
    "name":"none"
  },
  "return_filter":{
    "language":"javascript",
    "body":"position.endNode().getProperty('name').contains('1');"
  },
  "order":"depth_first",
  "relationships":{
    "type":"NEXT",
    "direction":"out"
  }
}
```

Example response

- `201: Created`
- `Content-Type: application/json`
- `Location: http://localhost:7474/db/data/node/34/paged/traverse/node/bd84026c605a472a99373e31585e176a`


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/35/relationships/out",
  "data" : {
    "name" : "1"
  },
  "traverse" : "http://localhost:7474/db/data/node/35/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/35/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/35/properties/{key}",
```

```
  "self" : "http://localhost:7474/db/data/node/35",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/35/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/35/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/35/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/35/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/35/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/35/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/35/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/44/relationships/out",
  "data" : {
    "name" : "10"
  },
  "traverse" : "http://localhost:7474/db/data/node/44/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/44/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/44/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/44",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/44/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/44/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/44/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/44/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/44/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/44/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/44/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/45/relationships/out",
  "data" : {
    "name" : "11"
  },
  "traverse" : "http://localhost:7474/db/data/node/45/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/45/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/45/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/45",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/45/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/45/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/45/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/45/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/45/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/45/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/45/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/46/relationships/out",
  "data" : {
    "name" : "12"
  },
  "traverse" : "http://localhost:7474/db/data/node/46/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/46/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/46/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/46",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/46/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/46/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/46/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/46/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/46/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/46/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/46/relationships/in/{-list|&|types}"
}, {
```

```
  "outgoing_relationships" : "http://localhost:7474/db/data/node/47/relationships/out",
  "data" : {
    "name" : "13"
  },
  "traverse" : "http://localhost:7474/db/data/node/47/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/47/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/47/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/47",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/47/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/47/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/47/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/47/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/47/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/47/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/47/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/48/relationships/out",
  "data" : {
    "name" : "14"
  },
  "traverse" : "http://localhost:7474/db/data/node/48/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/48/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/48/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/48",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/48/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/48/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/48/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/48/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/48/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/48/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/48/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/49/relationships/out",
  "data" : {
    "name" : "15"
  },
  "traverse" : "http://localhost:7474/db/data/node/49/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/49/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/49/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/49",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/49/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/49/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/49/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/49/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/49/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/49/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/49/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/50/relationships/out",
  "data" : {
    "name" : "16"
  },
  "traverse" : "http://localhost:7474/db/data/node/50/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/50/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/50/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/50",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/50/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/50/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/50/relationships/in",
```

```
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/50/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/50/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/50/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/50/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/51/relationships/out",
  "data" : {
    "name" : "17"
  },
  "traverse" : "http://localhost:7474/db/data/node/51/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/51/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/51/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/51",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/51/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/51/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/51/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/51/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/51/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/51/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/51/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/52/relationships/out",
  "data" : {
    "name" : "18"
  },
  "traverse" : "http://localhost:7474/db/data/node/52/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/52/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/52/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/52",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/52/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/52/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/52/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/52/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/52/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/52/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/52/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/53/relationships/out",
  "data" : {
    "name" : "19"
  },
  "traverse" : "http://localhost:7474/db/data/node/53/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/53/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/53/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/53",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/53/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/53/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/53/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/53/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/53/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/53/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/53/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/55/relationships/out",
  "data" : {
    "name" : "21"
  },
```

```
  "traverse" : "http://localhost:7474/db/data/node/55/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/55/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/55/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/55",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/55/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/55/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/55/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/55/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/55/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/55/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/55/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/65/relationships/out",
  "data" : {
    "name" : "31"
  },
  "traverse" : "http://localhost:7474/db/data/node/65/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/65/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/65/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/65",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/65/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/65/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/65/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/65/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/65/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/65/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/65/relationships/in/{-list|&|types}"
} ]
```


18.10.6. Paging through the results of a paged traverser

Paged traversers hold state on the server, and allow clients to page through the results of a traversal. To progress to the next page of traversal results, the client issues a HTTP GET request on the paged traversal URI which causes the traversal to fill the next page (or partially fill it if insufficient results are available).

Note that if a traverser expires through inactivity it will cause a 404 response on the next `GET` request. Traversers' leases are renewed on every successful access for the same amount of time as originally specified.

When the paged traverser reaches the end of its results, the client can expect a 404 response as the traverser is disposed by the server.

Example request

- `GET http://localhost:7474/db/data/node/67/paged/traverse/node/53942d85f7184ce882d7c70d8f641033`
- `Accept: application/json`

Example response

- `200: OK`
- `Content-Type: application/json`


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/398/relationships/out",
  "data" : {
    "name" : "331"
  },
```

```
  "traverse" : "http://localhost:7474/db/data/node/398/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/398/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/398/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/398",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/398/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/398/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/398/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/398/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/398/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/398/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/398/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/408/relationships/out",
  "data" : {
    "name" : "341"
  },
  "traverse" : "http://localhost:7474/db/data/node/408/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/408/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/408/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/408",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/408/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/408/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/408/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/408/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/408/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/408/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/408/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/418/relationships/out",
  "data" : {
    "name" : "351"
  },
  "traverse" : "http://localhost:7474/db/data/node/418/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/418/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/418/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/418",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/418/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/418/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/418/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/418/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/418/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/418/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/418/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/428/relationships/out",
  "data" : {
    "name" : "361"
  },
  "traverse" : "http://localhost:7474/db/data/node/428/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/428/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/428/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/428",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/428/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/428/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/428/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/428/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/428/paged/traverse/{returnType}{?pageSize,leaseTime}",
```

```
  "all_relationships" : "http://localhost:7474/db/data/node/428/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/428/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/438/relationships/out",
  "data" : {
    "name" : "371"
  },
  "traverse" : "http://localhost:7474/db/data/node/438/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/438/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/438/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/438",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/438/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/438/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/438/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/438/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/438/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/438/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/438/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/448/relationships/out",
  "data" : {
    "name" : "381"
  },
  "traverse" : "http://localhost:7474/db/data/node/448/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/448/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/448/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/448",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/448/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/448/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/448/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/448/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/448/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/448/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/448/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/458/relationships/out",
  "data" : {
    "name" : "391"
  },
  "traverse" : "http://localhost:7474/db/data/node/458/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/458/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/458/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/458",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/458/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/458/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/458/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/458/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/458/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/458/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/458/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/468/relationships/out",
  "data" : {
    "name" : "401"
  },
  "traverse" : "http://localhost:7474/db/data/node/468/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/468/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/468/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/468",
```

```
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/468/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/468/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/468/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/468/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/468/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/468/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/468/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/477/relationships/out",
  "data" : {
    "name" : "410"
  },
  "traverse" : "http://localhost:7474/db/data/node/477/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/477/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/477/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/477",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/477/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/477/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/477/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/477/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/477/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/477/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/477/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/478/relationships/out",
  "data" : {
    "name" : "411"
  },
  "traverse" : "http://localhost:7474/db/data/node/478/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/478/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/478/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/478",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/478/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/478/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/478/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/478/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/478/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/478/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/478/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/479/relationships/out",
  "data" : {
    "name" : "412"
  },
  "traverse" : "http://localhost:7474/db/data/node/479/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/479/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/479/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/479",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/479/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/479/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/479/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/479/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/479/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/479/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/479/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/480/relationships/out",
```

```
  "data" : {
    "name" : "413"
  },
  "traverse" : "http://localhost:7474/db/data/node/480/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/480/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/480/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/480",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/480/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/480/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/480/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/480/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/480/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/480/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/480/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/481/relationships/out",
  "data" : {
    "name" : "414"
  },
  "traverse" : "http://localhost:7474/db/data/node/481/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/481/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/481/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/481",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/481/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/481/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/481/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/481/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/481/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/481/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/481/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/482/relationships/out",
  "data" : {
    "name" : "415"
  },
  "traverse" : "http://localhost:7474/db/data/node/482/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/482/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/482/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/482",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/482/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/482/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/482/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/482/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/482/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/482/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/482/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/483/relationships/out",
  "data" : {
    "name" : "416"
  },
  "traverse" : "http://localhost:7474/db/data/node/483/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/483/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/483/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/483",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/483/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/483/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/483/relationships/in",
  "extensions" : {
```

```
  },
  "create_relationship" : "http://localhost:7474/db/data/node/483/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/483/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/483/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/483/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/484/relationships/out",
  "data" : {
    "name" : "417"
  },
  "traverse" : "http://localhost:7474/db/data/node/484/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/484/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/484/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/484",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/484/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/484/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/484/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/484/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/484/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/484/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/484/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/485/relationships/out",
  "data" : {
    "name" : "418"
  },
  "traverse" : "http://localhost:7474/db/data/node/485/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/485/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/485/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/485",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/485/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/485/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/485/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/485/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/485/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/485/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/485/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/486/relationships/out",
  "data" : {
    "name" : "419"
  },
  "traverse" : "http://localhost:7474/db/data/node/486/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/486/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/486/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/486",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/486/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/486/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/486/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/486/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/486/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/486/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/486/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/488/relationships/out",
  "data" : {
    "name" : "421"
  },
  "traverse" : "http://localhost:7474/db/data/node/488/traverse/{returnType}",
```

```
  "all_typed_relationships" : "http://localhost:7474/db/data/node/488/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/488/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/488",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/488/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/488/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/488/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/488/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/488/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/488/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/488/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/498/relationships/out",
  "data" : {
    "name" : "431"
  },
  "traverse" : "http://localhost:7474/db/data/node/498/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/498/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/498/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/498",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/498/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/498/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/498/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/498/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/498/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/498/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/498/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/508/relationships/out",
  "data" : {
    "name" : "441"
  },
  "traverse" : "http://localhost:7474/db/data/node/508/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/508/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/508/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/508",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/508/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/508/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/508/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/508/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/508/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/508/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/508/relationships/in/{-list|&|types}"
} ]
```


18.10.7. Paged traverser page size

The default page size is 50 items, but depending on the application larger or smaller pages sizes might be appropriate. This can be set by adding a `pageSize` query parameter.

Example request

- POST `http://localhost:7474/db/data/node/544/paged/traverse/node?pageSize=1`
- `Accept: application/json`
- `Content-Type: application/json`


```
{
  "prune_evaluator":{
    "language":"builtin",
```

```
    "name":"none"
  },
  "return_filter":{
    "language":"javascript",
    "body":"position.endNode().getProperty('name').contains('1');"
  },
  "order":"depth_first",
  "relationships":{
    "type":"NEXT",
    "direction":"out"
  }
}
```

Example response

- 201: Created
- Content-Type: application/json
- Location: http://localhost:7474/db/data/node/544/paged/traverse/node/75c4c3acb90b402884e955aa5dc0d23e


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/545/relationships/out",
  "data" : {
    "name" : "1"
  },
  "traverse" : "http://localhost:7474/db/data/node/545/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/545/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/545/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/545",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/545/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/545/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/545/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/545/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/545/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/545/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/545/relationships/in/{-list|&|types}"
} ]
```


18.10.8. Paged traverser timeout

The default timeout for a paged traverser is 60 seconds, but depending on the application larger or smaller timeouts might be appropriate. This can be set by adding a `leaseTime` query parameter with the number of seconds the paged traverser should last.

Example request

- POST http://localhost:7474/db/data/node/577/paged/traverse/node?leaseTime=10
- Accept: application/json
- Content-Type: application/json

```
{
  "prune_evaluator":{
    "language":"builtin",
    "name":"none"
  },
  "return_filter":{
    "language":"javascript",
    "body":"position.endNode().getProperty('name').contains('1');"
  },
  "order":"depth_first",
```

```
  "relationships":{
    "type":"NEXT",
    "direction":"out"
  }
}
```


Example response

- 201: Created
- Content-Type: application/json
- Location: http://localhost:7474/db/data/node/577/paged/traverse/node/a6634609c1a841f5b10fc31657b9665e


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/578/relationships/out",
  "data" : {
    "name" : "1"
  },
  "traverse" : "http://localhost:7474/db/data/node/578/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/578/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/578/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/578",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/578/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/578/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/578/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/578/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/578/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/578/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/578/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/587/relationships/out",
  "data" : {
    "name" : "10"
  },
  "traverse" : "http://localhost:7474/db/data/node/587/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/587/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/587/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/587",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/587/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/587/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/587/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/587/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/587/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/587/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/587/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/588/relationships/out",
  "data" : {
    "name" : "11"
  },
  "traverse" : "http://localhost:7474/db/data/node/588/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/588/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/588/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/588",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/588/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/588/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/588/relationships/in",
  "extensions" : {
  },
```

```
  "create_relationship" : "http://localhost:7474/db/data/node/588/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/588/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/588/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/588/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/589/relationships/out",
  "data" : {
    "name" : "12"
  },
  "traverse" : "http://localhost:7474/db/data/node/589/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/589/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/589/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/589",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/589/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/589/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/589/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/589/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/589/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/589/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/589/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/590/relationships/out",
  "data" : {
    "name" : "13"
  },
  "traverse" : "http://localhost:7474/db/data/node/590/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/590/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/590/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/590",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/590/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/590/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/590/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/590/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/590/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/590/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/590/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/591/relationships/out",
  "data" : {
    "name" : "14"
  },
  "traverse" : "http://localhost:7474/db/data/node/591/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/591/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/591/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/591",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/591/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/591/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/591/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/591/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/591/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/591/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/591/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/592/relationships/out",
  "data" : {
    "name" : "15"
  },
  "traverse" : "http://localhost:7474/db/data/node/592/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/592/relationships/all/{-list|&|types}",
```

```
  "property" : "http://localhost:7474/db/data/node/592/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/592",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/592/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/592/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/592/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/592/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/592/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/592/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/592/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/593/relationships/out",
  "data" : {
    "name" : "16"
  },
  "traverse" : "http://localhost:7474/db/data/node/593/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/593/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/593/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/593",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/593/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/593/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/593/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/593/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/593/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/593/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/593/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/594/relationships/out",
  "data" : {
    "name" : "17"
  },
  "traverse" : "http://localhost:7474/db/data/node/594/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/594/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/594/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/594",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/594/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/594/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/594/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/594/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/594/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/594/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/594/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/595/relationships/out",
  "data" : {
    "name" : "18"
  },
  "traverse" : "http://localhost:7474/db/data/node/595/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/595/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/595/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/595",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/595/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/595/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/595/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/595/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/595/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/595/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/595/relationships/in/{-list|&|types}"
```

```
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/596/relationships/out",
  "data" : {
    "name" : "19"
  },
  "traverse" : "http://localhost:7474/db/data/node/596/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/596/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/596/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/596",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/596/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/596/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/596/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/596/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/596/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/596/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/596/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/598/relationships/out",
  "data" : {
    "name" : "21"
  },
  "traverse" : "http://localhost:7474/db/data/node/598/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/598/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/598/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/598",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/598/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/598/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/598/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/598/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/598/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/598/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/598/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/608/relationships/out",
  "data" : {
    "name" : "31"
  },
  "traverse" : "http://localhost:7474/db/data/node/608/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/608/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/608/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/608",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/608/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/608/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/608/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/608/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/608/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/608/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/608/relationships/in/{-list|&|types}"
} ]
```

18.11. Built-in Graph Algorithms

Neo4j comes with a number of built-in graph algorithms. They are performed from a start node. The traversal is controlled by the URI and the body sent with the request.

algorithm
: The algorithm to choose. If not set, default is `shortestPath`. `algorithm` can have one of these values:

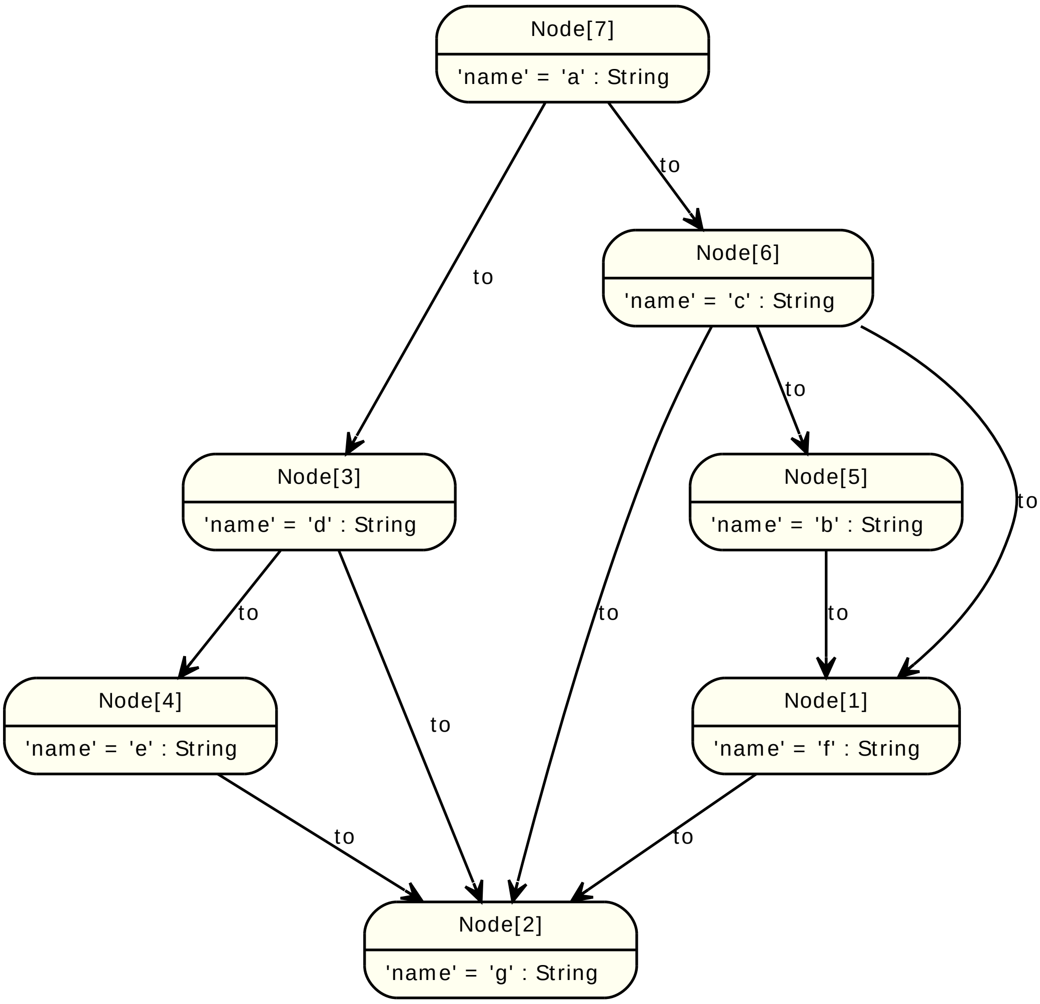
- `shortestPath`
- `allSimplePaths`
- `allPaths`
- `dijkstra` (optional with `cost_property` and `default_cost` parameters)

max_depth
: The maximum depth as an integer for the algorithms like ShortestPath, where applicable. Default is `1`.

18.11.1. Find all shortest paths

The `shortestPath` algorithm can find multiple paths between the same nodes, like in this example.

Figure 18.33. Final Graph



Example request

- `POST http://localhost:7474/db/data/node/7/paths`

- Accept: application/json
- Content-Type: application/json


```
{
  "to":"http://localhost:7474/db/data/node/2",
   "max_depth":3,
   "relationships":{
    "type":"to",
     "direction":"out"
  },
   "algorithm":"shortestPath"
}
```


Example response

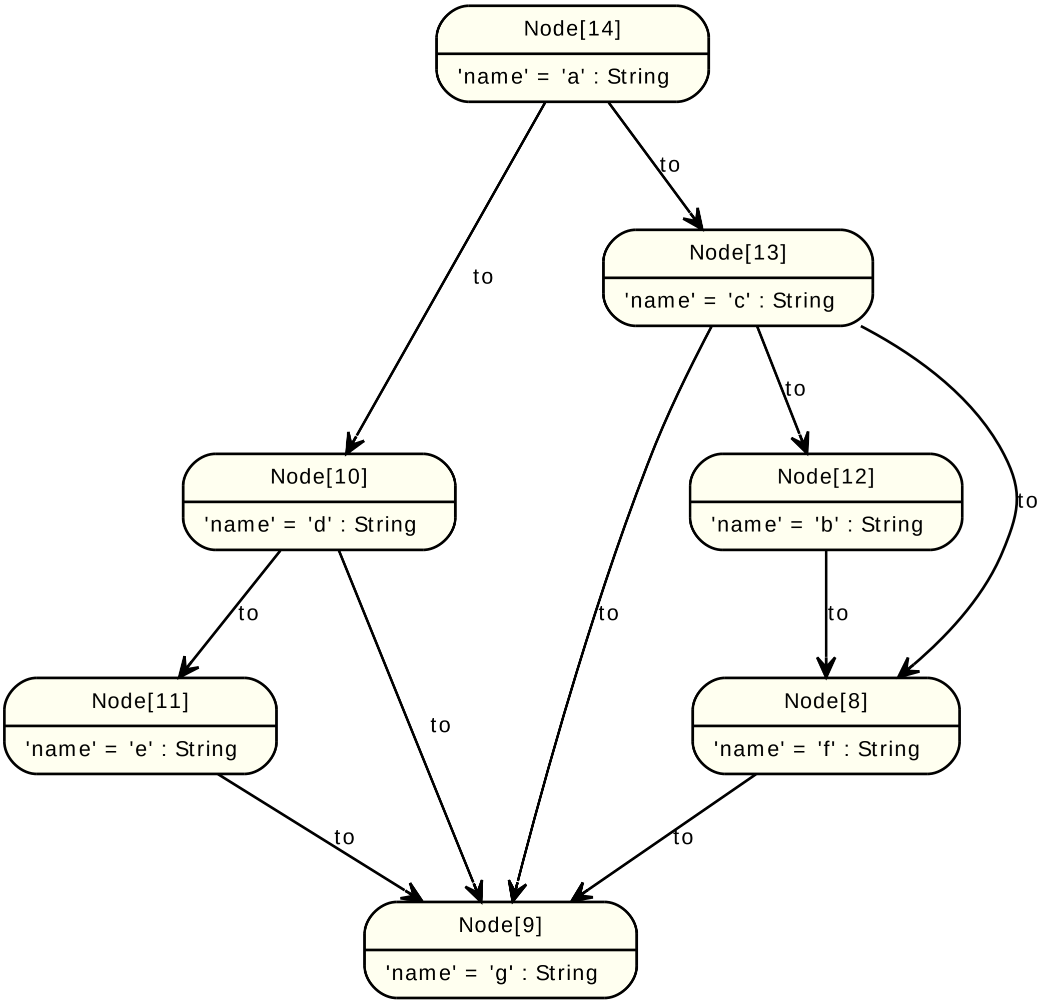
- 200: OK
- Content-Type: application/json


```
[ {
  "start" : "http://localhost:7474/db/data/node/7",
  "nodes" : [ "http://localhost:7474/db/data/node/7", "http://localhost:7474/db/data/node/3", "http://localhost:7474/db/data/node/2"
  "length" : 2,
  "relationships" : [ "http://localhost:7474/db/data/relationship/1", "http://localhost:7474/db/data/relationship/7" ],
  "end" : "http://localhost:7474/db/data/node/2"
}, {
  "start" : "http://localhost:7474/db/data/node/7",
  "nodes" : [ "http://localhost:7474/db/data/node/7", "http://localhost:7474/db/data/node/6", "http://localhost:7474/db/data/node/2"
  "length" : 2,
  "relationships" : [ "http://localhost:7474/db/data/relationship/0", "http://localhost:7474/db/data/relationship/9" ],
  "end" : "http://localhost:7474/db/data/node/2"
} ]
```


18.11.2. Find one of the shortest paths between nodes

If no path algorithm is specified, a ShortestPath algorithm with a max depth of 1 will be chosen. In this example, the max_depth is set to 3 in order to find the shortest path between 3 linked nodes.

Figure 18.34. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/14/path`
- `Accept: application/json`
- `Content-Type: application/json`

```
{
  "to":"http://localhost:7474/db/data/node/9",
   "max_depth":3,
   "relationships":{
    "type":"to",
     "direction":"out"
  },
   "algorithm":"shortestPath"
}
```

Example response

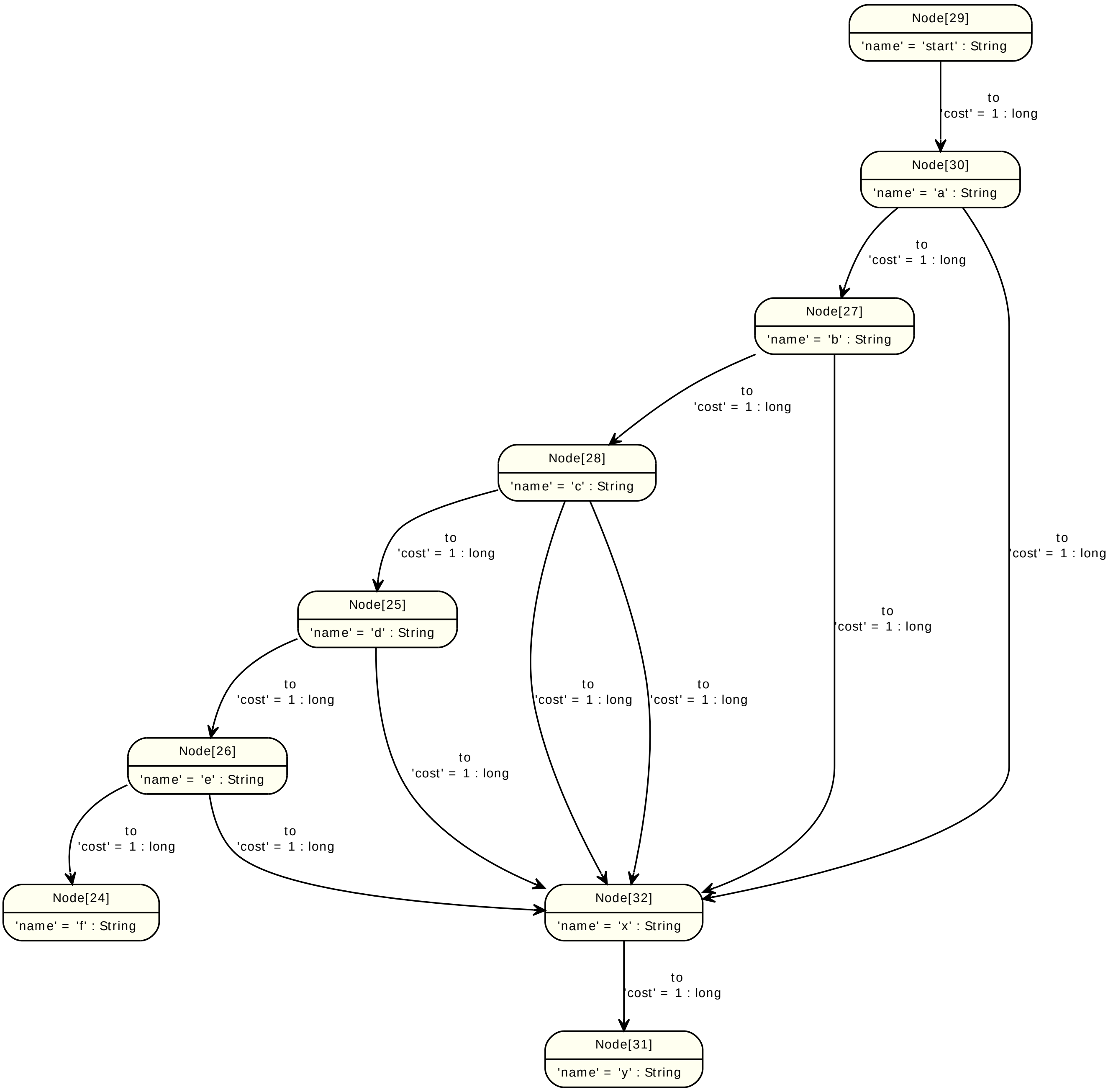
- `200: OK`
- `Content-Type: application/json`

```
{
  "start" : "http://localhost:7474/db/data/node/14",
  "nodes" : [ "http://localhost:7474/db/data/node/14", "http://localhost:7474/db/data/node/10", "http://localhost:7474/db/data/node/
  "length" : 2,
```

```
  "relationships" : [ "http://localhost:7474/db/data/relationship/11", "http://localhost:7474/db/data/relationship/17" ],
  "end" : "http://localhost:7474/db/data/node/9"
}
```

18.11.3. Execute a Dijkstra algorithm with similar weights on relationships

Figure 18.35. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/29/path`
- Accept: `application/json`
- Content-Type: `application/json`

```
{
  "to":"http://localhost:7474/db/data/node/32",
   "cost_property":"cost",
   "relationships":{
```

```
    "type":"to",
     "direction":"out"
  },
   "algorithm":"dijkstra"
}
```

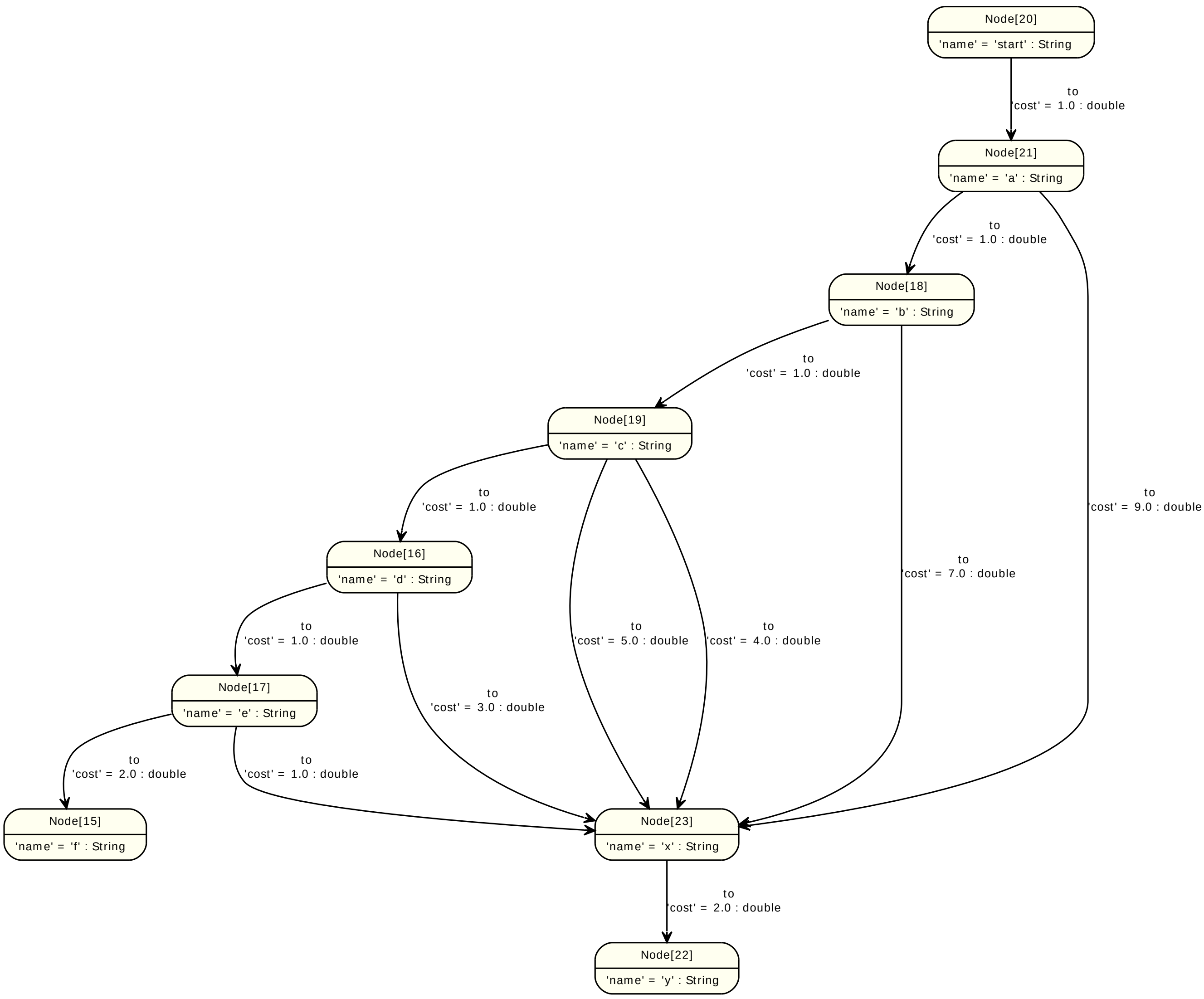
Example response

- 200: OK
- Content-Type: application/json


```
{
  "weight" : 2.0,
  "start" : "http://localhost:7474/db/data/node/29",
  "nodes" : [ "http://localhost:7474/db/data/node/29", "http://localhost:7474/db/data/node/30", "http://localhost:7474/db/data/node/3
  "length" : 2,
  "relationships" : [ "http://localhost:7474/db/data/relationship/33", "http://localhost:7474/db/data/relationship/34" ],
  "end" : "http://localhost:7474/db/data/node/32"
}
```


18.11.4. Execute a Dijkstra algorithm with weights on relationships

Figure 18.36. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/20/path`
- `Accept: application/json`
- `Content-Type: application/json`

```
{
  "to":"http://localhost:7474/db/data/node/23",
   "cost_property":"cost",
   "relationships":{
    "type":"to",
     "direction":"out"
  },
   "algorithm":"dijkstra"
}
```

Example response

- `200: OK`
- `Content-Type: application/json`


```
{
  "weight" : 6.0,
  "start" : "http://localhost:7474/db/data/node/20",
  "nodes" : [ "http://localhost:7474/db/data/node/20", "http://localhost:7474/db/data/node/21", "http://localhost:7474/db/data/node/
  "length" : 6,
  "relationships" : [ "http://localhost:7474/db/data/relationship/20", "http://localhost:7474/db/data/relationship/22", "http://loca
  "end" : "http://localhost:7474/db/data/node/23"
}
```

18.12. Batch operations

Caution

Batch support is currently *experimental*. Expect this part of the API to change.

18.12.1. Execute multiple operations in batch

This lets you execute multiple API calls through a single HTTP call, significantly improving performance for large insert and update operations.

The batch service expects an array of job descriptions as input, each job description describing an action to be performed via the normal server API.

This service is transactional. If any of the operations performed fails (returns a non-2xx HTTP status code), the transaction will be rolled back and all changes will be undone.

Each job description should contain a `path` attribute, with a value relative to the data API root (so http://localhost:7474/db/data/node becomes just /node), and a `method` attribute containing HTTP verb to use.

Optionally you may provide a `body` attribute, and an `id` attribute to help you keep track of responses, although responses are guaranteed to be returned in the same order the job descriptions are received.

The following figure outlines the different parts of the job descriptions:

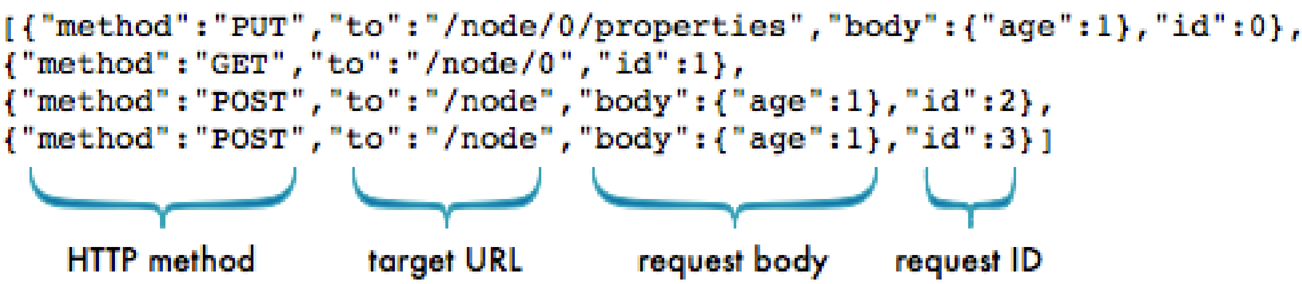
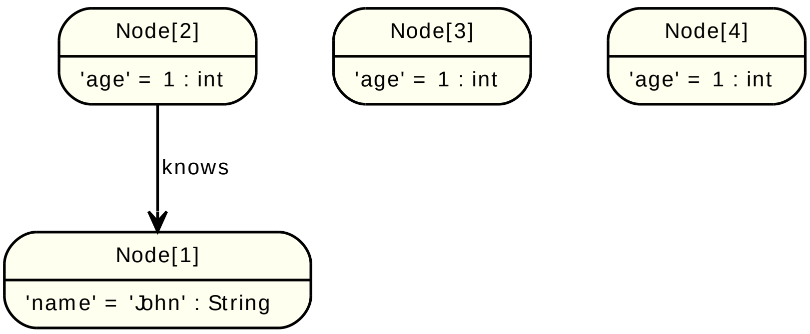


Figure 18.37. Final Graph



Example request

- `POST http://localhost:7474/db/data/batch`
- `Accept: application/json`
- `Content-Type: application/json`

```
[
```

```
  {
    "method":"PUT",
    "to":"/node/2/properties",
    "body":{
      "age":1
    },
    "id":0
  },
  {
    "method":"GET",
    "to":"/node/2",
    "id":1
  },
  {
    "method":"POST",
    "to":"/node",
    "body":{
      "age":1
    },
    "id":2
  },
  {
    "method":"POST",
    "to":"/node",
    "body":{
      "age":1
    },
    "id":3
  }
]
```


Example response

- 200: OK
- Content-Type: application/json


```
[{"id":0,"from":"/node/2/properties"},{"id":1,"body":{
  "outgoing_relationships" : "http://localhost:7474/db/data/node/2/relationships/out",
  "data" : {
    "age" : 1
  },
  "traverse" : "http://localhost:7474/db/data/node/2/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/2/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/2/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/2",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/2/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/2/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/2/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/2/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/2/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/2/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/2/relationships/in/{-list|&|types}"
},"from":"/node/2"},{"id":2,"location":"http://localhost:7474/db/data/node/3","body":{
  "outgoing_relationships" : "http://localhost:7474/db/data/node/3/relationships/out",
  "data" : {
    "age" : 1
  },
  "traverse" : "http://localhost:7474/db/data/node/3/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/3/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/3/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/3",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/3/relationships/out/{-list|&|types}",
```

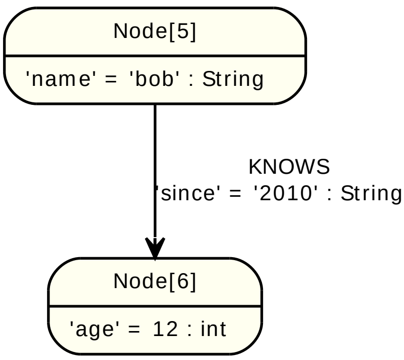
```
  "properties" : "http://localhost:7474/db/data/node/3/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/3/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/3/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/3/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/3/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/3/relationships/in/{-list|&|types}"
},"from":"/node"},{"id":3,"location":"http://localhost:7474/db/data/node/4","body":{
  "outgoing_relationships" : "http://localhost:7474/db/data/node/4/relationships/out",
  "data" : {
    "age" : 1
  },
  "traverse" : "http://localhost:7474/db/data/node/4/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/4/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/4/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/4",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/4/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/4/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/4/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/4/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/4/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/4/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/4/relationships/in/{-list|&|types}"
},"from":"/node"}]
```


18.12.2. Refer to items created earlier in the same batch job

The batch operation API allows you to refer to the URI returned from a created resource in subsequent job descriptions, within the same batch call.

Use the `{[JOB ID]}` special syntax to inject URIs from created resources into JSON strings in subsequent job descriptions.

Figure 18.38. Final Graph



Example request

- `POST http://localhost:7474/db/data/batch`
- `Accept: application/json`
- `Content-Type: application/json`

```
[
  {
    "method":"POST",
    "to":"/node",
    "id":0,
```

```
    "body":{
      "name":"bob"
    }
  },
  {
    "method":"POST",
    "to":"/node",
    "id":1,
    "body":{
      "age":12
    }
  },
  {
    "method":"POST",
    "to":"{0}/relationships",
    "id":3,
    "body":{
      "to":"{1}",
      "data":{
        "since":"2010"
      },
      "type":"KNOWS"
    }
  },
  {
    "method":"POST",
    "to":"/index/relationship/my_rels",
    "id":4,
    "body":{
      "key":"since",
      "value":"2010",
      "uri":"{3}"
    }
  }
]
```

Example response

- 200: OK
- Content-Type: application/json


```
[{"id":0,"location":"http://localhost:7474/db/data/node/5","body":{
  "outgoing_relationships" : "http://localhost:7474/db/data/node/5/relationships/out",
  "data" : {
    "name" : "bob"
  },
  "traverse" : "http://localhost:7474/db/data/node/5/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/5/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/5/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/5",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/5/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/5/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/5/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/5/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/5/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/5/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/5/relationships/in/{-list|&|types}"
},"from":"/node"},{"id":1,"location":"http://localhost:7474/db/data/node/6","body":{
  "outgoing_relationships" : "http://localhost:7474/db/data/node/6/relationships/out",
  "data" : {
    "age" : 12
  },
```

```
  "traverse" : "http://localhost:7474/db/data/node/6/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/6/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/6/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/6",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/6/relationships/out/{-list|&|types}",
  "properties" : "http://localhost:7474/db/data/node/6/properties",
  "incoming_relationships" : "http://localhost:7474/db/data/node/6/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/6/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/6/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/6/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/6/relationships/in/{-list|&|types}"
},"from":"/node"},{"id":3,"location":"http://localhost:7474/db/data/relationship/1","body":{
  "start" : "http://localhost:7474/db/data/node/5",
  "data" : {
    "since" : "2010"
  },
  "self" : "http://localhost:7474/db/data/relationship/1",
  "property" : "http://localhost:7474/db/data/relationship/1/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/1/properties",
  "type" : "KNOWS",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/6"
},"from":"http://localhost:7474/db/data/node/5/relationships"},{"id":4,"location":"http://localhost:7474/db/data/index/relationship/
  "indexed" : "http://localhost:7474/db/data/index/relationship/my_rels/since/2010/1",
  "start" : "http://localhost:7474/db/data/node/5",
  "data" : {
    "since" : "2010"
  },
  "self" : "http://localhost:7474/db/data/relationship/1",
  "property" : "http://localhost:7474/db/data/relationship/1/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/1/properties",
  "type" : "KNOWS",
  "extensions" : {
  },
  "end" : "http://localhost:7474/db/data/node/6"
},"from":"/index/relationship/my_rels"}]
```

18.13. Cypher Plugin

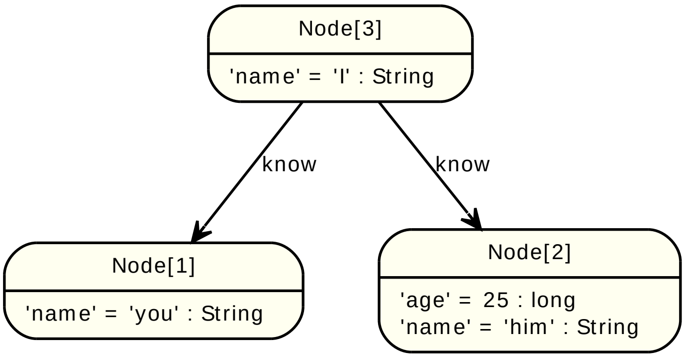
The Neo4j Cypher Plugin enables querying with the Chapter 16, *Cypher Query Language*. The results are returned as a list of string headers (`columns`), and a `data` part, consisting of a list of all rows, every row consisting of a list of REST representations of the field value - `Node`, `Relationship` or any simple value like `String`.

18.13.1. Send a Query

A simple query returning all nodes connected to node 1, returning the node and the name property, if it exists, otherwise `null`:

```
START x  = node(3)
MATCH (x) -[r]-> (n)
RETURN type(r), n.name?, n.age?
```

Figure 18.39. Final Graph



Example request

- `POST http://localhost:7474/db/data/ext/CypherPlugin/graphdb/execute_query`
- `Accept: application/json`
- `Content-Type: application/json`


```
{"query": "start x  = node(3) match (x) -[r]-> (n) return type(r), n.name?, n.age?","params": {}},
```


Example response

- `200: OK`
- `Content-Type: application/json`

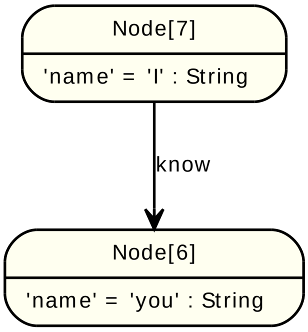

```
{
  "data" : [ [ "know", "him", 25 ], [ "know", "you", null ] ],
  "columns" : [ "TYPE(r)", "n.name", "n.age" ]
}
```


18.13.2. Return paths

Paths can be returned together with other return types by just specifying returns.

```
START x  = node(%I%)
MATCH path = (x--friend)
RETURN path, friend.name
```

Figure 18.40. Final Graph



Example request

- POST http://localhost:7474/db/data/ext/CypherPlugin/graphdb/execute_query
- Accept: application/json
- Content-Type: application/json

```
{"query": "start x  = node(7) match path = (x--friend) return path, friend.name","params": {}},
```

Example response

- 200: OK
- Content-Type: application/json

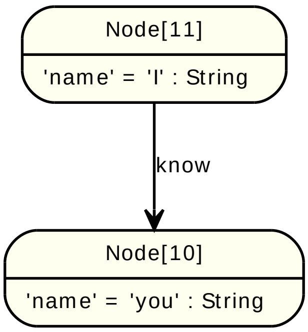
```
{
  "data" : [ [ {
    "start" : "http://localhost:7474/db/data/node/7",
    "nodes" : [ "http://localhost:7474/db/data/node/7", "http://localhost:7474/db/data/node/6" ],
    "length" : 1,
    "relationships" : [ "http://localhost:7474/db/data/relationship/3" ],
    "end" : "http://localhost:7474/db/data/node/6"
  }, "you" ] ],
  "columns" : [ "path", "friend.name" ]
}
```

18.13.3. Send queries with parameters

Cypher supports queries with parameters which are submitted as a JSON map.

```
START x  = node:node_auto_index(name={STARTName})
MATCH path = (x-[r]-friend)
WHERE friend.name = {name}
RETURN TYPE(r)
```

Figure 18.41. Final Graph



Example request

- POST http://localhost:7474/db/data/ext/CypherPlugin/graphdb/execute_query
- Accept: application/json
- Content-Type: application/json

```
{"query": "start x  = node:node_auto_index(name={startName}) match path = (x-[r]-friend) where friend.name = {name} return TYPE(r)",
```

Example response

- 200: OK
- Content-Type: application/json

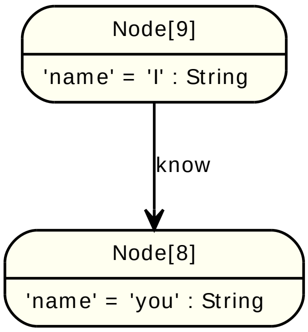
```
{
  "data" : [ [ "know" ] ],
  "columns" : [ "TYPE(r)" ]
}
```

18.13.4. Return JSON table format

The plugin can return a JSONTable representation of the results. For details, see Google Data Table Format <http://code.google.com/apis/chart/interactive/docs/reference.html#dataparam>

```
START x  = node(%I%)
MATCH path = (x--friend)
RETURN path, friend.name
```

Figure 18.42. Final Graph



Example request

- POST http://localhost:7474/db/data/ext/CypherPlugin/graphdb/execute_query
- Accept: application/json
- Content-Type: application/json

```
{"query": "start x  = node(9) match path = (x--friend) return path, friend.name","params": {}},
```

Example response

- 200: OK
- Content-Type: application/json

```
{
  "data" : [ [ {
    "start" : "http://localhost:7474/db/data/node/9",
    "nodes" : [ "http://localhost:7474/db/data/node/9", "http://localhost:7474/db/data/node/8" ],
    "length" : 1,
    "relationships" : [ "http://localhost:7474/db/data/relationship/4" ],
    "end" : "http://localhost:7474/db/data/node/8"
```

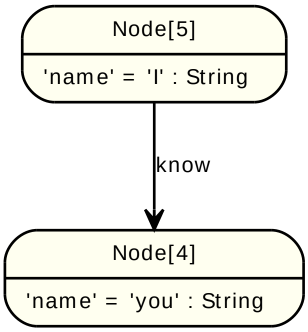
```
  }, "you" ] ],
  "columns" : [ "path", "friend.name" ]
}
```

18.13.5. Server errors

Errors on the server will be reported as a JSON-formatted stacktrace and message.

```
START x = node(%I%)
RETURN x.dummy
```

Figure 18.43. Final Graph



Example request

- POST http://localhost:7474/db/data/ext/CypherPlugin/graphdb/execute_query
- Accept: application/json
- Content-Type: application/json

```
{"query": "start x = node(5) return x.dummy","params": {}},
```

Example response

- 400: Bad Request
- Content-Type: application/json

```
{
  "message" : "dummy property not found for NodeImpl#5.",
  "exception" : "org.neo4j.graphdb.NotFoundException: dummy property not found for NodeImpl#5.",
  "stacktrace" : [ "org.neo4j.kernel.impl.core.Primitive.newPropertyNotFoundException(Primitive.java:173)", "org.neo4j.kernel.impl.co
}
```

18.14. Gremlin Plugin

Gremlin <http://gremlin.tinkerpop.com> is a Groovy based Graph Traversal Language. It provides a very expressive way of explicitly scripting traversals through a Neo4j graph.

The Neo4j Gremlin Plugin provides an endpoint to send Gremlin scripts to the Neo4j Server. The scripts are executed on the server database and the results are returned as Neo4j Node and Relationship representations. This keeps the types throughout the REST API consistent. The results are quite verbose when returning Neo4j `Node`, `Relationship` or `Graph` representations. On the other hand, just return properties like in the Section 18.14.4, "Send a Gremlin Script - JSON encoded with table results" example for responses tailored to specific needs.

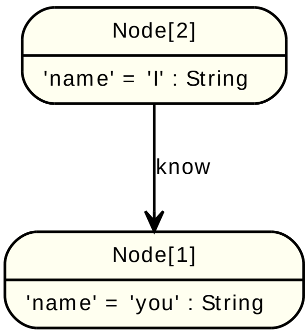
18.14.1. Send a Gremlin Script - URL encoded

Scripts can be sent as URL-encoded In this example, the graph has been autoindexed by Neo4j, so we can look up the name property on nodes.

Raw script source

```
g.idx('node_auto_index')[[name:'I']]._().out()
```

Figure 18.44. Final Graph



Example request

- `POST http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script`
- `Accept: application/json`
- `Content-Type: application/x-www-form-urlencoded`

```
script=g.idx%28%27node_auto_index%27%29%5B%5Bname%3A%27I%27%5D%5D._%28%29.out%28%29
```

Example response

- `200: OK`
- `Content-Type: application/json`

```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/1/relationships/out",
  "data" : {
    "name" : "you"
  },
  "traverse" : "http://localhost:7474/db/data/node/1/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/1/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/1",
  "properties" : "http://localhost:7474/db/data/node/1/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/1/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/1/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/1/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/1/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/1/relationships/in/{-list|&|types}"
} ]
```

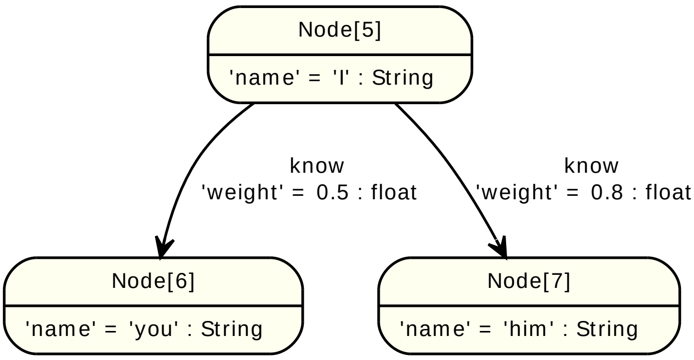

18.14.2. Load a sample graph

Import a graph form a GraphML <http://graphml.graphdrawing.org/> file can be achieved through the Gremlin GraphMLReader. The following script imports a small GraphML file from an URL into Neo4j, resulting in the depicted graph. It then returns a list of all nodes in the graph.

Raw script source

```
g.loadGraphML('https://raw.github.com/neo4j/gremlin-plugin/master/src/data/graphml1.xml')
g.V
```

Figure 18.45. Final Graph



Example request

- POST `http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script`
- Accept: `application/json`
- Content-Type: `application/json`

```
{"script": "g.loadGraphML('https://raw.github.com/neo4j/gremlin-plugin/master/src/data/graphml1.xml');g.V;","params": {}},
```

Example response

- 200: OK
- Content-Type: `application/json`

```
[ {
```

```
  "outgoing_relationships" : "http://localhost:7474/db/data/node/5/relationships/out",
  "data" : {
    "name" : "I"
  },
  "traverse" : "http://localhost:7474/db/data/node/5/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/5/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/5/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/5",
  "properties" : "http://localhost:7474/db/data/node/5/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/5/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/5/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/5/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/5/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/5/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/5/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/6/relationships/out",
  "data" : {
    "name" : "you"
  },
  "traverse" : "http://localhost:7474/db/data/node/6/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/6/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/6/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/6",
  "properties" : "http://localhost:7474/db/data/node/6/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/6/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/6/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/6/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/6/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/6/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/6/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/7/relationships/out",
  "data" : {
    "name" : "him"
  },
  "traverse" : "http://localhost:7474/db/data/node/7/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/7/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/7/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/7",
  "properties" : "http://localhost:7474/db/data/node/7/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/7/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/7/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/7/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/7/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/7/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/7/relationships/in/{-list|&|types}"
} ]
```

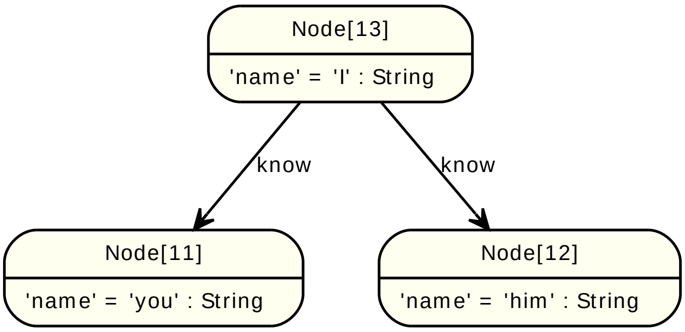

18.14.3. Sort a result using raw Groovy operations

The following script returns a sorted list of all nodes connected via outgoing relationships to node 1, sorted by their `name`-property.

Raw script source

```
g.idx('node_auto_index').get('name','I').toList()._().out().sort{it.name}.toList()
```

Figure 18.46. Final Graph



Example request

- POST `http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script`
- Accept: `application/json`
- Content-Type: `application/json`

```
{"script": "g.idx('node_auto_index').get('name','I').toList()._().out().sort{it.name}.toList()","params": {}},
```

Example response

- 200: OK
- Content-Type: `application/json`


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/12/relationships/out",
  "data" : {
    "name" : "him"
  },
  "traverse" : "http://localhost:7474/db/data/node/12/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/12/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/12/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/12",
  "properties" : "http://localhost:7474/db/data/node/12/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/12/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/12/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/12/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/12/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/12/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/12/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/11/relationships/out",
  "data" : {
    "name" : "you"
  },
  "traverse" : "http://localhost:7474/db/data/node/11/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/11/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/11/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/11",
  "properties" : "http://localhost:7474/db/data/node/11/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/11/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/11/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/11/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/11/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/11/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/11/relationships/in/{-list|&|types}"
```

```
} ]
```

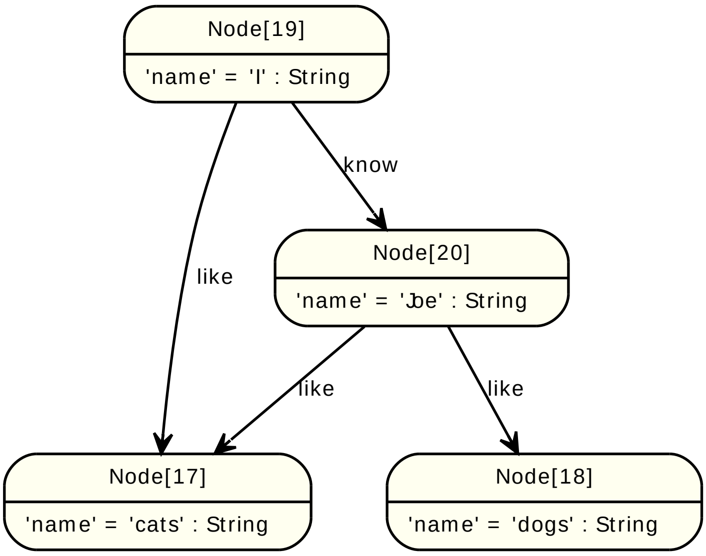
18.14.4. Send a Gremlin Script - JSON encoded with table results

To send a Script JSON encoded, set the payload Content-Type Header. In this example, find all the things that my friends like, and return a table listing my friends by their name, and the names of the things they like in a table with two columns, ignoring the third named step variable `I`. Remember that everything in Gremlin is an iterator - in order to populate the result table `t`, iterate through the pipes with `>> -1`.

Raw script source

```
i = g.v(%I%)
t= new Table()
i.as('I').out('know').as('friend').out('like').as('likes').table(t,['friend','likes']){it.name}{it.name} >> -1
t
```

Figure 18.47. Final Graph



Example request

- `POST http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script`
- `Accept: application/json`
- `Content-Type: application/json`


```
{"script": "i = g.v(19);t= new Table();i.as('I').out('know').as('friend').out('like').as('likes').table(t,['friend','likes']){it.name
```


Example response

- `200: OK`
- `Content-Type: application/json`


```
{
  "data" : [ [ "Joe", "cats" ], [ "Joe", "dogs" ] ],
  "columns" : [ "friend", "likes" ]
}
```


18.14.5. Set script variables

To set variables in the bindings for the Gremlin Script Engine on the server, you can include a `params` parameter with a String representing a JSON map of variables to set to initial values. These can then be accessed as normal variables within the script.

Raw script source

```
meaning_of_life
```

Figure 18.48. Final Graph

Example request

- POST `http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script`
- Accept: `application/json`
- Content-Type: `application/json`

```
{
  "script":"meaning_of_life",
  "params":{
    "meaning_of_life" : 42.0
  }
}
```

Example response

- 200: OK
- Content-Type: `application/json`

```
42.0
```

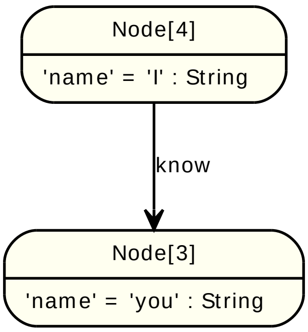
18.14.6. Send a Gremlin Script with variables in a JSON Map

Send a Gremlin Script, as JSON payload and additional parameters

Raw script source

```
g.v(me).out
```

Figure 18.49. Final Graph



Example request

- POST `http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script`
- Accept: `application/json`
- Content-Type: `application/json`

```
{"script": "g.v(me).out","params": {"me":"4"}},
```

Example response

- 200: OK
- Content-Type: `application/json`

```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/3/relationships/out",
  "data" : {
    "name" : "you"
  },
  "traverse" : "http://localhost:7474/db/data/node/3/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/3/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/3/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/3",
  "properties" : "http://localhost:7474/db/data/node/3/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/3/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/3/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/3/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/3/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/3/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/3/relationships/in/{-list|&|types}"
} ]
```

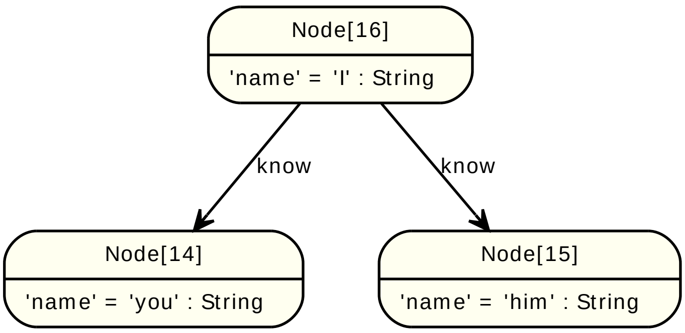

18.14.7. Return paths from a Gremlin script

The following script returns a sorted list of all nodes connected via outgoing relationships to node 1, sorted by their `name`-property.

Raw script source

```
g.v(%I%).out.name.paths
```

Figure 18.50. Final Graph



Example request

- POST `http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script`
- `Accept: application/json`
- `Content-Type: application/json`

```
{"script": "g.v(16).out.name.paths","params": {}},
```

Example response

- `200: OK`
- `Content-Type: application/json`

```
[ "[v[16], v[14], you]", "[v[16], v[15], him]" ]
```

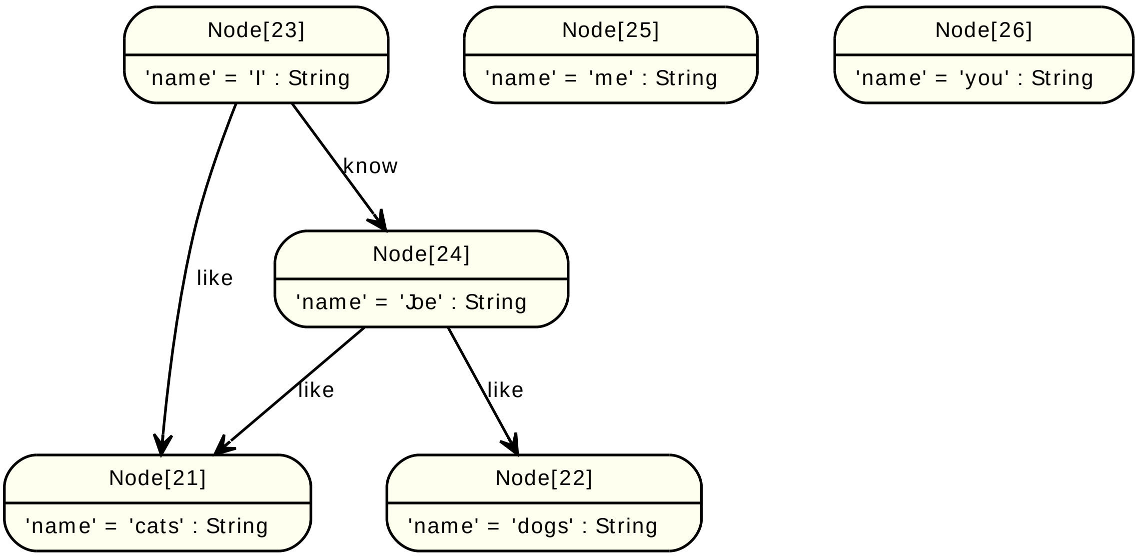
18.14.8. Send an arbitrary Groovy script - Lucene sorting

This example demonstrates that you via the Groovy runtime embedded with the server have full access to all of the servers Java APIs. The below example creates Nodes in the database both via the Blueprints and the Neo4j API indexes the nodes via the native Neo4j Indexing API constructs a custom Lucene sorting and searching returns a Neo4j IndexHits result iterator.

Raw script source

```
import org.neo4j.graphdb.index.*
import org.neo4j.index.lucene.*
import org.apache.lucene.search.*
neo4j = g.getRawGraph()
tx = neo4j.beginTx()
meVertex = g.addVertex([name:'me'])
meNode = meVertex.getRawVertex()
youNode = neo4j.createNode()
youNode.setProperty('name','you')
idxManager = neo4j.index()
personIndex = idxManager.forNodes('persons')
personIndex.add(meNode,'name',meVertex.name)
personIndex.add(youNode,'name',youNode.getProperty('name'))
tx.success()
tx.finish()
query = new QueryContext( 'name:*' ).sort( new Sort(new SortField( 'name',SortField.STRING, true ) ) )
results = personIndex.query( query )
```

Figure 18.51. Final Graph



Example request

- POST http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script
- Accept: application/json
- Content-Type: application/json

```
{"script": "import org.neo4j.graphdb.index.*;import org.neo4j.index.lucene.*;import org.apache.lucene.search.*;neo4j = g.getRawGraph
```

Example response

- 200: OK
- Content-Type: application/json

```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/26/relationships/out",
  "data" : {
    "name" : "you"
```

```
  },
  "traverse" : "http://localhost:7474/db/data/node/26/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/26/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/26/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/26",
  "properties" : "http://localhost:7474/db/data/node/26/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/26/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/26/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/26/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/26/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/26/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/26/relationships/in/{-list|&|types}"
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/25/relationships/out",
  "data" : {
    "name" : "me"
  },
  "traverse" : "http://localhost:7474/db/data/node/25/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/25/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/25/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/25",
  "properties" : "http://localhost:7474/db/data/node/25/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/25/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/25/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/25/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/25/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/25/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/25/relationships/in/{-list|&|types}"
} ]
```

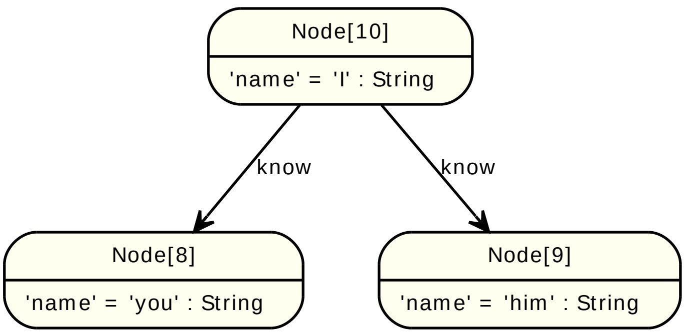

18.14.9. Emit a sample graph

Exporting a graph can be done by simple emitting the appropriate String.

Raw script source

```
writer = new GraphMLWriter(g)
out = new java.io.ByteArrayOutputStream()
writer.outputGraph(out)
result = out.toString()
```

Figure 18.52. Final Graph



Example request

- POST `http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script`
- `Accept: application/json`

- `Content-Type: application/json`

```
{"script": "writer = new GraphMLWriter(g);out = new java.io.ByteArrayOutputStream();writer.outputGraph(out);result = out.toString();
```

Example response

- `200: OK`
- `Content-Type: application/json`


```
"<?xml version=\"1.0\" ?><graphml xmlns=\"http://graphml.graphdrawing.org/xmlns\"><key id=\"name\" for=\"node\" attr.name=\"name\"
```


18.14.10. HyperEdges - find user roles in groups

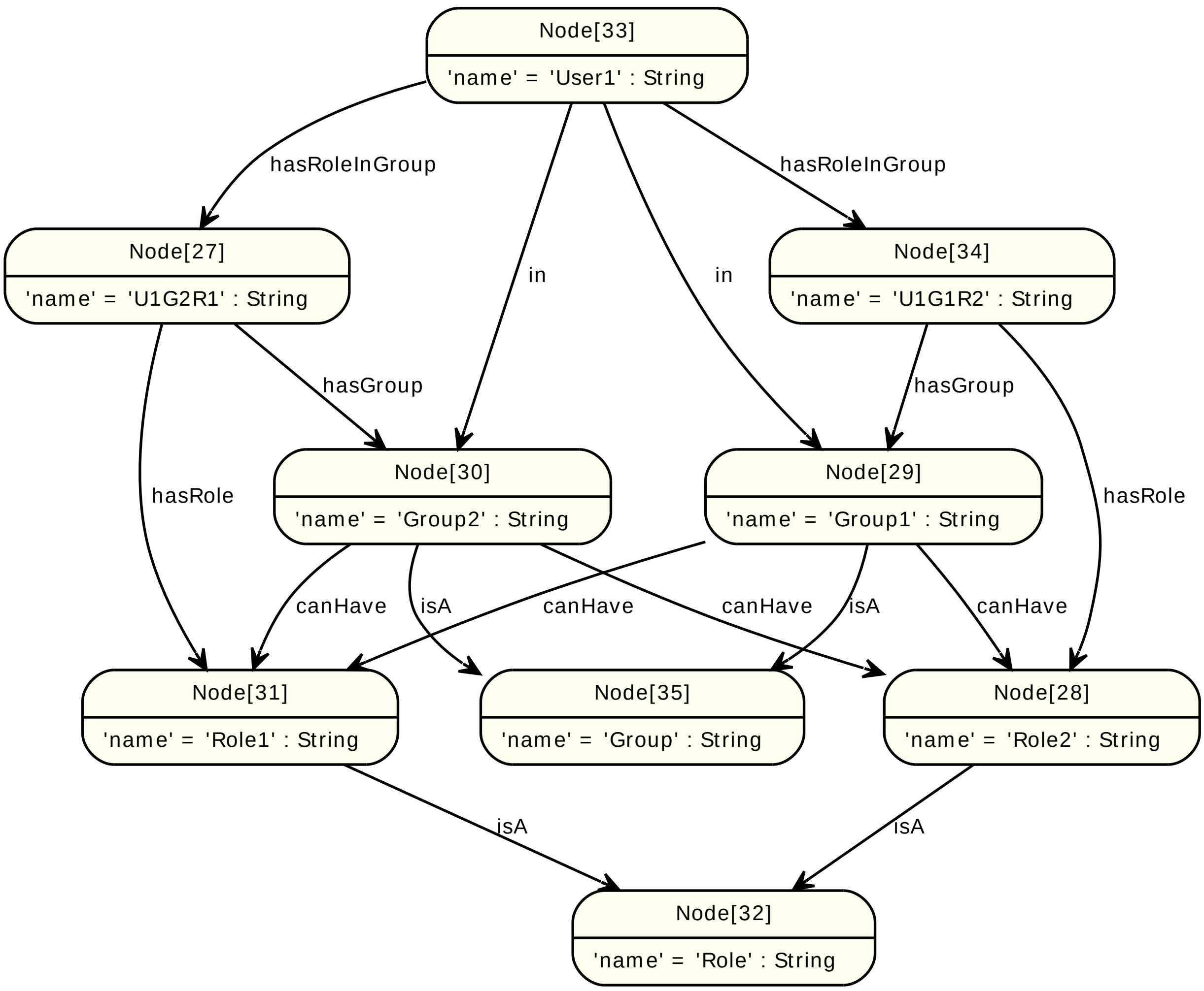
Imagine a user being part of different groups. A group can have different roles, and a user can be part of different groups. He also can have different roles in different groups apart from the membership. The association of a User, a Group and a Role can be referred to as a *HyperEdge*. However, it can be easily modeled in a property graph as a node that captures this n-ary relationship, as depicted below in the `U1G2R1` node.

To find out in what roles a user is for a particular groups (here *Group2*), the following script can traverse this HyperEdge node and provide answers.

Raw script source

```
g.v(%User1%).out('hasRoleInGroup').as('hyperedge').out('hasGroup').filter{it.name=='Group2'}.back('hyperedge').out('hasRole').name
```

Figure 18.53. Final Graph



Example request

- POST http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script
- Accept: application/json
- Content-Type: application/json

```
{"script": "g.v(33).out('hasRoleInGroup').as('hyperedge').out('hasGroup').filter{it.name=='Group2'}.back('hyperedge').out('hasRole')
```

Example response

- 200: OK
- Content-Type: application/json

```
[ "Role1" ]
```

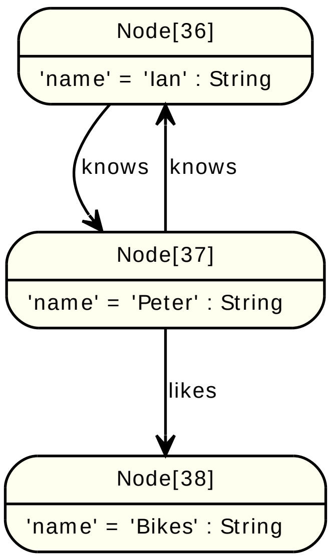
18.14.11. Group count

This example is showing a group count in Germlin, for instance the counting of the different relationship types connected to some the start node. The result is collected into a variable that then is returned.

Raw script source

```
m = [:]
g.v(%Peter%).bothE().label.groupCount(m) >> -1
m
```

Figure 18.54. Final Graph



Example request

- POST http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script
- Accept: application/json
- Content-Type: application/json

```
{"script": "m = [:];g.v(37).bothE().label.groupCount(m) >> -1;m","params": {}},
```

Example response

- 200: OK
- Content-Type: application/json

```
"{knows=2, likes=1}"
```

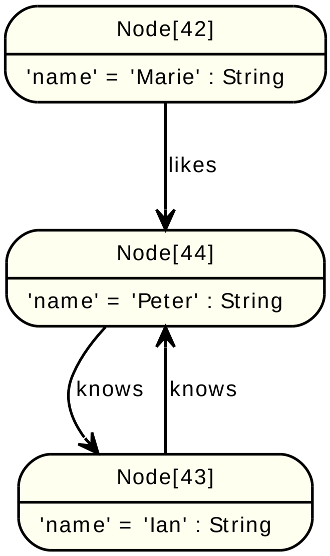
18.14.12. Collect multiple traversal results

Multiple traversals can be combined into a single result, using splitting and merging pipes in a lazy fashion.

Raw script source

```
g.idx('node_auto_index')[['name':'Peter']].copySplit(_().out('knows'), _().in('likes')).fairMerge.name
```

Figure 18.55. Final Graph



Example request

- POST http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script
- Accept: application/json
- Content-Type: application/json

```
{"script": "g.idx('node_auto_index')[['name':'Peter']].copySplit(_().out('knows'), _().in('likes')).fairMerge.name","params": {}},
```

Example response

- 200: OK
- Content-Type: application/json

```
[ "Ian", "Marie" ]
```

18.14.13. Collaborative filtering

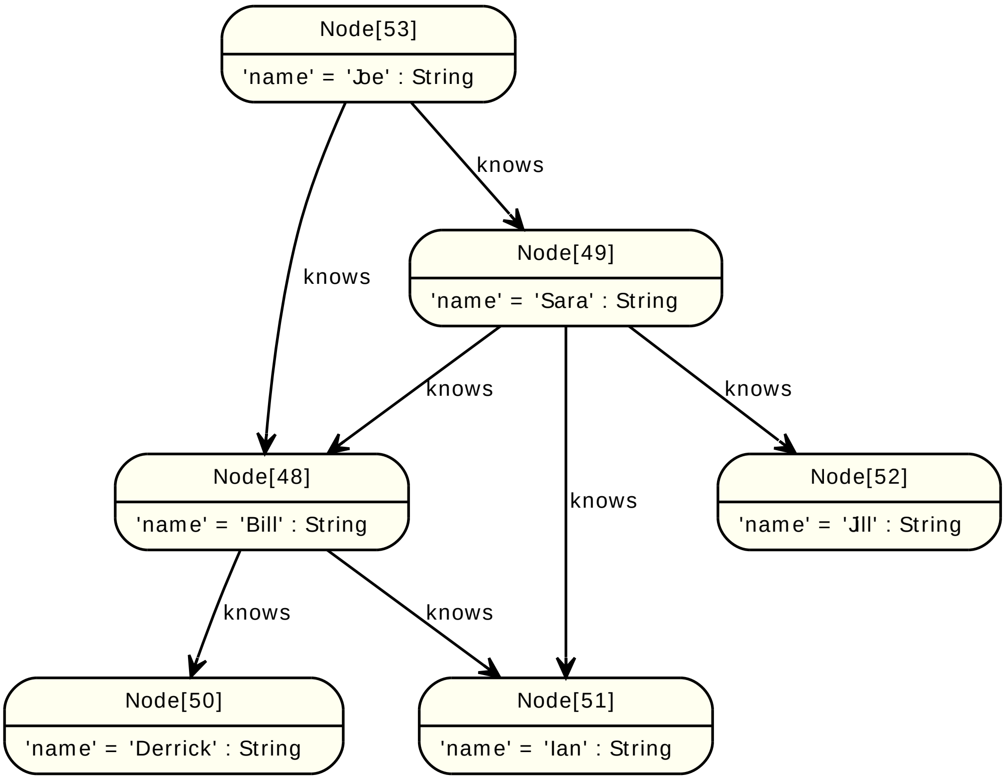
This example demonstrates basic collaborative filtering - ordering a traversal after occurence counts and substracting objects that are not interesting in the final result.

Here, we are finding Friends-of-Friends that are not Joes friends already. The same can be applied to graphs of users that `LIKE` things and others.

Raw script source

```
x=[]
fof=[:]
g.v(%Joe%).out('knows').aggregate(x).out('knows').except(x).groupCount(fof)>>-1
fof.sort{a,b -> b.value <=> a.value}
```

Figure 18.56. Final Graph



Example request

- POST `http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script`
- Accept: `application/json`
- Content-Type: `application/json`

```
{"script": "x=[];fof=[:];g.v(53).out('knows').aggregate(x).out('knows').except(x).groupCount(fof)>>-1;fof.sort{a,b -> b.value <=> a.v
```

Example response

- 200: OK
- Content-Type: `application/json`

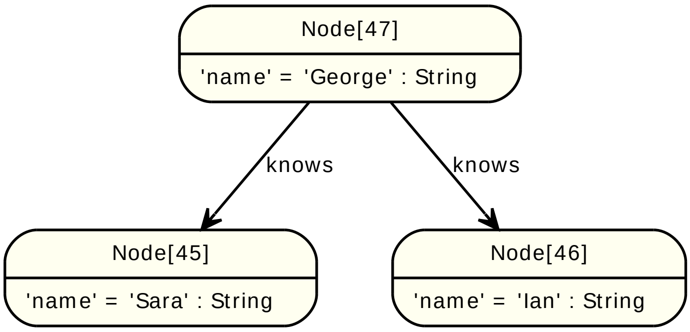
```
"{v[51]=2, v[50]=1, v[52]=1}"
```

18.14.14. Chunking and offsetting in Gremlin

Raw script source

```
g.v(%George%).outE[[label:'knows']].inV.filter{ it.name == 'Sara'}.drop(0).take(100)._()
```

Figure 18.57. Final Graph



Example request

- POST http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script
- Accept: application/json
- Content-Type: application/json

```
{"script": " g.v(47).outE[[label:'knows']].inV.filter{ it.name == 'Sara'}.drop(0).take(100)._()","params": {}},
```

Example response

- 200: OK
- Content-Type: application/json


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/45/relationships/out",
  "data" : {
    "name" : "Sara"
  },
  "traverse" : "http://localhost:7474/db/data/node/45/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/45/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/45/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/45",
  "properties" : "http://localhost:7474/db/data/node/45/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/45/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/45/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/45/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/45/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/45/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/45/relationships/in/{-list|&|types}"
} ]
```

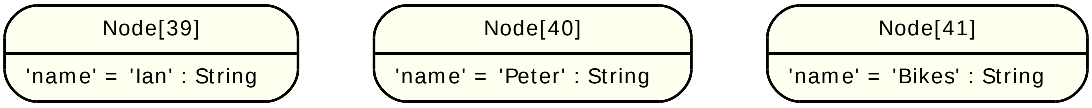

18.14.15. Modify the graph while traversing

This example is showing a group count in Germlin, for instance the counting of the different relationship types connected to some the start node. The result is collected into a variable that then is returned.

Raw script source

```
g.v(%Peter%).bothE()each{g.removeEdge(it)
}
```

Figure 18.58. Final Graph



Example request

- POST `http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script`
- Accept: `application/json`
- Content-Type: `application/json`

```
{"script": "g.v(40).bothE()each{g.removeEdge(it);};","params": {}},
```

Example response

- 200: OK
- Content-Type: `application/json`

```
[ ]
```

Chapter 19. 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:

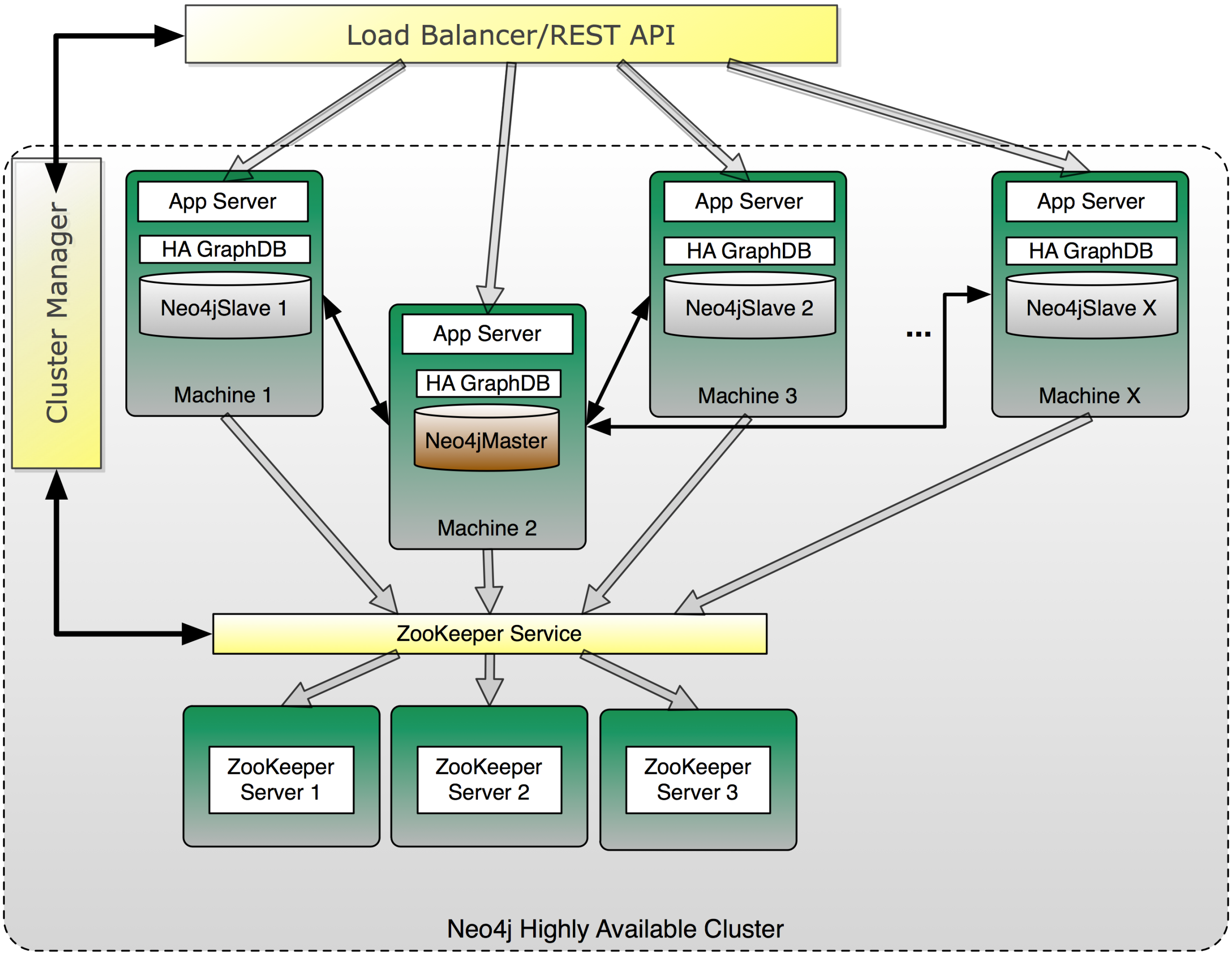
1. It enables a *fault-tolerant database architecture*, where several Neo4j slave databases can be configured to be exact replicas of a single Neo4j master database. This allows the end-user system to be fully functional and both read and write to the database in the event of hardware failure.
2. It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

19.1. 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 19.1. 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.

19.2. 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 coordinator service for initial master election, new master election (current master failing) and to publish general status information about the current Neo4j HA cluster (for example when a server joined or left the cluster). Read operations through the `GraphDatabaseService` API will always work and even writes can survive coordinator 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:

19.2.1. Small

- 3 physical (or virtual) machines
- 1 Coordinator 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 coordinator instances are running. Since the coordinator and Neo4j HA are running together on each machine this will in most scenarios mean that only one server is allowed to go down.

19.2.2. Medium

- 5-7+ machines
- Coordinator 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 coordinator and Neo4j HA). The fault tolerance will depend on how many machines there are that are running coordinators. With 3 coordinators the cluster can survive one coordinator going down, with 5 it can survive 2 and with 7 it can handle 3 coordinators 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 coordinator must be available).

19.2.3. Large

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

In this setup all coordinators are running on separate machines as a dedicated service. The dedicated coordinator cluster can handle half of the instances, minus 1, going down. The Neo4j HA cluster will

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

be able to operate with at least a single live machine. Adding more Neo4j HA instances is very easy in this setup since the coordinator cluster is operating as a separate service.

19.2.4. 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.

HighlyAvailableGraphDatabase configuration parameters

Parameter Name	Value	Example value	Required?
`ha.server_id`	integer >= 0	`1`	yes
`ha.server`	(auto-discovered) host & port to bind when acting as master	`my-domain.com:6001`	no
`ha.coordinators`	comma delimited coordinator connections	`localhost:2181, localhost:2182, localhost:2183`	yes
`ha.pull_interval`	interval for polling master from a slave, in seconds	`30`	no
`ha. slave_coordinator_update_mode`	creates a slave-only instance that will never become a master	`none`	no

Caution

Neo4j's HA setup depends on ZooKeeper a.k.a. Coordinator which makes certain assumptions about the state of the underlying operating system. In particular ZooKeeper expects that the system time on each machine is set correctly, synchronized with respect to each other. If this is not true, then Neo4j HA will appear to misbehave, caused by seemingly random ZooKeeper hiccups.

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

19.3. How Neo4j HA operates

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

On startup a Neo4j HA instance will connect to the coordinator service (ZooKeeper) 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.

You can make a database instance permanently slave-only by including the `ha.slave_coordinator_update_mode=none` configuration parameter in its configuration. Such instances will never become a master during fail-over elections though otherwise they behave identically to any other slaves, including the ability to write-through permanent slaves to the master.

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.

Whenever a server running a neo4j database becomes unavailable the coordinator 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.

19.4. High Availability setup tutorial

This is a guide to set up a Neo4j HA cluster and run embedded Neo4j or Neo4j Server instances participating as cluster nodes.

19.4.1. Set up a Coordinator cluster

The members of the HA cluster (see Chapter 19, *High Availability*) use a Coordinator cluster to manage themselves and coordinate lifecycle activity like electing a master. When running an Neo4j HA cluster, a Coordinator cluster is used for cluster collaboration and must be installed and configured before working with the Neo4j database HA instances.

Tip

Neo4j Server (see Chapter 17, *Neo4j Server*) and Neo4j Embedded (see Section 11.1, "Introduction") can both be used as nodes in the same HA cluster. This opens for scenarios where one application can insert and update data via a Java or JVM language based application, and other instances can run Neo4j Server and expose the data via the REST API (Chapter 18, *REST API*).

Below, there will be 3 coordinator instances set up on one local machine.

Download and unpack Neoj4 Enterprise

Download and unpack three installations of Neo4j Enterprise (called `$NEO4J_HOME1`, `$NEO4J_HOME2`, `$NEO4J_HOME3`) from the Neo4j download site <http://neo4j.org/download>.

Setup and start the Coordinator cluster

Now, in the *NEO4J_HOME1/conf/coord.cfg* file, adjust the coordinator `clientPort` and let the coordinator search for other coordinator cluster members at the localhost port ranges:

```
#$NEO4J_HOME1/conf/coord.cfg

server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890

clientPort=2181
```

The other two config files in `$NEO4J_HOME2` and `$NEO4J_HOME3` will have a different `clienttPort` set but the other parameters identical to the first one:

```
#$NEO4J_HOME2/conf/coord.cfg
...
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
...
clientPort=2182
```

```
#$NEO4J_HOME2/conf/coord.cfg
...
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
...
clientPort=2183
```

Next we need to create a file in each data directory called "myid" that contains an id for each server equal to the number in `server.1`, `server.2` and `server.3` from the configuration files.

```
neo4j_home1$ echo '1' > data/coordinator/myid
neo4j_home2$ echo '2' > data/coordinator/myid
neo4j_home3$ echo '3' > data/coordinator/myid
```

We are now ready to start the Coordinator instances:

```
neo4j_home1$ ./bin/neo4j-coordinator start
neo4j_home2$ ./bin/neo4j-coordinator start
neo4j_home3$ ./bin/neo4j-coordinator start
```

Start the Neo4j Servers in HA mode

In your *conf/neo4j.properties* file, enable HA by setting the necessary parameters for all 3 installations, adjusting the `ha.server_id` for all instances:

```
#$NEO4J_HOME1/conf/neo4j.properties
#unique server id for this graph database
#can not be negative id and must be unique
ha.server_id = 1

#ip and port for this instance to bind to
ha.server = localhost:6001

#connection information to the coordinator cluster client ports
ha.coordinators = localhost:2181,localhost:2182,localhost:2183
```

```
#$NEO4J_HOME2/conf/neo4j.properties
#unique server id for this graph database
#can not be negative id and must be unique
ha.server_id = 2

#ip and port for this instance to bind to
ha.server = localhost:6001

#connection information to the coordinator cluster client ports
ha.coordinators = localhost:2181,localhost:2182,localhost:2183
```

```
#$NEO4J_HOME3/conf/neo4j.properties
#unique server id for this graph database
#can not be negative id and must be unique
ha.server_id = 3

#ip and port for this instance to bind to
ha.server = localhost:6001

#connection information to the coordinator cluster client ports
ha.coordinators = localhost:2181,localhost:2182,localhost:2183
```

To avoid port clashes when starting the servers, adjust the ports for the REST endpoints in all instances under *conf/neo4j-server.properties* and enable HA mode:

```
#$NEO4J_HOME1/conf/neo4j-server.properties
...
# http port (for all data, administrative, and UI access)
org.neo4j.server.webserver.port=7474
...
# Allowed values:
# HA - High Availability
# SINGLE - Single mode, default.
# To run in High Availability mode, configure the coord.cfg file, and the
# neo4j.properties config file, then uncomment this line:
org.neo4j.server.database.mode=HA
```

```
#$NEO4J_HOME2/conf/neo4j-server.properties
...
```

```
# http port (for all data, administrative, and UI access)
org.neo4j.server.webserver.port=7475
...
# Allowed values:
# HA - High Availability
# SINGLE - Single mode, default.
# To run in High Availability mode, configure the coord.cfg file, and the
# neo4j.properties config file, then uncomment this line:
org.neo4j.server.database.mode=HA
```

```
#$NEO4J_HOME3/conf/neo4j-server.properties
...
# http port (for all data, administrative, and UI access)
org.neo4j.server.webserver.port=7476
...
# Allowed values:
# HA - High Availability
# SINGLE - Single mode, default.
# To run in High Availability mode, configure the coord.cfg file, and the
# neo4j.properties config file, then uncomment this line:
org.neo4j.server.database.mode=HA
```

To avoid JMX port clashes adjust the assigned ports for all instances under *conf/neo4j-wrapper.properties*:

```
#$NEO4J_HOME1/conf/neo4j-wrapper.properties
...
# Remote JMX monitoring, adjust the following lines if needed.
# Also make sure to update the jmx.access and jmx.password files with appropriate permission roles and passwords,
# the shipped configuration contains only a read only role called 'monitor' with password 'Neo4j'.
# For more details, see: http://download.oracle.com/javase/6/docs/technotes/guides/management/agent.html
wrapper.java.additional.4=-Dcom.sun.management.jmxremote.port=3637
...
```

```
#$NEO4J_HOME2/conf/neo4j-wrapper.properties
...
# Remote JMX monitoring, adjust the following lines if needed.
# Also make sure to update the jmx.access and jmx.password files with appropriate permission roles and passwords,
# the shipped configuration contains only a read only role called 'monitor' with password 'Neo4j'.
# For more details, see: http://download.oracle.com/javase/6/docs/technotes/guides/management/agent.html
wrapper.java.additional.4=-Dcom.sun.management.jmxremote.port=3638
...
```

```
#$NEO4J_HOME3/conf/neo4j-server.properties
...
# Remote JMX monitoring, adjust the following lines if needed.
# Also make sure to update the jmx.access and jmx.password files with appropriate permission roles and passwords,
# the shipped configuration contains only a read only role called 'monitor' with password 'Neo4j'.
# For more details, see: http://download.oracle.com/javase/6/docs/technotes/guides/management/agent.html
wrapper.java.additional.4=-Dcom.sun.management.jmxremote.port=3639
...
```

Now, start all three server instances.

```
neo4j_home1$ ./bin/neo4j start
neo4j_home2$ ./bin/neo4j start
neo4j_home3$ ./bin/neo4j start
```

Now, you should be able to access the 3 servers (the first one being elected as master since it was started first) at http://localhost:7474/webadmin/#/info/org.neo4j/High%20Availability/, http://localhost:7475/webadmin/#/info/org.neo4j/High%20Availability/ and http://localhost:7476/webadmin/#/info/org.neo4j/High%20Availability/ and check the status of the HA configuration. Alternatively, the REST API is exposing JMX, so you can check the HA JMX bean with e.g.

```
curl -H "Content-Type:application/json" -d '["org.neo4j:*"]' http://localhost:7474/db/manage/server/jmx/query
```

And find in the response

```
"description" : "Information about all instances in this cluster",
    "name" : "InstancesInCluster",
    "value" : [ {
      "description" : "org.neo4j.management.InstanceInfo",
      "value" : [ {
        "description" : "address",
        "name" : "address"
      }, {
        "description" : "instanceId",
        "name" : "instanceId"
      }, {
        "description" : "lastCommittedTransactionId",
        "name" : "lastCommittedTransactionId",
        "value" : 1
      }, {
        "description" : "machineId",
        "name" : "machineId",
        "value" : 1
      }, {
        "description" : "master",
        "name" : "master",
        "value" : true
      } ],
      "type" : "org.neo4j.management.InstanceInfo"
    }
```

Start Neo4j Embedded in HA mode

If you are using Maven and Neo4j Embedded, simply add the following dependency to your project:

```
<dependency>
   <groupId>org.neo4j</groupId>
   <artifactId>neo4j-ha</artifactId>
   <version>${neo4j-version}</version>
</dependency>
```

Where `${neo4j-version}` *is the Neo4j version used.*

If you prefer to download the jar files manually, they are included in the Neo4j distribution <http://neo4j.org/download/>.

The difference in code when using Neo4j-HA is the creation of the graph database service.

```
GraphDatabaseService db = new HighlyAvailableGraphDatabase( path, config );
```

The configuration can contain the standard configuration parameters (provided as part of the `config` above or in *neo4j.properties* but will also have to contain:

```
#HA instance1
#unique machine id for this graph database
#can not be negative id and must be unique
ha.server_id = 1

#ip and port for this instance to bind to
ha.server = localhost:6001

#connection information to the coordinator cluster client ports
ha.coordinators = localhost:2181,localhost:2182,localhost:2183

enable_remote_shell = port=1331
```

First we need to create a database that can be used for replication. This is easiest done by just starting a normal embedded graph database, pointing out a path and shutdown.

```
Map<String,String> config = HighlyAvailableGraphDatabase.loadConfigurations( configFile );
GraphDatabaseService db = new HighlyAvailableGraphDatabase( path, config );
```

We created a config file with machine id=1 and enabled remote shell. The main method will expect the path to the db as first parameter and the configuration file as the second parameter.

It should now be possible to connect to the instance using Chapter 25, *Neo4j Shell*:

```
neo4j_home1$ ./bin/neo4j-shell -port 1331
NOTE: Remote Neo4j graph database service 'shell' at port 1331
Welcome to the Neo4j Shell! Enter 'help' for a list of commands

neo4j-sh (0)$ hainfo
I'm currently master
Connected slaves:
```

Since it is the first instance to join the cluster it is elected master. Starting another instance would require a second configuration and another path to the db.

```
#HA instance2
#unique machine id for this graph database
#can not be negative id and must be unique
ha.server_id = 2

#ip and port for this instance to bind to
ha.server = localhost:6001

#connection information to the coordinator cluster client ports
ha.coordinators = localhost:2181,localhost:2182,localhost:2183

enable_remote_shell = port=1332
```

Now start the shell connecting to port 1332:

```
neo4j_home1$ ./bin/neo4j-shell -port 1332
NOTE: Remote Neo4j graph database service 'shell' at port 1332
Welcome to the Neo4j Shell! Enter 'help' for a list of commands

neo4j-sh (0)$ hainfo
I'm currently slave
```

19.5. Setting up HAProxy as a load balancer

In the Neo4j HA architecture, the cluster is typically fronted by a load balancer. In this section we will explore how to set up HAProxy to perform load balancing across the HA cluster.

19.5.1. Installing HAProxy

For this tutorial we will assume a Linux environment. We will also be installing HAProxy from source, and we'll be using version 1.4.18. You need to ensure that your Linux server has a development environment set up. On Ubuntu/apt systems, simply do:

```
aptitude install build-essential
```

And on CentOS/yum systems do:

```
yum -y groupinstall 'Development Tools'
```

Then download the tarball from the HAProxy website <http://haproxy.1wt.eu/>. Once you've downloaded it, simply build and install HAProxy:

```
tar -zvxf haproxy-1.4.18.tar.gz
cd haproxy-1.4.18
make
cp haproxy /usr/sbin/haproxy
```

19.5.2. Configuring HAProxy

HAProxy can be configured in many ways. The full documentation is available at their website.

For this example, we will configure HAProxy to load balance requests to three HA servers. Simply write the follow configuration to `/etc/haproxy.cfg`:

```
global
    daemon
    maxconn 256

defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

frontend http-in
    bind *:80
    default_backend neo4j

backend neo4j
    server s1 10.0.1.10:7474 maxconn 32
    server s2 10.0.1.11:7474 maxconn 32
    server s3 10.0.1.12:7474 maxconn 32

listen admin
    bind *:8080
    stats enable
```

HAProxy can now be started by running:

```
/usr/sbin/haproxy -f /etc/haproxy.cfg
```

You can connect to http://<ha-proxy-ip>:8080/haproxy?stats to view the status dashboard. This dashboard can be moved to run on port 80, and authentication can also be added. See the HAProxy documentation for details on this.

19.5.3. Configuring separate sets for master and slaves

It is possible to set HAProxy backends up to only include slaves or the master. For example, it may be desired to only write to slaves. To accomplish this, you need to have a small extension on the server than can report whether or not the machine is master via HTTP response codes. In this example, the extension exposes two URLs:

- `/hastatus/master`, which returns 200 if the machine is the master, and 404 if the machine is a slave
- `/hastatus/slave`, which returns 200 if the machine is a slave, and 404 if the machine is the master

The following example excludes the master from the set of machines. Request will only be sent to the slaves.

```
global
    daemon
    maxconn 256

defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

frontend http-in
    bind *:80
    default_backend neo4j-slaves

backend neo4j-slaves
    option httpchk GET /hastatus/slave
    server s1 10.0.1.10:7474 maxconn 32 check
    server s2 10.0.1.11:7474 maxconn 32 check
    server s3 10.0.1.12:7474 maxconn 32 check

listen admin
    bind *:8080
    stats enable
```

19.5.4. Cache-based sharding with HAProxy

Neo4j HA enables what is called cache-based sharding. If the dataset is too big to fit into the cache of any single machine, then by applying a consistent routing algorithm to requests, the caches on each machine will actually cache different parts of the graph. A typical routing key could be user ID.

In this example, the user ID is a query parameter in the URL being requested. This will route the same user to the same machine for each request.

```
global
    daemon
    maxconn 256

defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

frontend http-in
    bind *:80
    default_backend neo4j-slaves

backend neo4j-slaves
    balance url_param user_id
```

```
    server s1 10.0.1.10:7474 maxconn 32
    server s2 10.0.1.11:7474 maxconn 32
    server s3 10.0.1.12:7474 maxconn 32

listen admin
    bind *:8080
    stats enable
```

Naturally the health check and query parameter-based routing can be combined to only route requests to slaves by user ID. Other load balancing algorithms are also available, such as routing by source IP (`source`), the URI (`uri`) or HTTP headers(`hdr()`).

Chapter 20. Python embedded bindings

This describes *neo4j-embedded*, a Python library that lets you use the embedded Neo4j database in Python.

Apart from the reference documentation and installation instructions in this section, you may also want to take a look at Chapter 5, *Using Neo4j embedded in Python applications*.

The source code for this project lives on github: https://github.com/neo4j/python-embedded

20.1. Installation

Note

The Neo4j database itself (from the Community Edition) is included in the neo4j-embedded distribution.

20.1.1. Installation on OSX/Linux

Prerequisites

Caution

Make sure that the entire stack used is either 64bit or 32bit (no mixing, that is). That means the JVM, Python and JPype.

First, install JPype:

1. Download the latest version of JPype from http://sourceforge.net/projects/jpype/files/JPype/.
2. Unzip the file.
3. Open a console and navigate into the unzipped folder.
4. Run `sudo python setup.py install`

JPype is also available in the Debian repos:

```
sudo apt-get install python-jpype
```

Then, make sure the `JAVA_HOME` environment variable is set to your *jre* or *jdk* folder, so that JPype can find the JVM.

Installing neo4j-embedded

You can install neo4j-embedded with your python package manager of choice:

```
sudo pip install neo4j-embedded
```

```
sudo easy_install neo4j-embedded
```

Or install manually:

1. Download the latest appropriate version of JPype from http://sourceforge.net/projects/jpype/files/JPype/ for 32bit or from http://www.lfd.uci.edu/~gohlke/pythonlibs/ for 64bit.
2. Unzip the file.
3. Open a console and navigate into the unzipped folder.
4. Run `sudo python setup.py install`

20.1.2. Installation on Windows

Prerequisites

Warning

It is *imperative* that the entire stack used is either 64bit or 32bit (no mixing, that is). That means the JVM, Python, JPype and all extra DLLs (see below).

First, install JPype:

Note

Notice that JPype only works with Python 2.6 and 2.7. Also note that there are different downloads depending on which version you use.

1. Download the latest appropriate version of JPype from http://sourceforge.net/projects/jpype/files/JPype/ for 32bit or from http://www.lfd.uci.edu/~gohlke/pythonlibs/ for 64bit.
2. Run the installer.

Then, make sure the `JAVA_HOME` environment variable is set to your *jre* or *jdk* folder. There is a description of how to set environment variables in the section called “Solving problems with missing DLL files”.

Note

There may be DLL files missing from your system that are required by JPype. See the section called “Solving problems with missing DLL files” for instructions for how to fix this.

Installing neo4j-embedded

1. Download the latest version from http://pypi.python.org/pypi/neo4j-embedded/.
2. Run the installer.

Solving problems with missing DLL files

Certain versions of Windows ship without DLL files needed to programmatically launch a JVM. You will need to make `IEShims.dll` and certain debugging dlls available on Windows.

`IEShims.dll` is normally included with Internet Explorer installs. To make windows find this file globally, you need to add the IE install folder to your `PATH`.

1. Right click on "My Computer" or "Computer".
2. Select "Properties".
3. Click on "Advanced" or "Advanced system settings".
4. Click the "Environment variables" button.
5. Find the path varible, and add *C:\Program Files\Internet Explorer* to it (or the install location of IE, if you have installed it somewhere else).

Required debugging dlls are bundled with Microsoft Visual C++ Redistributable libraries.

- 32bit Windows: http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=5555
- 64bit Windows: http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=14632

If you are still getting errors about missing DLL files, you can use http://www.dependencywalker.com/ to open your `jvm.dll` (located in *JAVA_HOME/bin/client/* or *JAVA_HOME/bin/server/*), and it will tell you if there are other missing dlls.

20.2. Core API

This section describes how get get up and running, and how to do basic operations.

20.2.1. Getting started

Creating a database

```
from neo4j import GraphDatabase

# Create db
db = GraphDatabase(folder_to_put_db_in)

# Always shut down your database
db.shutdown()
```

Creating a database, with configuration

Please see Chapter 11, *Configuration & Performance* for what options you can use here.

```
from neo4j import GraphDatabase

# Example configuration parameters
db = GraphDatabase(folder_to_put_db_in, string_block_size=200, array_block_size=240)

db.shutdown()
```

JPype JVM configuration

You can set extra arguments to be passed to the JVM using the "NEO4J_PYTHON_JVMARGS" environment variable. This can be used to, for instance, increase the max memory for the database.

Note that you must set this before you import the neo4j package, either by setting it before you start python, or by setting it programatically in your app.

```
import os
os.environ['NEO4J_PYTHON_JVMARGS'] = '-Xms128M -Xmx512M'
import neo4j
```

You can also override the classpath used by neo4j-embedded, by setting the "NEO4J_PYTHON_CLASSPATH" environment variable.

20.2.2. Transactions

All write operations to the database need to be performed from within transactions. This ensures that your database never ends up in an inconsistent state.

See Chapter 13, *Transaction Management* for details on how Neo4j handles transactions.

We use the python `with` statement to define a transaction context. If you are using an older version of Python, you may have to import the `with` statement:

```
from __future__ import with_statement
```

Either way, this is how you get into a transaction:

```
# Start a transaction
with db.transaction:
    # This is inside the transactional
    # context. All work done here
    # will either entirely succeed,
    # or no changes will be applied at all.
```

```
    # Create a node
    node = db.node()

    # Give it a name
    node['name'] = 'Cat Stevens'

# The transaction is automatically
# commited when you exit the with
# block.
```

20.2.3. Nodes

This describes operations that are specific to node objects. For documentation on how to handle properties on both relationships and nodes, see Section 20.2.5, “Properties”.

Creating a node

```
with db.transaction:
    # Create a node
    thomas = db.node(name='Thomas Anderson', age=42)
```

Fetching a node by id

```
# You don't have to be in a transaction
# to do read operations.
a_node = db.node[some_node_id]
```

Fetching the reference node

```
reference = db.reference_node
```

Removing a node

```
with db.transaction:
    node = db.node()
    node.delete()
```

Tip

See also Section 13.5, “Delete semantics”.

Removing a node by id

```
with db.transaction:
    del db.node[some_node_id]
```

Accessing relationships from a node

For details on what you can do with the relationship objects, see Section 20.2.4, “Relationships”.

```
# All relationships on a node
for rel in a_node.relationships:
    pass

# Incoming relationships
for rel in a_node.relationships.incoming:
    pass

# Outgoing relationships
for rel in a_node.relationships.outgoing:
    pass
```

```
# Relationships of a specific type
for rel in a_node.mayor_of:
    pass

# Incoming relationships of a specific type
for rel in a_node.mayor_of.incoming:
    pass

# Outgoing relationships of a specific type
for rel in a_node.mayor_of.outgoing:
    pass
```

20.2.4. Relationships

This describes operations that are specific to relationship objects. For documentation on how to handle properties on both relationships and nodes, see Section 20.2.5, “Properties”.

Creating a relationship

```
with self.graphdb.transaction:
    # Nodes to create a relationship between
    steven = self.graphdb.node(name='Steve Brook')
    poplar_bluff = self.graphdb.node(name='Poplar Bluff')

    # Create a relationship of type "mayor_of"
    relationship = steven.mayor_of(poplar_bluff, since="12th of July 2012")

    # Or, to create relationship types with names
    # that would not be possible with the above
    # method.
    steven.relationships.create('mayor_of', poplar_bluff, since="12th of July 2012")
```

Fetching a relationship by id

```
the_relationship = db.relationship[a_relationship_id]
```

Removing a relationship

```
with db.transaction:
    # Create a relationship
    source = db.node()
    target = db.node()
    rel = source.Knows(target)

    # Delete it
    rel.delete()
```

Tip
See also Section 13.5, “Delete semantics”.

Removing a relationship by id

```
with db.transaction:
    del db.relationship[some_relationship_id]
```

Relationship start node, end node and type

```
relationship_type = relationship.type

start_node = relationship.start
end_node = relationship.end
```

20.2.5. Properties

Both nodes and relationships can have properties, so this section applies equally to both node and relationship objects. Allowed property values include strings, numbers, booleans, as well as arrays of those primitives. Within each array, all values must be of the same type.

Setting properties

```
with db.transaction:
    node_or_rel['name'] = 'Thomas Anderson'
    node_or_rel['age'] = 42
    node_or_rel['favourite_numbers'] = [1,2,3]
    node_or_rel['favourite_words'] = ['banana','blue']
```

Getting properties

```
numbers = node_or_rel['favourite_numbers']
```

Removing properties

```
with db.transaction:
    del node_or_rel['favourite_numbers']
```

Looping through properties

```
# Loop key and value at the same time
for key, value in node_or_rel.items():
    pass

# Loop property keys
for key in node_or_rel.keys():
    pass

# Loop property values
for value in node_or_rel.values():
    pass
```

20.2.6. Paths

A path object represents a path between two nodes in the graph. Paths thus contain at least two nodes and one relationship, but can reach arbitrary length. It is used in various parts of the API, most notably in traversals.

Accessing the start and end nodes

```
start_node = path.start
end_node = path.end
```

Accessing the last relationship

```
last_relationship = path.last_relationship
```

Looping through the entire path

You can loop through all elements of a path directly, or you can choose to only loop through nodes or relationships. When you loop through all elements, the first item will be the start node, the second will be the first relationship, the third the node that the relationship led to and so on.

```
for item in path:
    # Item is either a Relationship,
    # or a Node
    pass
```

```
for nodes in path.nodes:
    # All nodes in a path
    pass

for nodes in path.relationships:
    # All relationships in a path
    pass
```

20.3. Traversals

Traversing is a high-performance way to search your database. The traversal API used here is essentially the same as the one used in the Java API, with a few modifications.

Traversals start at a given node and uses a set of rules to move through the graph and to decide what parts of the graph to return.

20.3.1. Basic traversals

Following a relationship

The most basic traversals simply follow certain relationship types, and return everything they encounter. By default, each node is visited only once, so there is no risk of infinite loops.

```
traverser = db.traversal()\
    .relationships('related_to')\
    .traverse(start_node)

# The graph is traversed as
# you loop through the result.
for node in traverser.nodes:
    pass
```

Following a relationship in a specific direction

You can tell the traverser to only follow relationships in some specific direction.

```
from neo4j import OUTGOING, INCOMING, ANY

traverser = db.traversal()\
    .relationships('related_to', OUTGOING)\
    .traverse(start_node)
```

Following multiple relationship types

You can specify an arbitrary number of relationship types and directions to follow.

```
from neo4j import OUTGOING, INCOMING, ANY

traverser = db.traversal()\
    .relationships('related_to', INCOMING)\
    .relationships('likes')\
    .traverse(start_node)
```

20.3.2. Traversal results

A traversal can give you one of three different result types: nodes, relationships or paths.

Traversals are performed lazily, which means that the graph is traversed as you loop through the result.

```
traverser = db.traversal()\
    .relationships('related_to')\
    .traverse(start_node)

# Get each possible path
for path in traverser:
    pass

# Get each node
for node in traverser.nodes:
    pass
```

```
# Get each relationship
for relationship in traverser.relationships:
    pass
```

20.3.3. Uniqueness

To avoid infinite loops, it's important to define what parts of the graph can be re-visited during a traversal. By default, uniqueness is set to `NODE_GLOBAL`, which means that each node is only visited once.

Here are the other options that are available.

```
from neo4j import Uniqueness

# Available options are:

Uniqueness.NONE
# Any position in the graph may be revisited.

Uniqueness.NODE_GLOBAL
# Default option
# No node in the entire graph may be visited
# more than once. This could potentially
# consume a lot of memory since it requires
# keeping an in-memory data structure
# remembering all the visited nodes.

Uniqueness.RELATIONSHIP_GLOBAL
# No relationship in the entire graph may be
# visited more than once. For the same
# reasons as NODE_GLOBAL uniqueness, this
# could use up a lot of memory. But since
# graphs typically have a larger number of
# relationships than nodes, the memory
# overhead of this uniqueness level could
# grow even quicker.

Uniqueness.NODE_PATH
# A node may not occur previously in the
# path reaching up to it.

Uniqueness.RELATIONSHIP_PATH
# A relationship may not occur previously in
# the path reaching up to it.

Uniqueness.NODE_RECENT
# Similar to NODE_GLOBAL uniqueness in that
# there is a global collection of visited
# nodes each position is checked against.
# This uniqueness level does however have a
# cap on how much memory it may consume in
# the form of a collection that only
# contains the most recently visited nodes.
# The size of this collection can be
# specified by providing a number as the
# second argument to the
# uniqueness()-method along with the
# uniqueness level.

Uniqueness.RELATIONSHIP_RECENT
# works like NODE_RECENT uniqueness, but
# with relationships instead of nodes.


traverser = db.traversal()\
```

```
    .uniqueness(Uniqueness.NODE_PATH)\
    .traverse(start_node)
```

20.3.4. Ordering

You can traverse either depth first, or breadth first. Depth first is the default, because it has lower memory overhead.

```
# Depth first traversal, this
# is the default.
traverser = db.traversal()\
    .depthFirst()\
    .traverse(self.source)

# Breadth first traversal
traverser = db.traversal()\
    .breadthFirst()\
    .traverse(start_node)
```

20.3.5. Evaluators - advanced filtering

In order to traverse based on other critera, such as node properties, or more complex things like neighboring nodes or patterns, we use evaluators. An evaluator is a normal Python method that takes a path as an argument, and returns a description of what to do next.

The path argument is the current position the traverser is at, and the description of what to do can be one of four things, as seen in the example below.

```
from neo4j import Evaluation

# Evaluation contains the four
# options that an evaluator can
# return. They are:

Evaluation.INCLUDE_AND_CONTINUE
# Include this node in the result and
# continue the traversal

Evaluation.INCLUDE_AND_PRUNE
# Include this node in the result, but don't
# continue the traversal

Evaluation.EXCLUDE_AND_CONTINUE
# Exclude this node from the result, but
# continue the traversal

Evaluation.EXCLUDE_AND_PRUNE
# Exclude this node from the result and
# don't continue the traversal

# An evaluator
def my_evaluator(path):
    # Filter on end node property
    if path.end['message'] == 'world':
        return Evaluation.INCLUDE_AND_CONTINUE

    # Filter on last relationship type
    if path.last_relationship.type.name() == 'related_to':
        return Evaluation.INCLUDE_AND_PRUNE

    # You can do even more complex things here, like subtraversals.

    return Evaluation.EXCLUDE_AND_CONTINUE
```

```
# Use the evaluator
traverser = db.traversal()\
    .evaluator(my_evaluator)\
    .traverse(start_node)
```

20.4. Indexes

In order to rapidly find nodes or relationship based on properties, Neo4j supports indexing. This is commonly used to find start nodes for traversals.

By default, the underlying index is powered by Apache Lucene <http://lucene.apache.org/java/docs/index.html>, but it is also possible to use Neo4j with other index implementations.

You can create an arbitrary number of named indexes. Each index handles either nodes or relationships, and each index works by indexing key/value/object triplets, object being either a node or a relationship, depending on the index type.

20.4.1. Index management

Just like the rest of the API, all write operations to the index must be performed from within a transaction.

Creating an index

If an index already exists with the name you choose, that index will be returned, and the optional arguments you passed will be ignored.

```
with db.transaction:
    # Create a relationship index
    rel_idx = db.relationship.indexes.create('my_rels')

    # Create a node index, passing optional
    # arguments to the index provider.
    # In this case, enable full-text indexing.
    node_idx = db.node.indexes.create('my_nodes', type='fulltext')
```

Retrieving a pre-existing index

```
with db.transaction:
    node_idx = db.node.indexes.get('my_nodes')

    rel_idx = db.relationship.indexes.get('my_rels')
```

Deleting indexes

```
with db.transaction:
    node_idx = db.node.indexes.get('my_nodes')
    node_idx.delete()

    rel_idx = db.relationship.indexes.get('my_rels')
    rel_idx.delete()
```

20.4.2. Indexing things

Adding nodes or relationships to an index

```
with db.transaction:
    # Indexing nodes
    a_node = db.node()
    node_idx = db.node.indexes.create('my_nodes')

    # Add the node to the index
    node_idx['akey']['avalue'] = a_node

    # Indexing relationships
    a_relationship = a_node.knows(db.node())
    rel_idx = db.relationship.indexes.create('my_rels')
```

```
    # Add the relationship to the index
    rel_idx['akey']['avalue'] = a_relationship
```

Removing indexed items

Removing items from an index can be done at several levels of granularity. See the example below.

```
# Remove specific key/value/item triplet
del idx['akey']['avalue'][item]

# Remove all instances under a certain
# key
del idx['akey'][item]

# Remove all instances all together
del idx[item]
```

20.4.3. Searching the index

You can retrieve indexed items in two ways. Either you do a direct lookup, or you perform a query. The direct lookup is the same across different index providers while the query syntax depends on what index provider you use. As mentioned previously, Lucene is the default and by far most common index provider. For querying Lucene you will want to use the lucene query language <http://lucene.apache.org/java/3_1_0/queryparsersyntax.html>.

There is a python library for programatically generating Lucene queries, available at GitHub <https://github.com/scholrly/lucene-querybuilder>.

Important

Unless you loop through the entire index result, you have to close the result when you are done with it. If you do not, the database does not know when it can release the resources the result is taking up.

Direct lookups

```
hits = idx['akey']['avalue']
for item in hits:
    pass

# Always close index results when you are
# done, to free up resources.
hits.close()
```

Querying

```
hits = idx.query('akey:avalue')
for item in hits:
    pass

# Always close index results when you are
# done, to free up resources.
hits.close()
```

Part IV. Operations

This part 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.

Chapter 21. Backup

> **Note**
> The Backup features are only available in the Neo4j Enterprise Edition.

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 database files without acquiring any locks, allowing for continued operations on the target instance. This of course means that while copying, transactions will continue and the store will change. For this reason, the transaction that was running when the backup operation started is noted and, when the copy operation completes, all transactions from the latter down to the one happening at the end of the copy are replayed on the backup files. This ensures that the backed up data represent a consistent and up-to-date snapshot of the database storage.

In contrast, *incremental backup* does not copy store files - instead it copies the logs of the transactions that have taken place since the last full or incremental backup which are then replayed over an existing backup store. This makes incremental backups far more efficient that doing full backups every time but they also require that a *full backup* has taken place before they are executed.

Regardless of the mode a backup is created with, the resulting files represent a consistent database snapshot and they can be used to boot up a Neo4j instance.

The database to be backed up is specified using a URI with syntax

<running mode>://<host>[:port]{,<host>[:port]*}

Running mode must be defined and is either *single* for non-HA or *ha* for HA clusters. The <host>[:port] part points to a host running the database, on port *port* if not the default. The additional *host:port* arguments are useful for passing multiple coordinator instances

> **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.

21.1. Embedded and Server

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

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

# Performing an incremental backup
./neo4j-backup -incremental -from single://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 single://192.168.1.34:9999 -to /mnt/backup/neo4j-backup
```

21.2. Online Backup from Java

In order to programmatically backup your data full or subsequently incremental from a JVM based program, you need to write Java code like

```
OnlineBackup backup = OnlineBackup.from( "localhost" );
backup.full( backupPath );
backup.incremental( backupPath );
```

For more information, please see the Javadocs for OnlineBackup <http://components.neo4j.org/neo4j-enterprise/1.5-SNAPSHOT/apidocs/org/neo4j/backup/OnlineBackup.html>

21.3. High Availability

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

```
# Performing a full backup from HA cluster, specifying two possible coordinators
./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 coordinator
./neo4j-backup -incremental -from ha://192.168.1.15:2181 -to /mnt/backup/neo4j-backup
```

21.4. 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.

Chapter 22. 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.

22.1. Securing access to the Neo4j Server

22.1.1. Secure the port and remote client connection accepts

By default, the Neo4j Server is bundled with a Web server that binds to host `localhost` on port `7474`, answering only requests from the local machine.

This is configured in the `conf/neo4j-server.properties` file:

```
# http port (for all data, administrative, and UI access)
org.neo4j.server.webserver.port=7474

#let the webserver only listen on the specified IP. Default
#is localhost (only accept local connections). Uncomment to allow
#any connection.
#org.neo4j.server.webserver.address=0.0.0.0
```

If you need to enable access from external hosts, configure the Web server in the `conf/neo4j-server.properties` by setting the property `org.neo4j.server.webserver.address=0.0.0.0` to enable access from any host.

22.1.2. Server Authorization Rules

Administrators may require more fine-grained security policies in addition to IP-level restrictions on the Web server. Neo4j server supports administrators in allowing or disallowing access the specific aspects of the database based on credentials that users or applications provide.

To facilitate domain-specific authorization policies in Neo4j Server, SecurityRules can be implemented and registered with the server. This makes scenarios like user and role based security and authentication against external lookup services possible.

Enforcing Server Authorization Rules

In this example, a (dummy) failing security rule is registered to deny access to all URIs to the server by listing the rules class in `neo4j-server.properties`:

```
org.neo4j.server.rest.security_rules=my.rules.PermanentlyFailingSecurityRule
```

with the rule source code of:

```
public class PermanentlyFailingSecurityRule implements SecurityRule
{

    public static final String REALM = "WallyWorld"; // as per RFC2617 :-);

    @Override
    public boolean isAuthorized( HttpServletRequest request )
    {
        return false; // always fails - a production implementation performs
                      // deployment-specific authorization logic here
    }

    @Override
    public String forUriPath()
    {
        return SecurityRule.DEFAULT_DATABASE_PATH;
    }

    @Override
    public String wwwAuthenticateHeader()
    {
        return SecurityFilter.basicAuthenticationResponse(REALM);
```

```
    }
}
```

With this rule registered, any access to the server will be denied. In a production-quality implementation the rule will likely lookup credentials/claims in a 3rd party directory service (e.g. LDAP) or in a local database of authorized users.

Example request

- `POST http://localhost:7474/db/data/node`
- `Accept: application/json`

Example response

- `401: Unauthorized`
- `WWW-Authenticate: Basic realm="WallyWorld"`

22.1.3. Hosted Scripting

Important

The neo4j server exposes remote scripting functionality by default that allow full access to the underlying system. Exposing your server without implementing a security layer poses a substantial security vulnerability.

22.1.4. Security in Depth

Although the Neo4j server has competent security features built-in, for sensitive deployments it is often sensible to front it with a proxy like Apache `mod_proxy` [1]. This provides a number of advantages:

- 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>
```

While equivalent functionality can be implemented with Neo4j's `SecurityRule` plugins (see above), for operations professionals configuring servers like Apache is often preferable to developing plugins. However it should be noted that where both approaches are used, they will work harmoniously providing the behavior is consistent across proxy server and `SecurityRule` plugins.

- 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
```

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

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

22.1.5. Rewriting URLs with a Proxy installation

When installing Neo4j Server behind proxies, you need to enable rewriting of the JSON calls, otherwise they will point back to the servers own base URL (normally http://localhost:7474).

To do this, you can use Apache mod_substitute <http://httpd.apache.org/docs/2.2/mod/mod_substitute.html>

```
ProxyPass / http://localhost:7474/
ProxyPassReverse / http://localhost:7474/
<Location />
    AddOutputFilterByType SUBSTITUTE application/json
    AddOutputFilterByType SUBSTITUTE text/html
    Substitute s/localhost:7474/myproxy.example.com/n
    Substitute s/http/https/n
</Location>
```

Chapter 23. 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. Most of these are exposed through the JMX API <http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html>.

23.1. JMX

23.1.1. Adjusting remote JMX access to the Neo4j Server

Per default, the Advanced and Enterprise Neo4j Server editions do not allow remove JMX connections, since the relevant options in the `conf/neo4j-wrapper.conf` configuration file are commented out. To enable this feature, you have to remove the `#` characters from the various `com.sun.management.jmxremote` options there.

When commented in, the default values are set up to allow remote JMX connections with certain roles, refer to the `conf/jmx.password`, `conf/jmx.access` and `conf/wrapper.conf` for details.

Make sure that `conf/jmx.password` has the correct file permissions. The owner of the file has to be the user that will run the service, and the permissions should be read only for that user. On Unix systems, this is 0600.

On Windows, follow this tutorial to set the correct permissions. If you are running the service under the Local System Account, the user that owns the file and has access to it should be SYSTEM.

With this setup, you should be able to connect to JMX monitoring of the Neo4j server using `<IP-OF-SERVER>:3637`, with the username `monitor` and the password `Neo4j`.

Note that it is possible that you have to update the permissions and/or ownership of the `conf/jmx.password` and `conf/jmx.access` files - refer to the relevant section in `conf/wrapper.conf` for details

Warning
For maximum security, please adjust at least the password settings in `conf/jmx.password` for a production installation.

For more details, see: http://download.oracle.com/javase/6/docs/technotes/guides/management/agent.html

23.1.2. How to connect to a Neo4j instance using JMX and JConsole

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

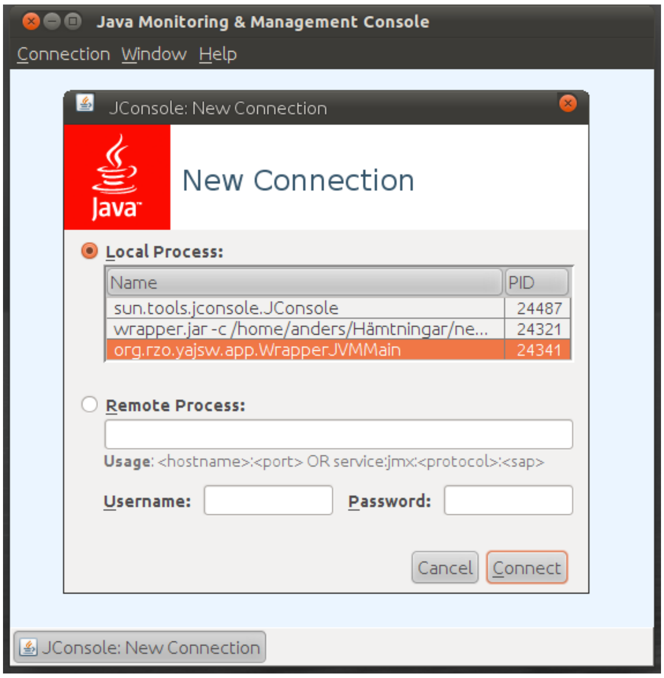
```
$NEO4j_SERVER_HOME/bin/neo4j start
```

Now, start JConsole with

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

Connect to the process running your Neo4j database instance:

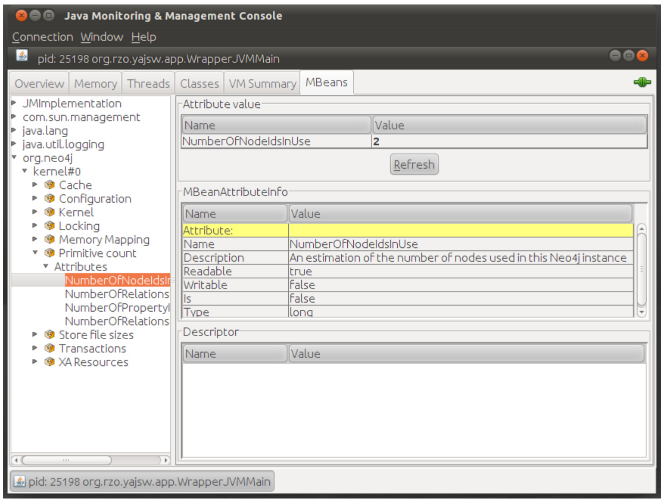
Figure 23.1. Connecting JConsole to the Neo4j Java process



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.

For opening JMX to remote monitoring access, please refer to the JMX documention <http://download.oracle.com/javase/1.5.0/docs/guide/management/agent.html#remote> and pass the `com.sun.management.jmxremote.port=portNum` or other configuration as JVM parameters to your running Java process.

Figure 23.2. Neo4j MBeans View



23.1.3. 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.

23.1.4. Reference of supported JMX MBeans

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.

MBean Memory Mapping

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

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

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

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

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
`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

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

MBean XA Resources

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

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

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

Attribute	Description	Type
`StoreCreationDate`	The time when this Neo4j graph store was created	Date
`StoreId`	An identifier that, together with store creation time, 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

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 conenced slaves in this cluster	String
`InstancesInCluster`	Information about the other Neo4j instances in this HA cluster	String

Part V. Tools

Chapter 24. Web Administration

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

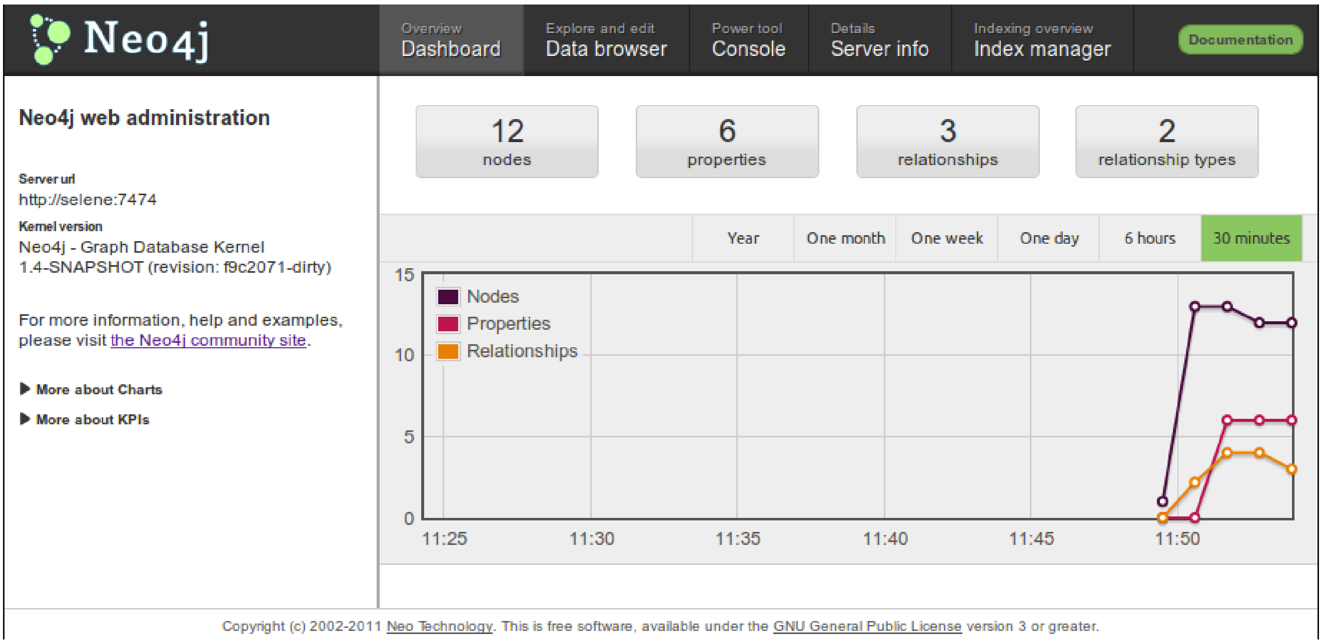
- monitor the Neo4j Server
- manipulate and browse data
- interact with the database via various consoles
- view raw data management objects (JMX MBeans)

The tool is available at http://127.0.0.1:7474/ after you have installed the Neo4j Server. To use it together with an embedded database, see Section 17.4, "Using the server (including web administration) with an embedded database".

24.1. Dashboard tab

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

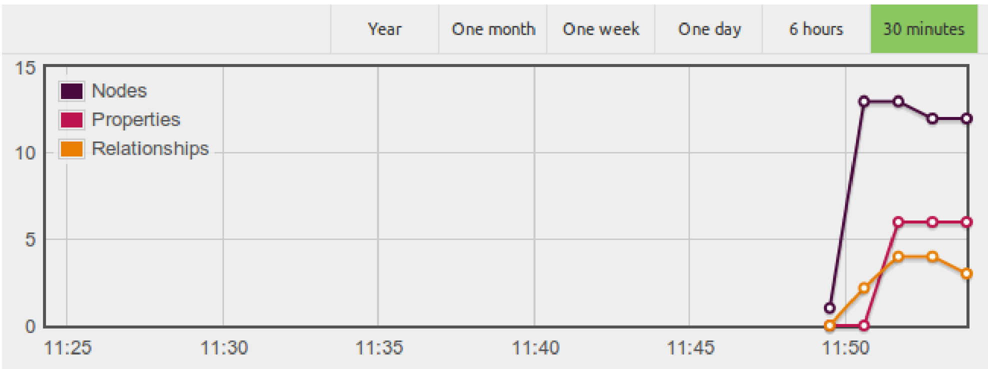
Figure 24.1. Web Administration Dashboard



24.1.1. Entity chart

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

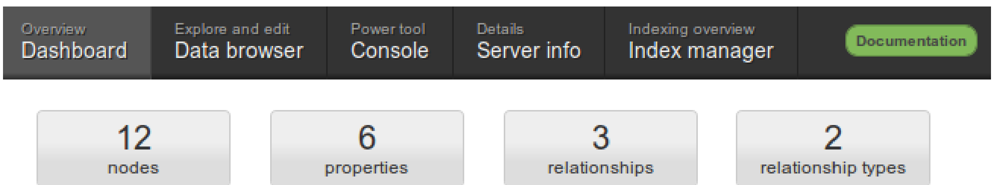
Figure 24.2. Entity charting



24.1.2. Status monitoring

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

Figure 24.3. Status indicator panels



24.2. Data tab

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

Figure 24.4. Browsing and manipulating data

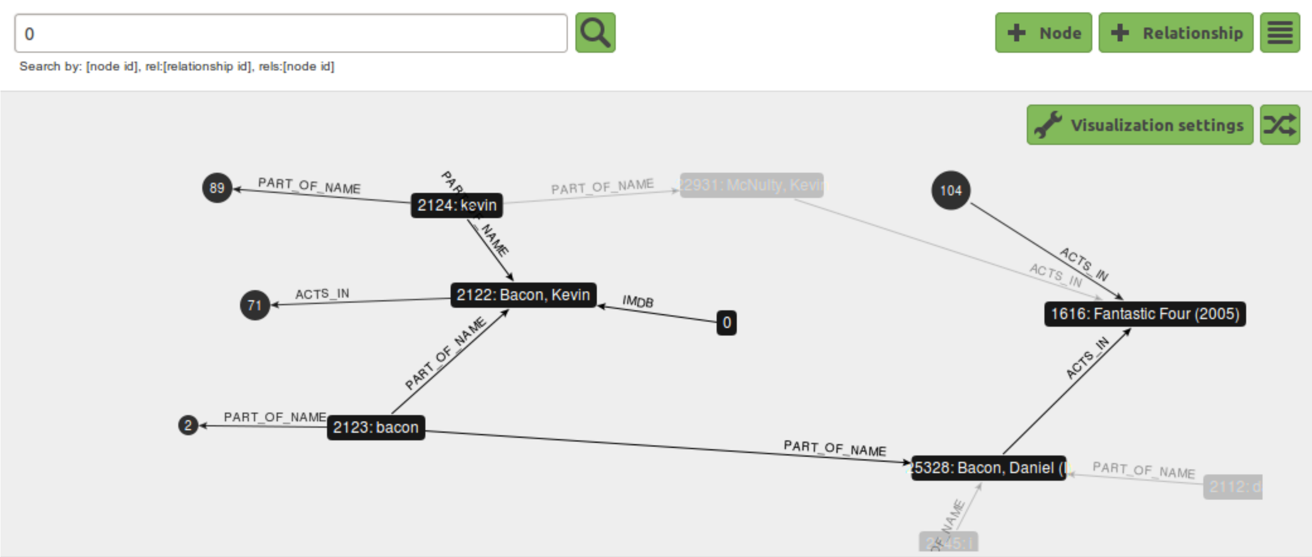
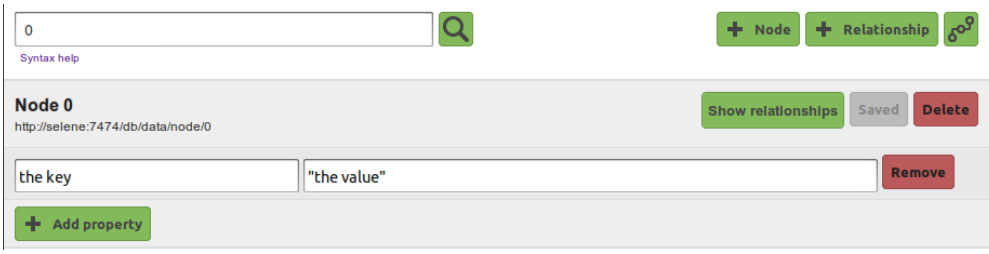


Figure 24.5. Editing properties



24.3. Console tab

The Console tab gives:

- scripting access to the database via the Gremlin <http://gremlin.tinkerpop.com> scripting engine,
- query access via Cypher,
- HTTP access via the HTTP console.

Figure 24.6. Traverse data with Gremlin

Figure 24.7. Query data with Cypher

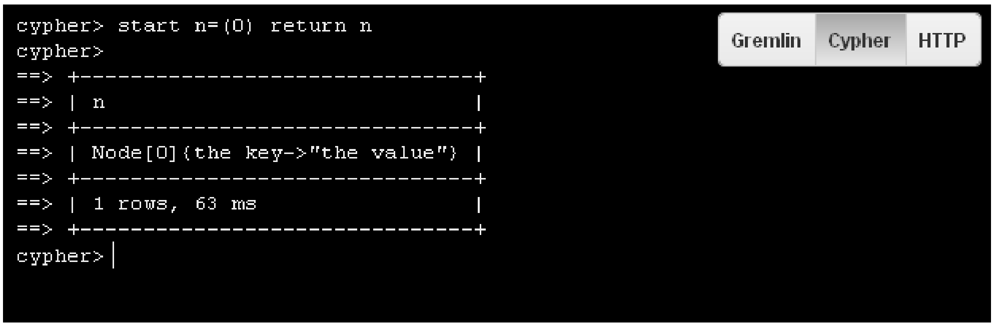
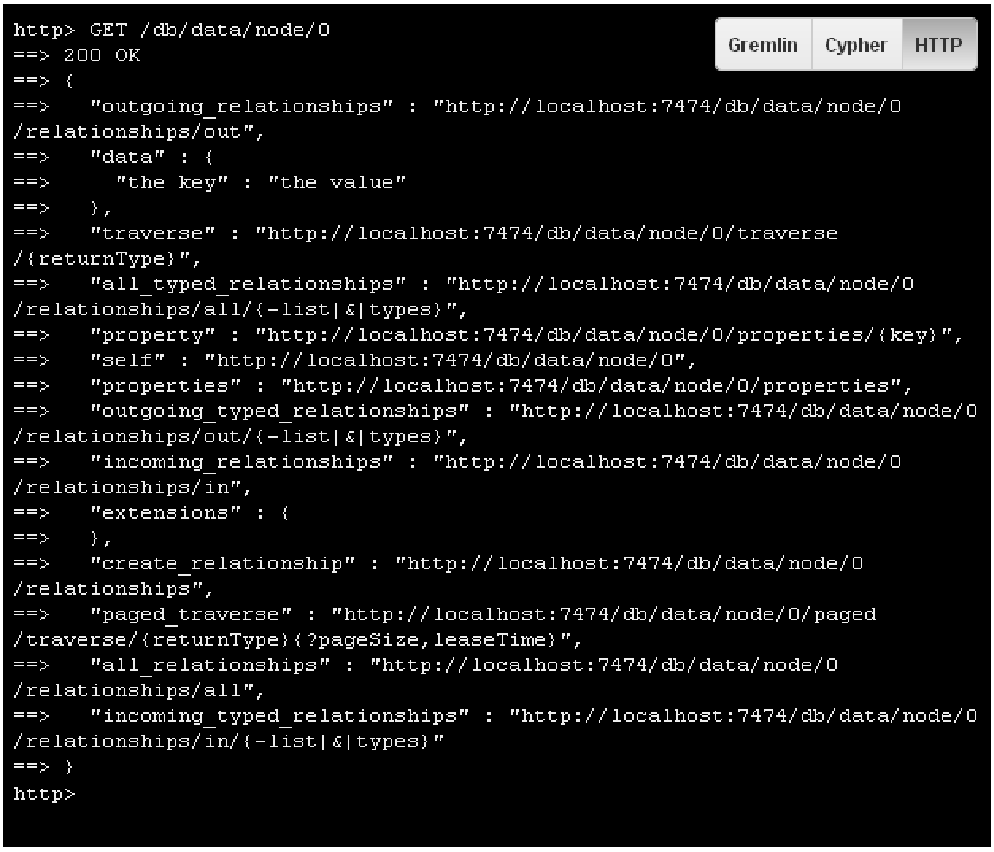


Figure 24.8. Interact over HTTP

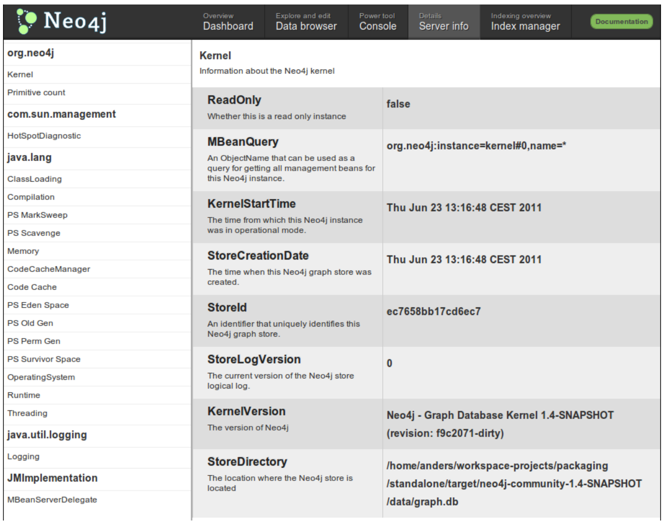



```
http> GET /db/data/node/0
==> 200 OK
==> {
==>   "outgoing_relationships" : "http://localhost:7474/db/data/node/0
/relationships/out",
==>   "data" : {
==>     "the key" : "the value"
==>   },
==>   "traverse" : "http://localhost:7474/db/data/node/0/traverse
/{returnType}",
==>   "all_typed_relationships" : "http://localhost:7474/db/data/node/0
/relationships/all/{-list|&|types}",
==>   "property" : "http://localhost:7474/db/data/node/0/properties/{key}",
==>   "self" : "http://localhost:7474/db/data/node/0",
==>   "properties" : "http://localhost:7474/db/data/node/0/properties",
==>   "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/0
/relationships/out/{-list|&|types}",
==>   "incoming_relationships" : "http://localhost:7474/db/data/node/0
/relationships/in",
==>   "extensions" : {
==>   },
==>   "create_relationship" : "http://localhost:7474/db/data/node/0
/relationships",
==>   "paged_traverse" : "http://localhost:7474/db/data/node/0/paged
/traverse/{returnType}{?pageSize,leaseTime}",
==>   "all_relationships" : "http://localhost:7474/db/data/node/0
/relationships/all",
==>   "incoming_typed_relationships" : "http://localhost:7474/db/data/node/0
/relationships/in/{-list|&|types}"
==> }
http>
```

24.4. The Server Info tab

The Server Info tab provides raw access to all available management objects (see Chapter 23, *Monitoring* for details).

Figure 24.9. JMX Attributes



Chapter 25. 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!

25.1. 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 25.1.4, "Read-only mode" for local databases and see Section 25.1.1, "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.

25.1.1. Enabling the shell server

Shell is enabled from the configuration of the Neo4j kernel, see Section 17.2, "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 17.2, "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.

25.1.2. 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)$
```

25.1.3. 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
```

25.1.4. 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
```

25.1.5. 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
```

25.2. 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.

25.3. 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`.

25.4. 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.

25.5. 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.*`

25.6. 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:

25.6.1. 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.

25.6.2. 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 25.4, "Filters" and for information about how to specify direction see Section 25.3, "Enum options". Use `man ls` for more info.

25.6.3. 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.

25.6.4. 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]
```

Example of setting a `String` property containing a JSON string:

```
mkrel -c -d i -t DOMAIN_OF --np "{'app':'foobar'}"
```

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

25.6.5. Deleting nodes and relationships

Deletion of nodes and relationships is done with the `rmnode` and `rmrel` commands. `rmnode` can delete nodes, if the node to be deleted still has relationships they can also be deleted by supplying -f option. `rmrel` can delete relationships, it tries to ensure connectedness in the graph, but relationships can be deleted regardless with the -f option. `rmrel` can also delete the node on the other side of the deleted relationship if it's left with no more relationships, see -d option.

25.6.6. 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`

25.6.7. 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.

25.6.8. 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 25.4, "Filters". See Section 25.3, "Enum options" for the traversal order option. Even relationship types/directions are supplied using the same format as filters.

25.6.9. Query with Cypher

You can use Cypher to query the graph. For that, use the `start` command.

- `start n = (0) return n` will give you a listing of the node with ID 0

25.6.10. 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. Key and value is optional.

25.7. 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/1.5-SNAPSHOT/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:

```
import org.neo4j.helpers.Service;
import org.neo4j.shell.kernel.apps.GraphDatabaseApp;

@Service.Implementation( App.class )
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;
    }
}
```

And you could now use it in the shell by typing `lsreltypes` (its name is based on the class name) if `getName` method isn't overridden.

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.

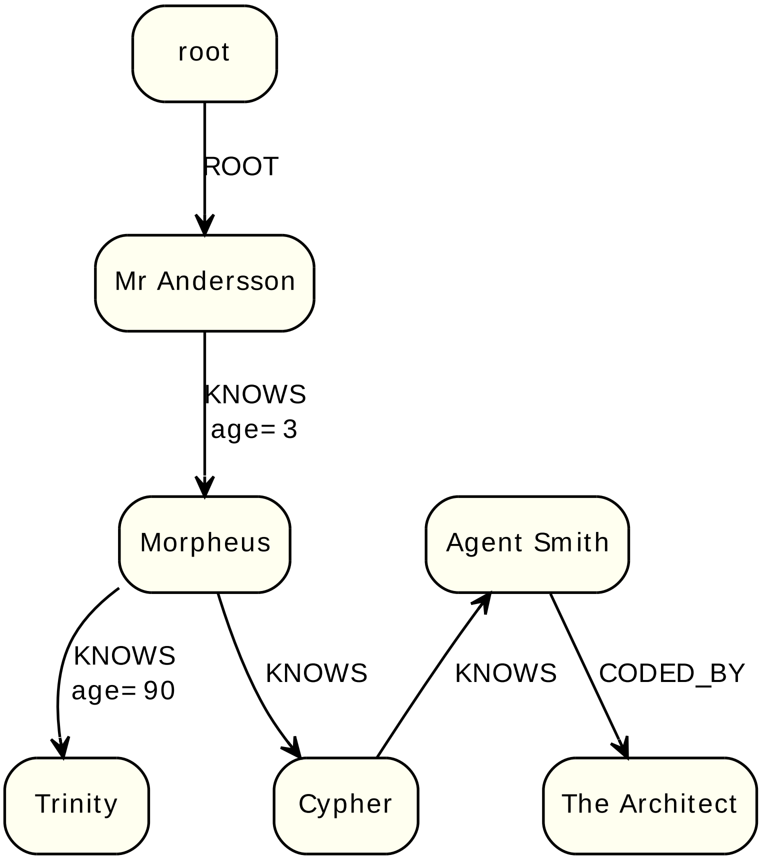
Know that the apps reside server-side so if you have a running server and starts a remote client to it from another JVM you can't add your apps on the client.

25.8. An example shell session

```
# where are we?
neo4j-sh (0)$ pwd
Current is (0)
(0)


# On the current node, set the key "name" to value "Jon"
neo4j-sh (0)$ set name "Jon"

# make an incoming relationship of type LIKES, create the end node with the node properties specified.
neo4j-sh (Jon,0)$ mkrel -c -d i -t LIKES --np "{'app':'foobar'}"

# where are we?
neo4j-sh (Jon,0)$ ls
*name =[Jon]
(me) <-[LIKES]-- (1)


# change to the newly created node
neo4j-sh (Jon,0)$ cd 1

# list relationships, including relationshship id
neo4j-sh (1)$ ls -avr
(me) --[LIKES,0]-> (Jon,0)


# create one more KNOWS relationship and the end node
neo4j-sh (1)$ mkrel -c -d i -t KNOWS --np "{'name':'Bob'}"

# print current history stack
neo4j-sh (1)$ pwd
Current is (1)
(Jon,0)-->(1)


# verbose list relationships
neo4j-sh (1)$ ls -avr
(me) --[LIKES,0]-> (Jon,0)
(me) <-[KNOWS,1]-- (Bob,2)
```

25.9. A Matrix example

This example is creating a graph of the characters in the Matrix via the shell and then executing Cypher queries against it:



Neo4j is configured for autoindexing in this case with

```
node_auto_indexing=true
node_keys_indexable=name,age

relationship_auto_indexing=true
relationship_keys_indexable=ROOT,KNOWS,CODED_BY
```

in the neo4j configuration file.

The following is a sample shell session creating the Matrix graph and querying it.

```
# create the Thomas Andersson node
neo4j-sh (0)$ mkrel -t ROOT -c -v
Node (1) created
Relationship [ROOT,0] created


# go to the new node
neo4j-sh (0)$ cd 1

# set the name property
neo4j-sh (1)$ set name "Thomas Andersson"

# create Thomas direct friends
neo4j-sh (Thomas Andersson,1)$ mkrel -t KNOWS -cv
Node (2) created
Relationship [KNOWS,1] created
```

```
# go to the new node
neo4j-sh (Thomas Andersson,1)$ cd 2

# set the name property
neo4j-sh (2)$ set name "Trinity"

# go back in the history stack
neo4j-sh (Trinity,2)$ cd ..

# create Thomas direct friends
neo4j-sh (Thomas Andersson,1)$ mkrel -t KNOWS -cv
Node (3) created
Relationship [KNOWS,2] created


# go to the new node
neo4j-sh (Thomas Andersson,1)$ cd 3

# set the name property
neo4j-sh (3)$ set name "Morpheus"

# create relationship to Trinity
neo4j-sh (Morpheus,3)$ mkrel -t KNOWS -n 2

# list the relationships of node 3
neo4j-sh (Morpheus,3)$ ls -rv
(me) --[KNOWS,3]-> (Trinity,2)
(me) <-[KNOWS,2]-- (Thomas Andersson,1)


# change the current position to relationship #2
neo4j-sh (Morpheus,3)$ cd -r 2

# set the age property on the relationship
neo4j-sh [KNOWS,2]$ set -t int age 3

# back to Morpheus
neo4j-sh [KNOWS,2]$ cd ..

# next relationsip
neo4j-sh (Morpheus,3)$ cd -r 3

# set the age property on the relationship
neo4j-sh [KNOWS,3]$ set -t int age 90

# position to the start node of the current relationship
neo4j-sh [KNOWS,3]$ cd start

# new node
neo4j-sh (Morpheus,3)$ mkrel -t KNOWS -c

# list relationships on the current node
neo4j-sh (Morpheus,3)$ ls -r
(me) --[KNOWS]-> (Trinity,2)
(me) --[KNOWS]-> (4)
(me) <-[KNOWS]-- (Thomas Andersson,1)


# go to Cypher
neo4j-sh (Morpheus,3)$ cd 4

# set the name
neo4j-sh (4)$ set name Cypher

# create new node from Cypher
```

```
neo4j-sh (Cypher,4)$ mkrel -ct KNOWS

# list relationships
neo4j-sh (Cypher,4)$ ls -r
(me) --[KNOWS]-> (5)
(me) <-[KNOWS]-- (Morpheus,3)


# go to the Agent Smith node
neo4j-sh (Cypher,4)$ cd 5

# set the name
neo4j-sh (5)$ set name "Agent Smith"

# outgoing relationship and new node
neo4j-sh (Agent Smith,5)$ mkrel -cvt CODED_BY
Node (6) created
Relationship [CODED_BY,6] created


# go there
neo4j-sh (Agent Smith,5)$ cd 6

# set the name
neo4j-sh (6)$ set name "The Architect"

# go to the first node in the history stack
neo4j-sh (The Architect,6)$ cd
```

Part VI. Community

The Neo4j project has a strong community around it. Read about how to get help from the community and how to contribute to it.

Chapter 26. Community Support

You can learn a lot about Neo4j on different *events.* To get information on upcoming Neo4j events, have a look here:

- http://events.neo4j.org/
- http://neo4j.meetup.com/

Get help from the Neo4j open source community; here are some starting points.

- Forum / Mailing list (searchable) <http://neo4j.org/forums/>, archive <http://www.mail-archive.com/user@lists.neo4j.org/info.html>, sign up <http://neo4j.org/#SubscribeForm>.
- Twitter: http://twitter.com/neo4j
- IRC channel: irc://irc.freenode.net/neo4j web chat <http://webchat.freenode.net/?randomnick=1&channels=neo4j>.
- Neo4j wiki <http://wiki.neo4j.org/>.

Report a *bug* or add a *feature request*:

- General: https://github.com/neo4j/community/issues
- Monitoring: https://github.com/neo4j/advanced/issues
- Backup and High Availability: https://github.com/neo4j/enterprise/issues

Questions regarding the *documentation:* The Neo4j Manual is published online with a comment function, please use that to post any questions or comments. See http://docs.neo4j.org/.

Chapter 27. Contributing to Neo4j

There's a rather outdated guide on this topic in the wiki: http://wiki.neo4j.org/content/Code_Contributor%27s_Guide This information will be updated and moved to the manual.

One crucial aspect of contributing is the Contributor License Agreement <http://wiki.neo4j.org/content/About_Contributor_License_Agreement>. In short: make sure to sign the CLA, or the Neo4j project won't be able to accept your contribution.

Note that you can contribute to Neo4j also by contributing documentation or giving feedback on the current documentation. Basically, at all the places where you can get help, there's also room for contributions.

27.1. Writing Neo4j Documentation

Note

Other than writing documentation, you can help out by providing comments - head over to the online HTML version <http://docs.neo4j.org/chunked/snapshot/> to do that!

For how to build the manual see: readme <https://github.com/neo4j/manual/blob/master/README.asciidoc>

The documents use the asciidoc format, see:

- Aciidoc Reference <http://www.methods.co.nz/asciidoc/>
- AsciiDoc cheatsheet <http://powerman.name/doc/asciidoc>

The cheatsheet is really useful!

27.1.1. Overall Flow

Each (sub)project has its own documentation, which will produce a *docs.jar* file. By default this file is assembled from the contents in *src/docs/*. Asciidoc documents have the `.txt` file extension.

The documents can use code snippets which will extract code from the project. The corresponding code must be deployed to the *sources.jar* or *test-sources.jar* file.

The above files are all consumed by the build of the manual (by adding them as dependencies). To get content included in the manual, it has to be explicitly included by a document in the manual as well.

27.1.2. File Structure in *docs.jar*

Directory	Contents
dev/	content aimed at developers
dev/images/	images used by the dev docs
ops/	content aimed at operations
ops/images/	images used by the ops docs
man/	manpages

Additional subdirectories are used as needed to structure the documents, like *dev/tutorial/*, *ops/tutorial/* etc.

27.1.3. Headings and document structure

Each document starts over with headings from level zero (the document title). Each document should have an id. In some cases sections in the document need to have id's as well, this depends on where they fit in the overall structure. To be able to link to content, it has to have an id. Missing id's in mandatory places will fail the build.

This is how a document should start:

```
[[unique-id-verbose-is-ok]]
The Document Title
==================
```

To push the headings down to the right level in the output, the `leveloffset` attribute is used when including the document inside of another document.

Subsequent headings in a document should use the following syntax:

```
== Subheading ==

... content here ...

=== Subsubheading ===

content here ...
```

Asciidoc comes with one more syntax for headings, but in this project it's not used.

27.1.4. Writing

Put one sentence on each line. This makes it easy to move content around, and also easy to spot (too) long sentences.

27.1.5. Gotchas

- A chapter can't be empty. (the build will fail on the docbook xml validity check)
- The document title should be "underlined" by the same number of `=` as there are characters in the title.
- Always leave a blank line at the end of documents (or the title of the next document might end up in the last paragraph of the document)
- As `{}` are used for Asciidoc attributes, everything inside will be treated as an attribute. What you have to do is to escape the opening brace: `\{`. If you don't, the braces and the text inside them will be removed without any warning being issued!

27.1.6. Links

To link to other parts of the manual the id of the target is used. This is how such a reference looks:

```
<<community-docs-overall-flow>>
```

Which will render like: Section 27.1.1, "Overall Flow"

Note

Just write "see <<target-id>>" and similar, that should suffice in most cases.

If you need to link to another document with your own link text, this is what to do:

```
<<target-id, link text that fits in the context>>
```

Note

Having lots of linked text may work well in a web context but is a pain in print, and we aim for both!

External links are added like this:

```
http://neo4j.org/[Link text here]
```

Which renders like: Link text here <http://neo4j.org/>

For short links it may be better not to add a link text, just do:

```
http://neo4j.org/
```

Which renders like: http://neo4j.org/

Note
It's ok to have a dot right after the URL, it won't be part of the link.

27.1.7. Text Formatting

- **Bold** - just don't do it, the editor in charge is likely to remove it anyhow!
- _Italics_ is rendered as *Italics*
- +methodName()+ is rendered as `methodName()` and is used for literals as well
- `command` is rendered as `command` (typically used for command-line)
- 'my/path/' is rendered as *my/path/* (used for file names and paths)

27.1.8. Admonitions

These are very useful and should be used where appropriate. Choose from the following (write all caps and no, we can't easily add new ones):

Note
Note.

Tip
Tip.

Important
Important

Caution
Caution

Warning
Warning

Here's how it's done:

```
NOTE: Note.
```

A multiline variation:

```
[TIP]
Tiptext.
Line 2.
```

Which is rendered as:

Tip
Tiptext. Line 2.

27.1.9. Images

Important

All images in the entire manual share the same namespace. You know how to handle that.

Images Files

To include an image file, make sure it resides in the *images/* directory relative to the document you're including it from. Then go:

```
image::neo4j-logo.png[]
```

Which is rendered as:

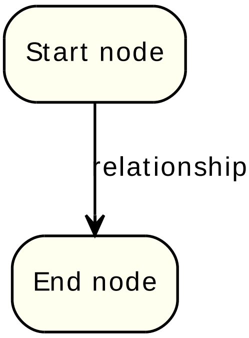
Static Graphviz/DOT

We use the Graphviz/DOT language to describe graphs. For documentation see http://graphviz.org/.

This is how to include a simple example graph:

```
["dot", "community-docs-graphdb-rels.svg"]
----
"Start node" -> "End node" [label="relationship"]
----
```

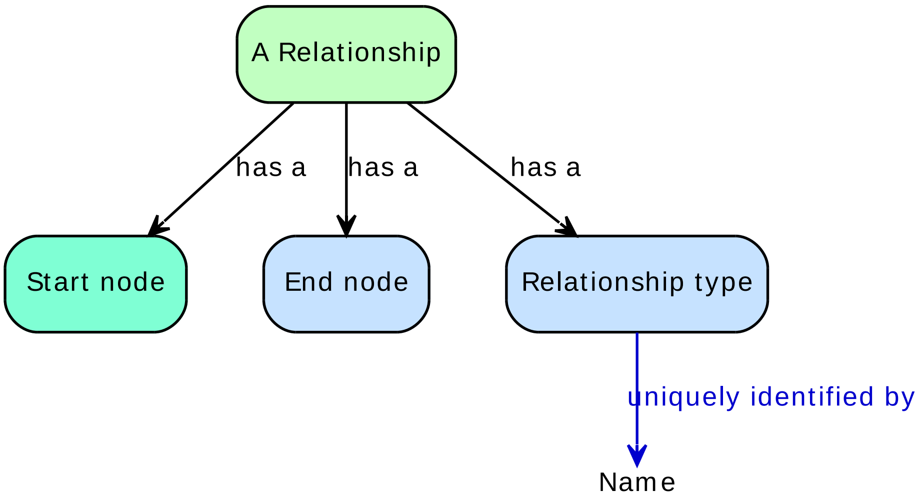
Which is rendered as:



Here's an example using some predefined variables available in the build:

```
["dot", "community-docs-graphdb-rels-overview.svg", "meta"]
----
"A Relationship" [fillcolor="NODEHIGHLIGHT"]
"Start node" [fillcolor="NODE2HIGHLIGHT"]
"A Relationship" -> "Start node" [label="has a"]
"A Relationship" -> "End node" [label="has a"]
"A Relationship" -> "Relationship type" [label="has a"]
Name [TEXTNODE]
"Relationship type" -> Name [label="uniquely identified by" color="EDGEHIGHLIGHT" fontcolor="EDGEHIGHLIGHT"]
----
```

Which is rendered as:



The optional second argument given to the dot filter defines the style to use:

- when not defined: Default styling for nodespace examples.
- `neoviz`: Nodespace view generated by Neoviz.
- `meta`: Graphs that don't resemble db contents, but rather concepts.

Caution
Keywords of the DOT language have to be surrounded by double quotes when used for other purposes. The keywords include *node, edge, graph, digraph, subgraph,* and *strict*.

27.1.10. Attributes

Common attributes you can use in documents:

- {neo4j-version} - rendered as "1.5-SNAPSHOT"
- {neo4j-git-tag} - rendered as "master"

These can substitute part of URLs that point to for example APIdocs or source code. Note that neo4j-git-tag also handles the case of snapshot/master.

27.1.11. Comments

There's a separate build including comments. The comments show up with a yellow background. It's located at http://docs.neo4j.org/annotated/. You can also use this page to search for content, as the full manual is on a single page. CAUTION: This version isn't intended for users in general, as the Disqus feature isn't available there.

Here's how to write a comment:

```
// this is a comment
```

The comments are not visible in the normal build. Comment blocks won't be included in the output of any build at all. Here's a comment block:

```
////
Note that includes in here will still be processed, but not make it into the output.
That is, missing includes here will still break the build!
////
```

27.1.12. Code Snippets

Explicitly defined in the document

Warning
Use this kind of code snippets as little as possible. They are well known to get out of sync with reality after a while.

This is how to do it:

```
[source,cypher]
----
start n=(2, 1) where (n.age < 30 and n.name = "Tobias") or not(n.name = "Tobias")  return n
----
```

Which is rendered as:

```
start n=(2, 1) where (n.age < 30 and n.name = "Tobias") or not(n.name = "Tobias")  return n
```

If there's no suitable syntax highlighter, just omit the language: `[source]`.

Currently the following syntax highlighters are enabled:

- Bash
- Cypher
- Groovy
- Java
- JavaScript
- Python
- XML

For other highlighters we could add see http://alexgorbatchev.com/SyntaxHighlighter/manual/brushes/.

Fetched from source code

Code can be automatically fetched from source files. You need to define:

- component: the `artifactId` of the Maven coordinates,
- source: path to the file inside the jar it's deployed to,
- classifier: `sources` or `test-sources` or any other classifier pointing to the artifact,
- tag: tag name to search the file for,
- the language of the code, if a corresponding syntax highlighter is available.

Note that the artifact has to be included as a Maven dependency of the Manual project so that the files can be found.

Be aware of that the tag "abc" will match "abcd" as well. It's a simple on/off switch, meaning that multiple occurrences will be assembled into a single code snippet in the output. This behavior can be user to hide away assertions from code examples sourced from tests.

This is how to define a code snippet inclusion:

```
[snippet,java]
----
```

```
component=neo4j-examples
source=org/neo4j/examples/HelloWorldTest.java
classifier=test-sources
tag=startDb
----
```

This is how it renders:

```
GraphDatabaseService graphDb = new EmbeddedGraphDatabase( DB_PATH );
registerShutdownHook( graphDb );
```

Query Results

There's a special filter for Cypher query results. This is how to tag a query result:

```
.Result
[queryresult]
----
+----------------------------------+
| friend_of_friend.name | count(*) |
+----------------------------------+
| Ian                   | 2        |
| Derrick               | 1        |
| Jill                  | 1        |
+----------------------------------+
3 rows, 12 ms
----
```

This is how it renders:

Result

friend_of_friend.name	count(*)
Ian	2
Derrick	1
Jill	1
3 rows, 12 ms	

27.1.13. A sample Java based documentation test

For Java, there are a couple of premade utilities that keep code and documentation together in Javadocs and code snippets that generate Asciidoc for the rest of the toolchain.

To illustrate this, look at the following documentation that generates the Asciidoc file `hello-world-title.txt` with a content of:

```
[[examples-hello-world-sample-chapter]]
Hello world Sample Chapter
==========================

This is a sample documentation test, demonstrating different ways of
bringing code and other artifacts into Asciidoc form. The title of the
generated document is determined from the method name, replacing "+_+" with
" ".

Below you see a number of different ways to generate text from source,
inserting it into the JavaDoc documentation (really being Asciidoc markup)
via the +@@+ snippet markers and programmatic adding with runtime data
in the Java code.

- The annotated graph as http://www.graphviz.org/[GraphViz]-generated visualization:
```

```
.Hello World Graph
["dot", "Hello-World-Graph-hello-world-Sample-Chapter.svg", "neoviz"]
----
  N1 [
    label = "{Node\[1\]|'name' = 'you' : String\l}"
  ]
  N2 [
    label = "{Node\[2\]|'name' = 'I' : String\l}"
  ]
  N2 -> N1 [
    label = "know\n"
  ]
----

- A sample Cypher query:

[source,cypher]
----
START n = node(Node[2])
RETURN n
----

- A sample text output snippet:

[source]
----
Hello graphy world!
----

- a generated source link to the original GIThub source for this test:

https://github.com/neo4j/community/blob/{neo4j-git-tag}/embedded-examples/src/test/java/org/neo4j/examples/DocumentationTest.java[Do

- The full source for this example as a source snippet, highlighted as Java code:

[snippet,java]
----
component=neo4j-examples
source=org/neo4j/examples/DocumentationTest.java
classifier=test-sources
tag=sampleDocumentation
----

This is the end of this chapter.
```

this file is included in this documentation via

```
:leveloffset: 3
include::{importdir}/neo4j-examples-docs-jar/dev/examples/hello-world-sample-chapter.txt[]
```

which renders the following chapter:

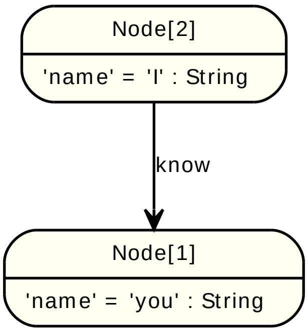
27.1.14. Hello world Sample Chapter

This is a sample documentation test, demonstrating different ways of bringing code and other artifacts into Asciidoc form. The title of the generated document is determined from the method name, replacing "_" with " ".

Below you see a number of different ways to generate text from source, inserting it into the JavaDoc documentation (really being Asciidoc markup) via the `@@` snippet markers and programmatic adding with runtime data in the Java code.

- The annotated graph as GraphViz <http://www.graphviz.org/>-generated visualization:

Figure 27.1. Hello World Graph



- A sample Cypher query:

```
START n = node(Node[2])
RETURN n
```

- A sample text output snippet:

```
Hello graphy world!
```

- a generated source link to the original GIThub source for this test:

DocumentationTest.java <https://github.com/neo4j/community/blob/master/embedded-examples/src/test/java/org/neo4j/examples/DocumentationTest.java>

- The full source for this example as a source snippet, highlighted as Java code:

```
// START SNIPPET: _sampleDocumentation
package org.neo4j.examples;

import static org.neo4j.visualization.asciidoc.AsciidocHelper.createCypherSnippet;
import static org.neo4j.visualization.asciidoc.AsciidocHelper.createGraphViz;
import static org.neo4j.visualization.asciidoc.AsciidocHelper.createOutputSnippet;

import org.junit.Test;
import org.neo4j.kernel.impl.annotations.Documented;
import org.neo4j.test.GraphDescription.Graph;

public class DocumentationTest extends AbstractJavaDocTestbase
{
    /**
     * This is a sample documentation test, demonstrating different ways of
     * bringing code and other artifacts into Asciidoc form. The title of the
     * generated document is determined from the method name, replacing "+_+" with
     * " ".
     *
     * Below you see a number of different ways to generate text from source,
     * inserting it into the JavaDoc documentation (really being Asciidoc markup)
     * via the +@@+ snippet markers and programmatic adding with runtime data
     * in the Java code.
     *
     * - The annotated graph as http://www.graphviz.org/[GraphViz]-generated visualization:
     *
     * @@graph
     *
     * - A sample Cypher query:
     *
     * @@cypher
     *
```

```
     * - A sample text output snippet:
     *
     * @@output
     *
     * - a generated source link to the original GIThub source for this test:
     *
     * @@github
     *
     * - The full source for this example as a source snippet, highlighted as Java code:
     *
     * @@sampleDocumentation
     *
     * This is the end of this chapter.
     */
    @Test
    // signaling this to be a documentation test
    @Documented
    // the graph data setup as simple statements
    @Graph( "I know you" )
    // title is determined from the method name
    public void hello_world_Sample_Chapter()
    {
        // initialize the graph with the annotation data
        data.get();
        gen.get().addSourceSnippets( this.getClass(), "sampleDocumentation" );
        gen.get().addGithubLink( "github", this.getClass(), "neo4j/community",
                "embedded-examples" );

        gen.get().addSnippet( "output",
                createOutputSnippet( "Hello graphy world!" ) );

        gen.get().addSnippet(
                "graph",
                createGraphViz( "Hello World Graph", graphdb(),
                        gen.get().getTitle() ) );
        // A cypher snippet referring to the generated graph in the start clause
        gen.get().addSnippet(
                "cypher",
                createCypherSnippet( "start n = node(" + data.get().get( "I" )
                                     + ") return n" ) );
    }
}
// END SNIPPET: _sampleDocumentation
```

This is the end of this chapter.

27.1.15. Toolchain

Useful links when configuring the docbook toolchain:

- http://www.docbook.org/tdg/en/html/docbook.html
- http://www.sagehill.net/docbookxsl/index.html
- http://docbook.sourceforge.net/release/xsl/1.76.1/doc/html/index.html
- http://docbook.sourceforge.net/release/xsl/1.76.1/doc/fo/index.html

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 the server.

status
: Current running state of the server.

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

remove
: Uninstalls the system service.

info
: Displays configuration information, such as the current NEO4J_HOME and CLASSPATH.

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.

Neo4j.bat install/remove

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

- Neo4j.bat install - install as a Windows service
  - will install the service
- Neo4j.bat remove - remove the Neo4j service
  - will stop and remove the Neo4j service
- Neo4j.bat start - will start the Neo4j service
  - will start the Neo4j service if installed or a console
  - session otherwise.
- Neo4j.bat stop - stop the Neo4j service if running

- Neo4j.bat restart - restart the Neo4j service if installed
- Neo4j.bat status - report on the status of the Neo4j service
  - returns RUNNING, STOPPED or NOT INSTALLED

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.

status
: Current running state of the server

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

remove
: Uninstalls the system service

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.

Appendix B. Questions & Answers

Q: What is the maximum number of nodes supported? What is the maximum number of edges supported per node?

A: At the moment it is 34.4 billion nodes, 34.4 billion relationships, and 68.7 billion properties, in total.

Q: What is the largest complete connected graph supported (i.e. every node is connecting to all other nodes)?

A: Theoretical limits can be derived from numbers above: It basically comes out to a full graph of 262144 nodes and 34359607296 relationships. We have never seen this use case though.

Q: Are read/write depending on the number of nodes/edges in the DB?

A: This question can mean a couple of different things. The performance of a single read/write operation does not depend on the size of the DB. Whether the graph has 10 nodes or 10 million nodes does not matter. — There is however another facet here, which is that if your graph is big on disk, you may not be able to fit it all into the cache in RAM. Therefore, you may end up hitting disk more often. Most customers don't have graphs of this size, but some do. If you happen to reach these sizes, we have approaches for scaling out on multiple machines to mitigate the performance impact by increasing the cache "surface area" across machines.

Q: How many concurrent read/write requests supported?

A: There is no limit on the number of concurrent requests. The amount of requests we can serve per second depends very much on the operation performed (heavy write operation, simple read, complex traversal, etc.), and the hardware used. A rough estimate is 1,000 hops per millisecond while traversing the graph in the simplest way possible. After a discussion about the specific use case, we would be able to give a better idea of the performance one can expect.

Q: How is data consistency maintained in cluster environment?

A: Master-slave replication. Slaves pull changes from the master. The pull interval can be configured per slave, from subsecond to minutes, as necessary. HA can also write through slaves. When that happens, the slave that is being written through catches up with the master, and then the write is made durable on the slave and the master. The other slaves then catch up as normal.

Q: How is the latency in updating all the servers when there is an update on the DB from one of them?

A: Pull interval can be configured per slave, from subsecond to minutes, as necessary. When writing through a slave, the slave is immediately synchronized with the master before the write is committed on the slave and the master. In general, read/write load does not affect slaves syncing up. A heavy write load will however put pressure on the filesystem of the master, which is also required for reading changes for the slaves. In practice, we have however not seen this become a notable issue.

Q: Will the latency increase proportional to the number of servers in the cluster?

A: When scaling beyond 10s of slaves in a cluster, we anticipate that the number of pull requests coming from slaves will reduce the performance of the master. Only write performance on the cluster would be affected. Read performance would continue to scale linearly.

Q: Is online expansion supported? In other words, do we need to bring down all the servers and the DB if we want to add new servers to the cluster?

A: New slaves can be added to an existing cluster without having to stop and start the whole cluster. Our HA protocol will bring a newly added slave up-to-date. Slaves can also be removed simply by shutting them down.

Q: How long will it take for the newly joined servers to sync up?

A: We recommend providing a new slave with a recent snapshot of the database before bringing it online. This is typically done from a backup. The slave will then only need to synchronize the most recent updates, which will typically be a matter of seconds.

Q: How long does it take to reboot?

A: If by reboot, you mean take the cluster down and take it up again, it's pretty much dependent on how fast you can type. So it could be <10s. The Neo4j caches will however not auto-warm up, but the OS filesystem cache will retain its data.

Q: Are there any backup and restore/recovery mechanisms?

A: Neo4j Enterprise Edition provides an online backup feature for full and incremental backups during operation.

Q: Is cross-continental clustering supported? Say, can servers in the cluster be located in different continents provided that the chance for inter-continental communication is much lower than the intra one?

A: We have customers who have tested multi-region deployments in AWS. Cross-continental latencies will have an impact, however on the efficiency of the cluster management and synchronization protocols; large latencies in the cluster management can trigger frequent master re-elections, which will slow down the cluster. Feature support in this area will be improving over time.

Q: Is there any special handling/policy for this kind of setup?

A: We'd have to have a more in-depth discussion about the requirements pertaining to this specific deployment.

Q: Is writing to the DB thread-safe? Or is it the application logic to protect writing to the same nodes/edges?

A: Whether in single instance or HA mode, the database provides thread safety by way of locking on nodes and relationships upon modification.

Q: What is the best strategy for reading back your writes on HA?

A:

1. Sticky sessions.
2. Send back data in response, removing the need to read back in a separate request.
3. Force a pull of updates from the master when required by the operation.

Q: What is the best strategy for get-or-create semantics?

A:

1. Single thread.
2. If not exists, pessimistically lock on a common node (or set of common nodes).
3. If not exists, optimistically create, and then double check afterwards. (explanation will be extended)

How does locking work?

Pessimistic locking. Locks are never required for reading. Writers will not block readers. It's impossible to make a read operation block without using explicit locking facilities. Read locks prevent writes. Acquiring a read lock means consistent view for all holders while held. Grabbing write locks is done automatically when a node/rel is modified/created, or through explicit locking facilities. It can be used to provide read committed semantics and data consistency when necessary.

What about on-size storage?

Neo4j is currently not suitable for storing BLOBs/CLOBs. Nodes, relationships, and properties are not co-located on disk. This might be introduced in the future.

What about indexing?
: Neo4j supports composite property indices. Promote index providers over in-graph indices. Lucene engine manages index paging separately and requires some heap for itself Neo4j currently supports one auto indexer and many individual indexes (search done via API)

How do I query the database?
: Core API, Traversal API, REST API, Cypher, Gremlin

Does Neo4j use journaling?
: Based on write change delta between master and slaves in HA cluster.

How do I tune Neo4j for performance?
: Uses memory-mapped store files Neo4j caching strategies need to be explained:

- Soft-ref cache: Soft references are cleaned when the GC thinks it's needed. Use if app load isn't very high & needs memory-sensitive cache
- Weak-ref cache: GC cleans weak references whenever it finds it. Use if app is under heavy load with lots of reads and traversals
- Strong-ref cache: all nodes & edges are fully cached in memory JVM needs pausing under heavy load, e.g., 1/2 minutes pause interval. Larger heap sizes good, however 12G and beyond is impractical with GC. 100x performance improvement with memory mapped file cache and 1000 improvement with Java heap comparing to fetching from disk I/O

ACID transactions between master & slaves
: Synchronous between slave-initiated transaction to master, eventual from master to slaves. Concurrent multi slave-initiated transaction support with deadlock detection. It's fully consistent from a data integrity point of view, but eventually consistent from sync point of view.

What about the standalone server?
: The REST API is completely stateless, but it can do batches for larger transaction scopes. Thread pooling & thread per socket: For standalone server & HA nodes, Neo4j uses Jetty for connection pooling (e.g., 25/node in HA cluster)

How is a load balancer used with HA?
: Typically a small server extension can be written to return 200 or 404 depending on whether the machine is master or slave. This extension can then be polled by the load balancer to determine the master and slave machine sets. Writing only to slaves ensures that committed transactions exist in at least two places.

What kind of monitoring support does Neo4j provide?
: Neo4j does not currently have built-in tracing or explain plans. JMX is the primary interface for statistics and monitoring. Thread dumps can be used to debug a malfunctioning system.

How do I import my data into Neo4j?
: The Neo4j batch inserter can be used to fill an initial database with data. After batch insertion, the store can be used in an embedded or HA environment. Future data load/refresh should go directly to Production server SQL Importer (built on top of Batch Inserter) is not officially supported