The Neo4j Manual

neotechnology
value in relationships

The Neo4j Manual v2.2.1

The Neo4j Team ***neo4j.com***[1]

[1] http://neo4j.com/

The Neo4j Manual v2.2.1

by The Neo4j Team *neo4j.com*[1]

Publication date 2015-04-13 20:26:27

Starting points

[1] http://neo4j.com/
[2] http://creativecommons.org/licenses/by-sa/3.0/

Preface

This is the reference manual for Neo4j version 2.2.1, authored by the Neo4j Team.

The main parts of the manual are:

- Part I, "Introduction" [1] — introducing graph database concepts and Neo4j.
- Part II, "Tutorials" [14] — learn how to use Neo4j.
- Part III, "Cypher Query Language" [101] — details on the Cypher query language.
- Part IV, "Reference" [264] — detailed information on Neo4j.
- Part V, "Operations" [422] — how to install and maintain Neo4j.
- Part VI, "Tools" [513] — guides on tools.
- Part VII, "Community" [545] — getting help from, contributing to.
- Part VIII, "Advanced Usage" [574] — using Neo4j in more advanced ways.
- Terminology [654] — terminology about graph databases.
- Appendix A, *Resources* [658] — find additional documentation resources.
- Appendix B, *Manpages* [659] — command line documentation.

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 how Neo4j actually works.

There's other documentation resources besides the manual as well, see Appendix A, *Resources* [658].

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 also outlines some specifics of Neo4j.

Chapter 1. Neo4j Highlights

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

It features:

- true ACID transactions,
- high availability,
- scales to billions of nodes and relationships,
- high speed querying through traversals,
- declarative graph query language.

Proper ACID behavior is the foundation of data reliability. Neo4j enforces that all operations that modify data 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 18, *Transaction Management* [272].

Reliable graph storage can easily be added to any application. A 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 25, *High Availability* [468] 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

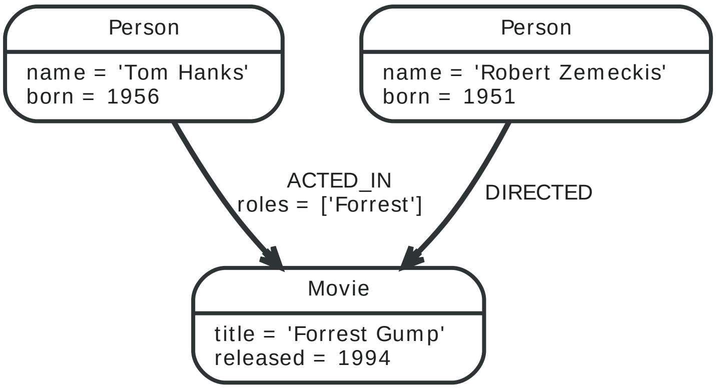
This chapter contains an introduction to the graph data model and also compares it to other data models used when persisting data.

2.1. The Neo4j 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.

For terminology around graph databases, see Terminology [654].

Here's an example graph which we will approach step by step in the following sections:



Nodes

A graph records data in nodes and relationships. Both can have properties. This is sometimes referred to as the *Property Graph Model*.

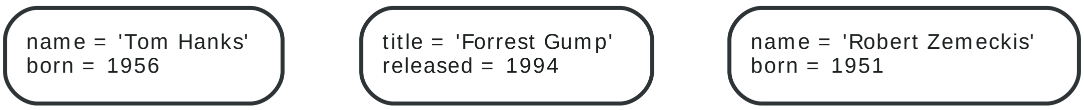
The fundamental units that form a graph are nodes and relationships. In Neo4j, both nodes and relationships can contain properties.

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

Apart from properties and relationships, nodes can also be labeled with zero or more labels.

The simplest possible graph is a single Node. A Node can have zero or more named values referred to as *properties.* Let's start out with one node that has a single property named `title`:

The next step is to have multiple nodes. Let's add two more nodes and one more property on the node in the previous example:



Relationships

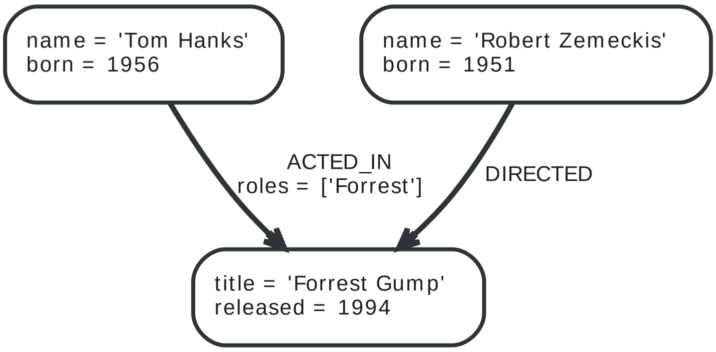
Relationships organize the nodes by connecting them. A relationship connects two nodes — a start node and an end node. Just like nodes, relationships can have properties.

Relationships between nodes are a key part of a graph database. They allow for finding related data. Just like nodes, relationships can have properties.

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

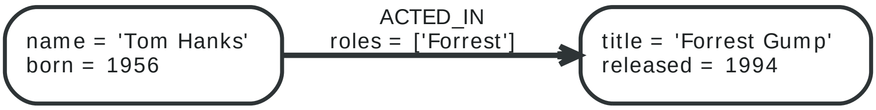
Relationships organize nodes into arbitrary structures, allowing a graph to resemble a list, a tree, a map, or a compound entity — any of which can be combined into yet more complex, richly inter-connected structures.

Our example graph will make a lot more sense once we add relationships to it:



Our example uses `ACTED_IN` and `DIRECTED` as relationship types. The `roles` property on the `ACTED_IN` relationship has an array value with a single item in it.

Below is an `ACTED_IN` relationship, with the `Tom Hanks` node as *start node* and `Forrest Gump` as *end node*.



You could also say that the `Tom Hanks` node has an *outgoing* relationship, while the `Forrest Gump` node has an *incoming* relationship.

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).

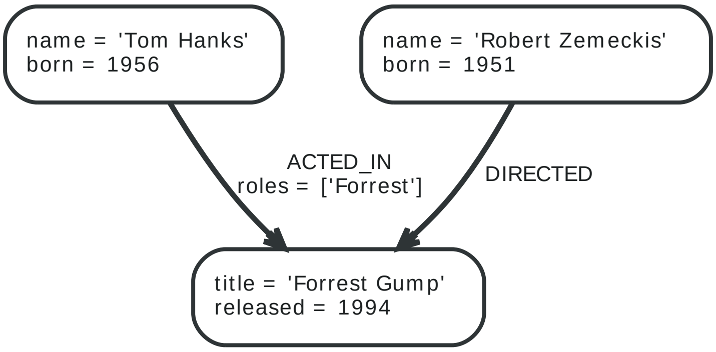
While relationships always have a direction, you can ignore the direction where it is not useful in your application.

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

The example above would mean that `Tom Hanks` `KNOWS` himself.

To further enhance graph traversal all relationships have a relationship type.

Let's have a look at what can be found by simply following the relationships of a node in our example graph:



Using relationship direction and type

What we want to know	Start from	Relationship type	Direction
get actors in movie	movie node	`ACTED_IN`	incoming
get movies with actor	person node	`ACTED_IN`	outgoing
get directors of movie	movie node	`DIRECTED`	incoming
get movies directed by	person node	`DIRECTED`	outgoing

Properties

Both nodes and relationships can have properties.

Properties are named values where the name is a string. The supported property values are:

- Numeric values,
- String values,
- Boolean values,
- Collections of any other type of value.

`NULL` **is not a valid property value.**
`NULL`s can instead be modeled by the absence of a key.

For further details on supported property values, see Section 35.3, "Property values" [597].

Labels

Labels assign roles or types to nodes.

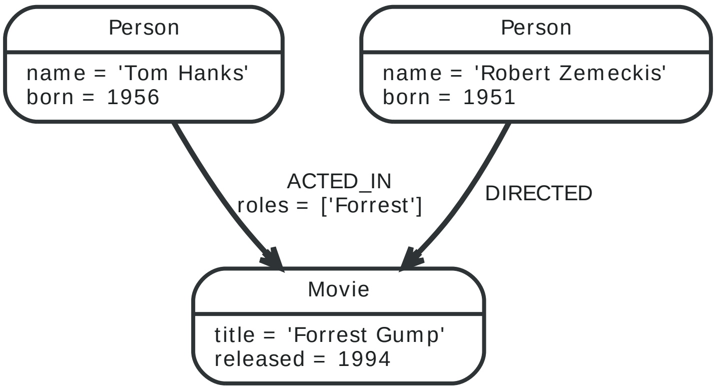
A label is a named graph construct that is used to group nodes into sets; all nodes labeled with the same label belongs to the same set. Many database queries can work with these sets instead of the whole graph, making queries easier to write and more efficient to execute. A node may be labeled with any number of labels, including none, making labels an optional addition to the graph.

Labels are used when defining constraints and adding indexes for properties (see the section called "Schema" [9]).

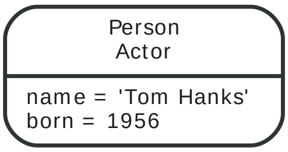
An example would be a label named `User` that you label all your nodes representing users with. With that in place, you can ask Neo4j to perform operations only on your user nodes, such as finding all users with a given name.

However, you can use labels for much more. For instance, since labels can be added and removed during runtime, they can be used to mark temporary states for your nodes. You might create an `Offline` label for phones that are offline, a `Happy` label for happy pets, and so on.

In our example, we'll add `Person` and `Movie` labels to our graph:



A node can have multiple labels, let's add an `Actor` label to the `Tom Hanks` node.



Label names

Any non-empty Unicode string can be used as a label name. In Cypher, you may need to use the backtick (`` ` ``) syntax to avoid clashes with Cypher identifier rules or to allow non-alphanumeric characters in a label. By convention, labels are written with CamelCase notation, with the first letter in upper case. For instance, `User` or `CarOwner`.

Labels have an id space of an int, meaning the maximum number of labels the database can contain is roughly 2 billion.

Traversal

A traversal navigates through a graph to find paths.

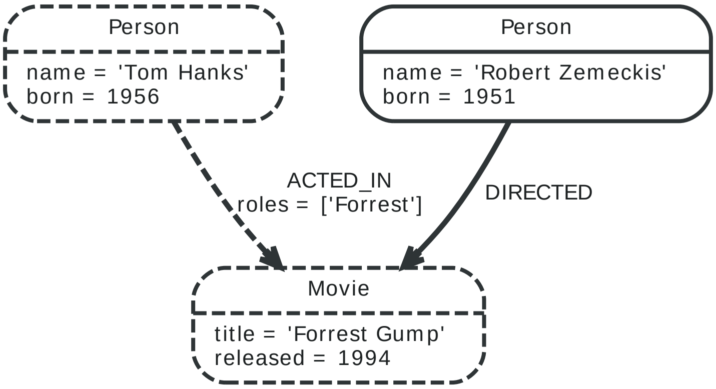
A traversal is how you query a graph, navigating from starting nodes to related nodes, finding answers to questions like "what music do my friends like that I don't yet own," or "if this power supply goes down, what web services are affected?"

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.

Cypher provides a declarative way to query the graph powered by traversals and other techniques. See Part III, "Cypher Query Language" [101] for more information.

When writing server plugins or using Neo4j embedded, Neo4j provides 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.

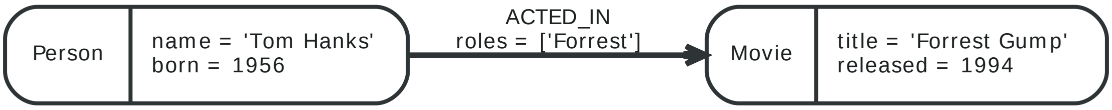
If we want to find out which movies Tom Hanks acted in according to our tiny example database the traversal would start from the `Tom Hanks` node, follow any `ACTED_IN` relationships connected to the node, and end up with `Forrest Gump` as the result (see the dashed lines):



Paths

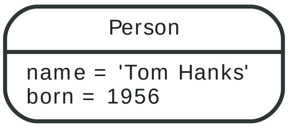
A path is one or more nodes with connecting relationships, typically retrieved as a query or traversal result.

In the previous example, the traversal result could be returned as a path:

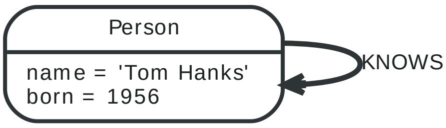


The path above has length one.

The shortest possible path has length zero — that is it contains only a single node and no relationships — and can look like this:



This path has length one:



Schema

Neo4j is a schema-optional graph database.

You can use Neo4j without any schema. Optionally you can introduce it in order to gain performance or modeling benefits. This allows a way of working where the schema does not get in your way until you are at a stage where you want to reap the benefits of having one.

Note
Schema commands can only be applied on the master machine in a Neo4j cluster (see Chapter 25, *High Availability* [468]). If you apply them on a slave you will receive a `Neo.ClientError.Transaction.InvalidType` error code (see Section 21.2, “Neo4j Status Codes” [293]).

Indexes

Performance is gained by creating indexes, which improve the speed of looking up nodes in the database.

Note
This feature was introduced in Neo4j 2.0, and is not the same as the legacy indexes (see Chapter 37, *Legacy Indexing* [632]).

Once you've specified which properties to index, Neo4j will make sure your indexes are kept up to date as your graph evolves. Any operation that looks up nodes by the newly indexed properties will see a significant performance boost.

Indexes in Neo4j are *eventually available*. That means that when you first create an index the operation returns immediately. The index is *populating* in the background and so is not immediately available for querying. When the index has been fully populated it will eventually come *online*. That means that it is now ready to be used in queries.

If something should go wrong with the index, it can end up in a **failed** state. When it is failed, it will not be used to speed up queries. To rebuild it, you can drop and recreate the index. Look at logs for clues about the failure.

You can track the status of your index by asking for the index state through the API you are using. Note, however, that this is not yet possible through Cypher.

How to use indexes through the different APIs:

- Cypher: Section 14.1, "Indexes" [236]
- REST API: Section 21.15, "Indexing" [351]
- Listing Indexes via Shell: the section called "Listing Indexes and Constraints" [537]
- Java Core API: Section 35.4, "User database with indexes" [598]

Constraints

Note
This feature was introduced in Neo4j 2.0.

Neo4j can help you keep your data clean. It does so using constraints, that allow you to specify the rules for what your data should look like. Any changes that break these rules will be denied.

In this version, unique constraints is the only available constraint type.

How to use constraints through the different APIs:

- Cypher: Section 14.2, "Constraints" [238]
- REST API: Section 21.16, "Constraints" [353]
- Listing Constraints via Shell: the section called "Listing Indexes and Constraints" [537]

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.

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

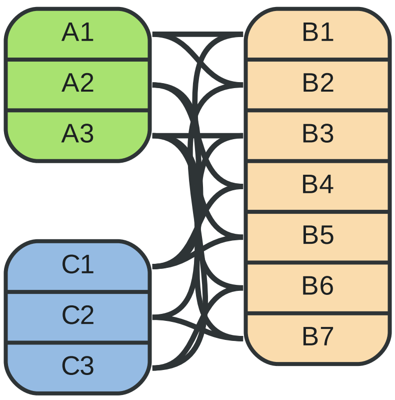
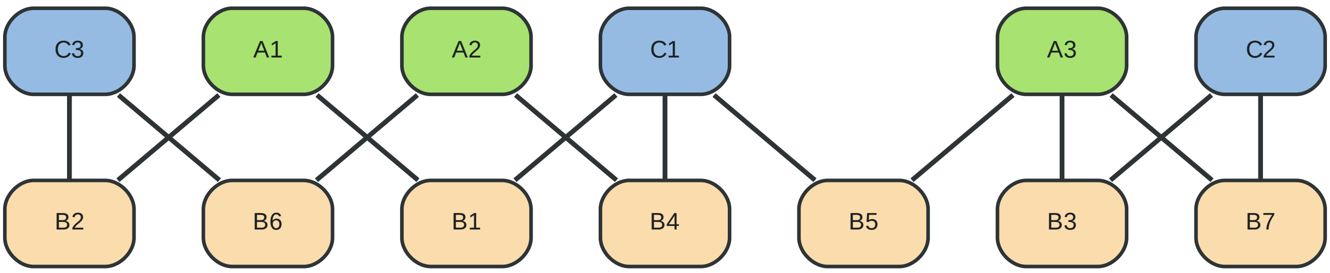


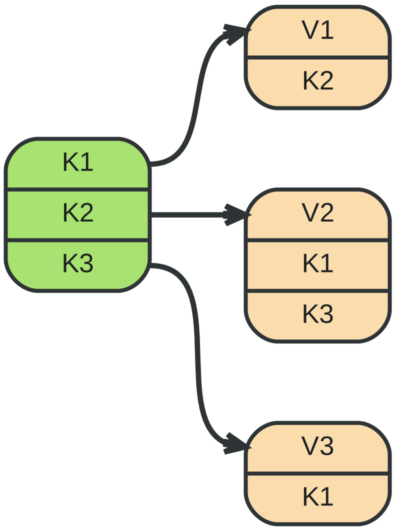
Figure 2.2. Graph Database as RDBMS



A Graph Database elaborates a Key-Value Store

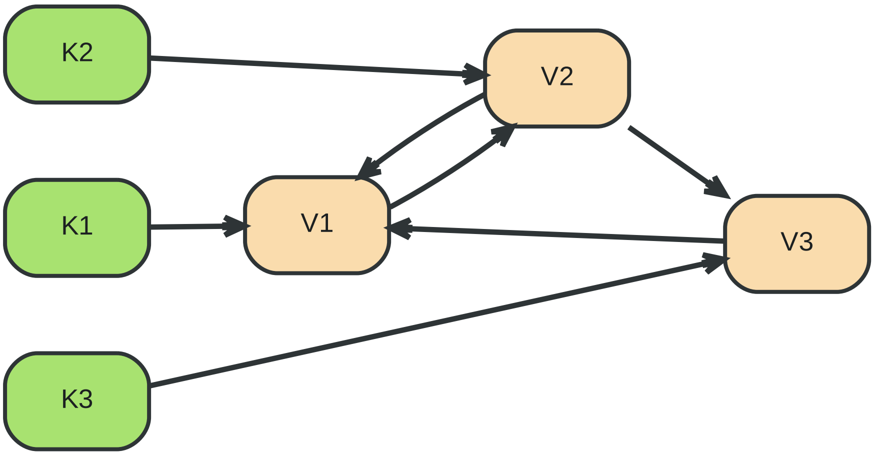
A Key-Value model is great for lookups of simple values or lists. When the values are themselves interconnected, you've got a graph. Neo4j lets you elaborate the simple data structures into more complex, interconnected data.

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



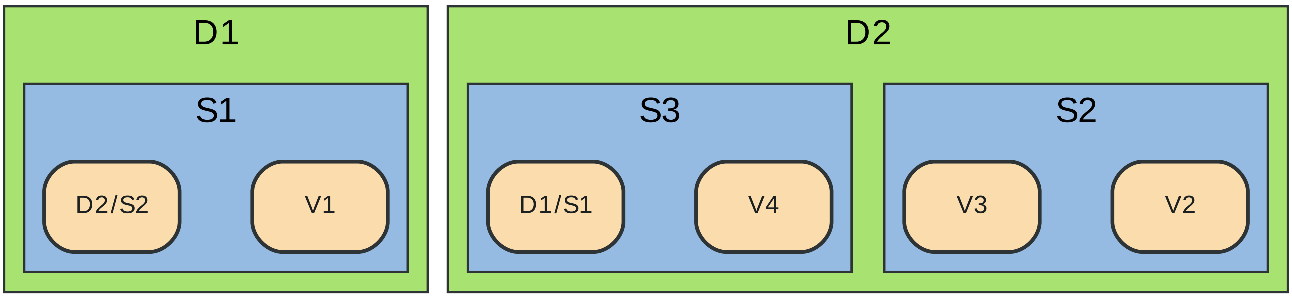
A Graph Database relates Column-Family

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.

A Graph Database navigates a Document Store

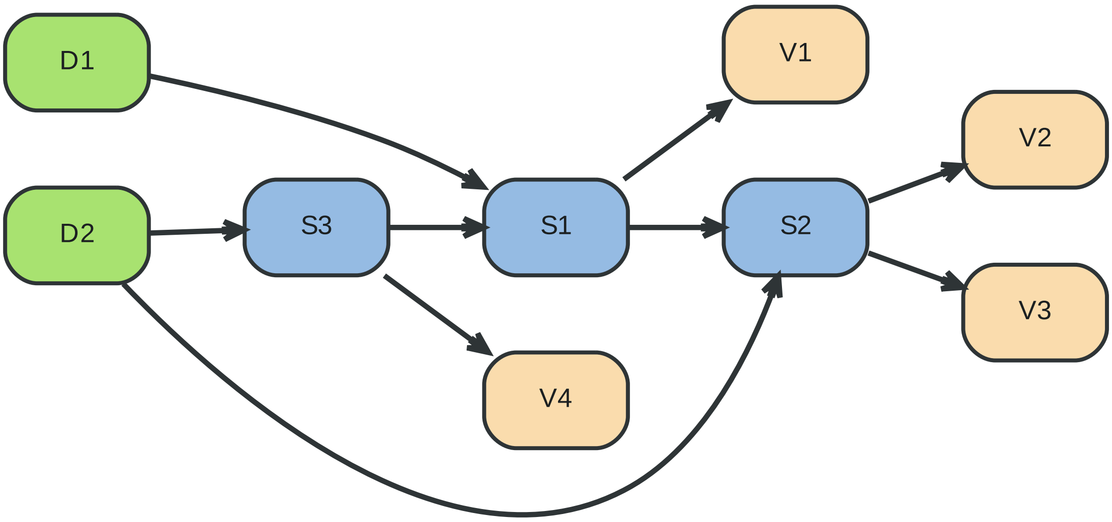
The container hierarchy of a document database accommodates nice, schema-free data that can easily be represented as a tree. Which is of course a graph. Refer to other documents (or document elements) within that tree and you have a more expressive representation of the same data. When in Neo4j, those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



Part II. Tutorials

The tutorial part describes how use Neo4j. It takes you from Hello World to advanced usage of graphs.

Chapter 3. Introduction to Cypher

This friendly guide will introduce you to Cypher, Neo4j's query language.

The guide will help you:

- start thinking about graphs and patterns,
- apply this knowledge to simple problems,
- learn how to write Cypher statements,
- use Cypher for loading data,
- transition from SQL to Cypher.

If you want to keep a reference at your side while reading, please see the Cypher Refcard[1].

Work in Progress
There may still be unfinished parts in this chapter. Please comment on it so we can make it suit our readers better!

[1] http://neo4j.com/docs/2.2.1/cypher-refcard/

3.1. Background and Motivation

Cypher provides a convenient way to express queries and other Neo4j actions. Although Cypher is particularly useful for exploratory work, it is fast enough to be used in production. Java-based approaches (eg, unmanaged extensions) can also be used to handle particularly demanding use cases.

Query processing

To use Cypher effectively, it's useful to have an idea of how it works. So, let's take a high-level look at the way Cypher processes queries.

- Parse and validate the query.
- Generate the execution plan.
- Locate the initial node(s).
- Select and traverse relationships.
- Change and/or return values.

Preparation

Parsing and validating the Cypher statement(s) is important, but mundane. However, generating an optimal search strategy can be far more challenging.

The execution plan must tell the database how to locate initial node(s), select relationships for traversal, etc. This involves tricky optimization problems (eg, which actions should happen first), but we can safely leave the details to the Neo4j engineers. So, let's move on to locating the initial node(s).

Locate the initial node(s)

Neo4j is highly optimized for traversing property graphs. Under ideal circumstances, it can traverse millions of nodes and relationships per second, following chains of pointers in the computer's memory.

However, before traversal can begin, Neo4j must know one or more starting nodes. Unless the user (or, more likely, a client program) can provide this information, Neo4j will have to search for these nodes.

A "brute force" search of the database (eg, for a specified property value) can be *very* time consuming. Every node must be examined, first to see if it has the property, then to see if the value meets the desired criteria. To avoid this effort, Neo4j creates and uses indexes. So, Neo4j uses a separate index for each label/property combination.

Traversal and actions

Once the initial nodes are determined, Neo4j can traverse portions of the graph and perform any requested actions. The execution plan helps Neo4j to determine which nodes are relevant, which relationships to traverse, etc.

3.2. Graphs, Patterns, and Cypher

Nodes, Relationships, and Patterns

Neo4j's Property Graphs are composed of nodes and relationships, either of which may have properties (ie, attributes). Nodes represent entities (eg, concepts, events, places, things); relationships (which may be directed) connect pairs of nodes.

However, nodes and relationships are simply low-level building blocks. The real strength of the Property Graph lies in its ability to encode *patterns* of connected nodes and relationships. A single node or relationship typically encodes very little information, but a pattern of nodes and relationships can encode arbitrarily complex ideas.

Cypher, Neo4j's query language, is strongly based on patterns. Specifically, patterns are used to match desired graph structures. Once a matching structure has been found (or created), Neo4j can use it for further processing.

Simple and Complex Patterns

A simple pattern, which has only a single relationship, connects a pair of nodes (or, occasionally, a node to itself). For example, *a Person* `LIVES_IN` *a City* or *a City is* `PART_OF` *a Country*.

Complex patterns, using multiple relationships, can express arbitrarily complex concepts and support a variety of interesting use cases. For example, we might want to match instances where *a Person* `LIVES_IN` *a Country*. The following Cypher code combines two simple patterns into a (mildly) complex pattern which performs this match:

```
(:Person) -[:LIVES_IN]-> (:City) -[:PART_OF]-> (:Country)
```

Pattern recognition is fundamental to the way that the brain works. Consequently, humans are very good at working with patterns. When patterns are presented visually (eg, in a diagram or map), humans can use them to recognize, specify, and understand concepts. As a pattern-based language, Cypher takes advantage of this capability.

Cypher Concepts

Like SQL[2] (used in relational databases[3]), Cypher is a textual, declarative query language. It uses a form of ASCII art[4] to represent graph-related patterns. SQL-like clauses and keywords (eg, `MATCH`, `WHERE`, `DELETE`) are used to combine these patterns and specify desired actions.

This combination tells Neo4j which patterns to match and what to do with the matching items (eg, nodes, relationships, paths, collections). However, as a declarative[5] language, Cypher does *not* tell Neo4j how to find nodes, traverse relationships, etc. (This level of control is available from Neo4j's Java[6] APIs[7], see Section 34.2, "Unmanaged Extensions" [581])

Diagrams made up of icons and arrows are commonly used to visualize graphs; textual annotations provide labels, define properties, etc. Cypher's ASCII-art syntax formalizes this approach, while adapting it to the limitations of text.

Node Syntax

Cypher uses a pair of parentheses (usually containing a text string) to represent a node, eg: `()`, `(foo)`. This is reminiscent of a circle or a rectangle with rounded end caps. Here are some ASCII-art encodings for example Neo4j nodes, providing varying types and amounts of detail:

```
()
```

[2] https://en.wikipedia.org/wiki/SQL
[3] https://en.wikipedia.org/wiki/Relational_database_management_system
[4] https://en.wikipedia.org/wiki/ASCII_art
[5] https://en.wikipedia.org/wiki/Declarative_programming
[6] https://en.wikipedia.org/wiki/Java_(programming_language)
[7] https://en.wikipedia.org/wiki/Application_programming_interface

```
(matrix)
(:Movie)
(matrix:Movie)
(matrix:Movie {title: "The Matrix"})
(matrix:Movie {title: "The Matrix", released: 1997})
```

The simplest form, `()`, represents an anonymous, uncharacterized node. If we want to refer to the node elsewhere, we can add an identifier, eg: `(matrix)`. Identifiers are restricted (ie, scoped) to a single statement: an identifier may have different (or no) meaning in another statement.

The `Movie` label (prefixed in use with a colon) declares the node's type. This restricts the pattern, keeping it from matching (say) a structure with an `Actor` node in this position. Neo4j's node indexes also use labels: each index is specific to the combination of a label and a property.

The node's properties (eg, `title`) are represented as a list of key/value pairs, enclosed within a pair of braces, eg: `{...}`. Properties can be used to store information and/or restrict patterns. For example, we could match nodes whose `title` is "The Matrix".

Relationship Syntax

Cypher uses a pair of dashes (--) to represent an undirected relationship. Directed relationships have an arrowhead at one end (eg, <--, -->). Bracketed expressions (eg: `[...]`) can be used to add details. This may include identifiers, properties, and/or type information, eg:

```
-->
-[role]->
-[:ACTED_IN]->
-[role:ACTED_IN]->
-[role:ACTED_IN {roles: ["Neo"]}]->
```

The syntax and semantics found within a relationship's bracket pair are very similar to those used between a node's parentheses. An identifier (eg, `role`) can be defined, to be used elsewhere in the statement. The relationship's type (eg, `ACTED_IN`) is analogous to the node's label. The properties (eg, `roles`) are entirely equivalent to node properties. (Note that the value of a property may be an array.)

Pattern Syntax

Combining the syntax for nodes and relationships, we can express patterns. The following could be a simple pattern (or fact) in this domain:

```
(keanu:Person:Actor {name:  "Keanu Reeves"} )
-[role:ACTED_IN     {roles: ["Neo"] } ]->
(matrix:Movie       {title: "The Matrix"} )
```

Like with node labels, the relationship type `ACTED_IN` is added as a symbol, prefixed with a colon: `:ACTED_IN`. Identifiers (eg, `role`) can be used elsewhere in the statement to refer to the relationship. Node and relationship properties use the same notation. In this case, we used an array property for the `roles`, allowing multiple roles to be specified.

Pattern Nodes vs. Database Nodes
When a node is used in a pattern, it *describes* zero or more nodes in the database. Similarly, each pattern describes zero or more paths of nodes and relationships.

Pattern Identifiers

To increase modularity and reduce repetition, Cypher allows patterns to be assigned to identifiers. This allow the matching paths to be inspected, used in other expressions, etc.

```
acted_in = (:Person)-[:ACTED_IN]->(:Movie)
```

The `acted_in` variable would contain two nodes and the connecting relationship for each path that was found or created. There are a number of functions to access details of a path, including `nodes(path)`, `rels(path)` (same as `relationships(path)`), and `length(path)`.

Clauses

Cypher statements typically have multiple *clauses*, each of which performs a specific task, eg:

- create and match patterns in the graph
- filter, project, sort, or paginate results
- connect/compose partial statements

By combining Cypher clauses, we can compose more complex statements that express what we want to know or create. Neo4j then figures out how to achieve the desired goal in an efficient manner.

3.3. Patterns in Practice

Creating Data

We'll start by looking into the clauses that allow us to create data.

To add data, we just use the patterns we already know. By providing patterns we can specify what graph structures, labels and properties we would like to make part of our graph.

Obviously the simplest clause is called `CREATE`. It will just go ahead and directly create the patterns that you specify.

For the patterns we've looked at so far this could look like the following:

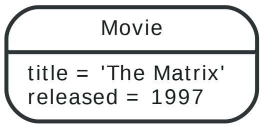
```
CREATE (:Movie { title:"The Matrix",released:1997 })
```

If we execute this statement, Cypher returns the number of changes, in this case adding 1 node, 1 label and 2 properties.

(empty result)

Nodes created: 1
Properties set: 2
Labels added: 1

As we started out with an empty database, we now have a database with a single node in it:



If case we also want to return the created data we can add a `RETURN` clause, which refers to the identifier we've assigned to our pattern elements.

```
CREATE (p:Person { name:"Keanu Reeves", born:1964 })
RETURN p
```

This is what gets returned:

p
`Node[1]{name:"Keanu Reeves",born:1964}`

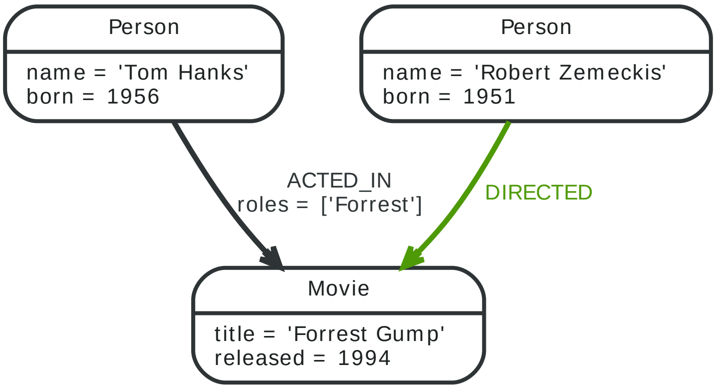
1 row
Nodes created: 1
Properties set: 2
Labels added: 1

If we want to create more than one element, we can separate the elements with commas or use multiple `CREATE` statements.

We can of course also create more complex structures, like an `ACTED_IN` relationship with information about the character, or `DIRECTED` ones for the director.

```
CREATE (a:Person { name:"Tom Hanks",
  born:1956 })-[r:ACTED_IN { roles: ["Forrest"]}]->(m:Movie { title:"Forrest Gump",released:1994 })
CREATE (d:Person { name:"Robert Zemeckis", born:1951 })-[:DIRECTED]->(m)
RETURN a,d,r,m
```

This is the part of the graph we just updated:



In most cases, we want to connect new data to existing structures. This requires that we know how to find existing patterns in our graph data, which we will look at next.

Matching Patterns

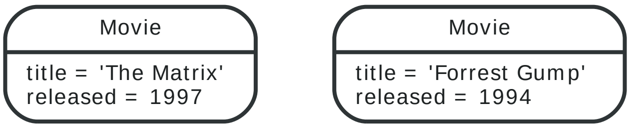
Matching patterns is a task for the `MATCH` statement. We pass the same kind of patterns we've used so far to `MATCH` to describe what we're looking for. It is similar to *query by example*, only that our examples also include the structures.

> **Note**
> A `MATCH` statement will search for the patterns we specify and return *one row per successful pattern match*.

To find the data we've created so far, we can start looking for all nodes labeled with the `Movie` label.

```
MATCH (m:Movie)
RETURN m
```

Here's the result:

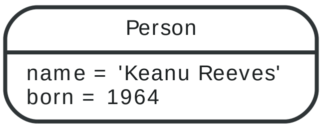


This should show both *The Matrix* and *Forrest Gump*.

We can also look for a specific person, like *Keanu Reeves*.

```
MATCH (p:Person { name:"Keanu Reeves" })
RETURN p
```

This query returns the matching node:



Note that we only provide enough information to find the nodes, not all properties are required. In most cases you have key-properties like SSN, ISBN, emails, logins, geolocation or product codes to look for.

We can also find more interesting connections, like for instance the movies titles that *Tom Hanks* acted in and the roles he played.

```
MATCH (p:Person { name:"Tom Hanks" })-[r:ACTED_IN]->(m:Movie)
```

```
RETURN m.title, r.roles
```

m.title	r.roles
"Forrest Gump"	["Forrest"]

1 row

In this case we only returned the properties of the nodes and relationships that we were interested in. You can access them everywhere via a dot notation `identifer.property`.

Of course this only lists his role as *Forrest* in *Forrest Gump* because that's all data that we've added.

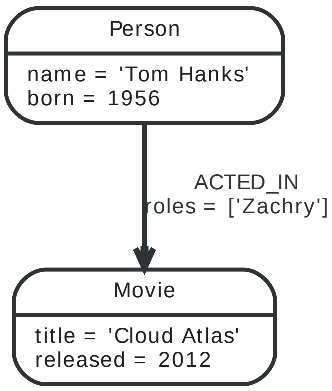
Now we know enough to connect new nodes to existing ones and can combine `MATCH` and `CREATE` to attach structures to the graph.

Attaching Structures

To extend the graph with new information, we first match the existing connection points and then attach the newly created nodes to them with relationships. Adding *Cloud Atlas* as a new movie for *Tom Hanks* could be achieved like this:

```
MATCH (p:Person { name:"Tom Hanks" })
CREATE (m:Movie { title:"Cloud Atlas",released:2012 })
CREATE (p)-[r:ACTED_IN { roles: ['Zachry']}]->(m)
RETURN p,r,m
```

Here's what the structure looks like in the database:



Tip
It is important to remember that we can assign identifiers to both nodes and relationships and use them later on, no matter if they were created or matched.

It is possible to attach both node and relationship in a single `CREATE` clause. For readability it helps to split them up though.

Important
A tricky aspect of the combination of `MATCH` and `CREATE` is that we get *one row per matched pattern*. This causes subsequent `CREATE` statements to be executed once for each row. In many cases this is what you want. If that's not intended, please move the `CREATE` statement before the `MATCH`, or change the cardinality of the query with means discussed later or use the *get or create* semantics of the next clause: `MERGE`.

Completing Patterns

Whenever we get data from external systems or are not sure if certain information already exists in the graph, we want to be able to express a repeatable (idempotent) update operation. In Cypher `MERGE` has this function. It acts like a combination of `MATCH` *or* `CREATE`, which checks for the existence of data first before creating it. With `MERGE` you define a pattern to be found or created. Usually, as with `MATCH` you only want to include the key property to look for in your core pattern. `MERGE` allows you to provide additional properties you want to set `ON CREATE`.

If we wouldn't know if our graph already contained *Cloud Atlas* we could merge it in again.

```
MERGE (m:Movie { title:"Cloud Atlas" })
ON CREATE SET m.released = 2012
RETURN m
```

m
Node[5]{title:"Cloud Atlas",released:2012}

1 row

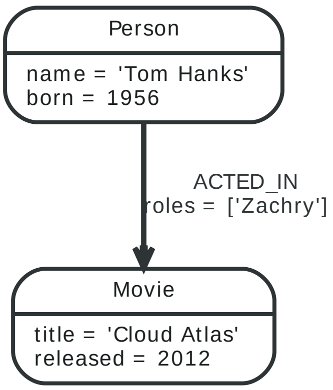
We get a result in any both cases: either the data (potentially more than one row) that was already in the graph or a single, newly created `Movie` node.

Note
A `MERGE` clause without any previously assigned identifiers in it either matches the full pattern or creates the full pattern. It never produces a partial mix of matching and creating within a pattern. To achieve a partial match/create, make sure to use already defined identifiers for the parts that shouldn't be affected.

So foremost `MERGE` makes sure that you can't create duplicate information or structures, but it comes with the cost of needing to check for existing matches first. Especially on large graphs it can be costly to scan a large set of labeled nodes for a certain property. You can alleviate some of that by creating supporting indexes or constraints, which we'll discuss later. But it's still not for free, so whenever you're sure to not create duplicate data use `CREATE` over `MERGE`.

Tip
`MERGE` can also assert that a relationship is only created once. For that to work you *have to pass in* both nodes from a previous pattern match.

```
MATCH (m:Movie { title:"Cloud Atlas" })
MATCH (p:Person { name:"Tom Hanks" })
MERGE (p)-[r:ACTED_IN]->(m)
ON CREATE SET r.roles =['Zachry']
RETURN p,r,m
```

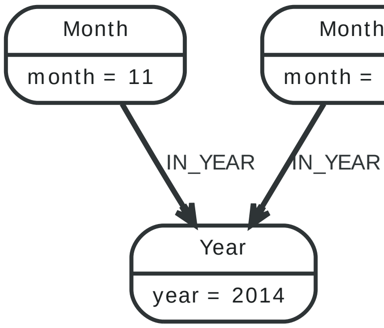


In case the direction of a relationship is arbitrary, you can leave off the arrowhead. `MERGE` will then check for the relationship in either direction, and create a new directed relationship if no matching relationship was found.

If you choose to pass in only one node from a preceding clause, `MERGE` offers an interesting functionality. It will then only match within the direct neighborhood of the provided node for the given pattern, and, if not found create it. This can come in very handy for creating for example tree structures.

```
CREATE (y:Year { year:2014 })
MERGE (y)<-[:IN_YEAR]-(m10:Month { month:10 })
MERGE (y)<-[:IN_YEAR]-(m11:Month { month:11 })
RETURN y,m10,m11
```

This is the graph structure that gets created:



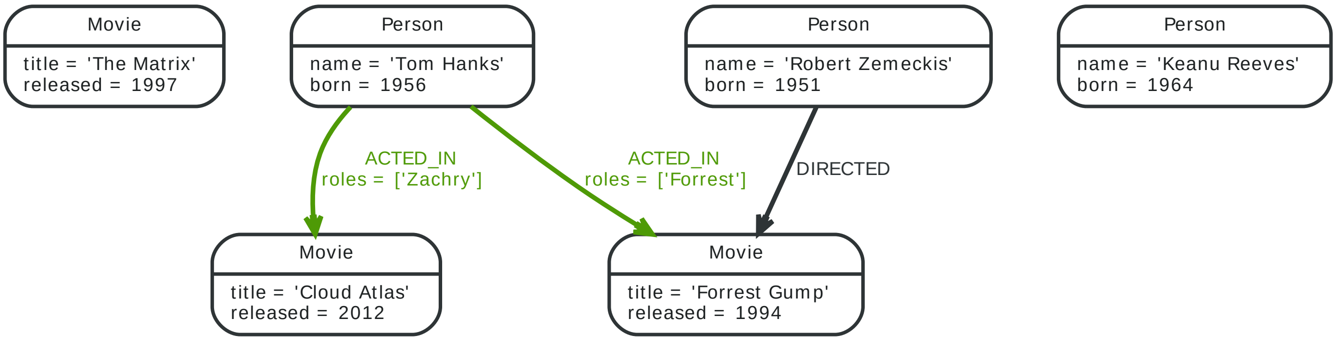
Here there is no global search for the two `Month` nodes; they are only searched for in the context of the *2014* `Year` node.

3.4. Getting the Results You Want

Let's first get some data in to retrieve results from:

```
CREATE (matrix:Movie { title:"The Matrix",released:1997 })
CREATE (cloudAtlas:Movie { title:"Cloud Atlas",released:2012 })
CREATE (forrestGump:Movie { title:"Forrest Gump",released:1994 })
CREATE (keanu:Person { name:"Keanu Reeves", born:1964 })
CREATE (robert:Person { name:"Robert Zemeckis", born:1951 })
CREATE (tom:Person { name:"Tom Hanks", born:1956 })
CREATE (tom)-[:ACTED_IN { roles: ["Forrest"]}]->(forrestGump)
CREATE (tom)-[:ACTED_IN { roles: ['Zachry']}]->(cloudAtlas)
CREATE (robert)-[:DIRECTED]->(forrestGump)
```

This is the data we will start out with:



Filtering Results

So far we've matched patterns in the graph and always returned all results we found. Quite often there are conditions in play for what we want to see. Similar to in *SQL* those filter conditions are expressed in a `WHERE` clause. This clause allows to use any number of boolean expressions (predicates) combined with `AND`, `OR`, `XOR` and `NOT`. The simplest predicates are comparisons, especially equality.

```
MATCH (m:Movie)
WHERE m.title = "The Matrix"
RETURN m
```

m
Node[0]{title:"The Matrix",released:1997}

1 row

For equality on one or more properties, a more compact syntax can be used as well:

```
MATCH (m:Movie { title: "The Matrix" })
RETURN m
```

Other options are numeric comparisons, matching regular expressions and checking the existence of values within a collection.

The `WHERE` clause below includes a regular expression match, a greater than comparison and a test to see if a value exists in a collection.

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE p.name =~ "K.+" OR m.released > 2000 OR "Neo" IN r.roles
RETURN p,r,m
```

p	r	m
Node[5]{name:"Tom Hanks", born:1956}	:ACTED_IN[1]{roles:["Zachry"]}	Node[1]{title:"Cloud Atlas", released:2012}

1 row

One aspect that might be a little surprising is that you can even use patterns as predicates. Where `MATCH` expands the number and shape of patterns matched, a pattern predicate restricts the current result set. It only allows the paths to pass that satisfy the additional patterns as well (or `NOT`).

```
MATCH (p:Person)-[:ACTED_IN]->(m)
WHERE NOT (p)-[:DIRECTED]->()
RETURN p,m
```

p	m
Node[5]{name:"Tom Hanks",born:1956}	Node[1]{title:"Cloud Atlas",released:2012}
Node[5]{name:"Tom Hanks",born:1956}	Node[2]{title:"Forrest Gump",released:1994}

2 rows

Here we find actors, because they sport an `ACTED_IN` relationship but then skip those that ever `DIRECTED` any movie.

There are also more advanced ways of filtering like collection-predicates which we will look at later on.

Returning Results

So far we've returned only nodes, relationships, or paths directly via their identifiers. But the `RETURN` clause can actually return any number of expressions. But what are actually expressions in Cypher?

The simplest expressions are literal values like numbers, strings and arrays as `[1,2,3]`, and maps like `{name:"Tom Hanks", born:1964, movies:["Forrest Gump", ...], count:13}`. You can access individual properties of any node, relationship, or map with a dot-syntax like `n.name`. Individual elements or slices of arrays can be retrieved with subscripts like `names[0]` or `movies[1..-1]`. Each function evaluation like `length(array)`, `toInt("12")`, `substring("2014-07-01",0,4)`, or `coalesce(p.nickname,"n/a")` is also an expression.

Predicates that you'd use in `WHERE` count as boolean expressions.

Of course simpler expressions can be composed and concatenated to form more complex expressions.

By default the expression itself will be used as label for the column, in many cases you want to alias that with a more understandable name using `expression AS alias`. You can later on refer to that column using its alias.

```
MATCH (p:Person)
RETURN p, p.name AS name, upper(p.name), coalesce(p.nickname,"n/a") AS nickname, { name: p.name,
  label:head(labels(p))} AS person
```

p	name	upper(p.name)	nickname	person
Node[3]{name:"Keanu Reeves",born:1964}	"Keanu Reeves"	"KEANU REEVES"	"n/a"	{name -> "Keanu Reeves", label -> "Person"}
Node[4]{name:"Robert Zemeckis", born:1951}	"Robert Zemeckis"	"ROBERT ZEMECKIS"	"n/a"	{name -> "Robert Zemeckis", label -> "Person"}
Node[5]{name:"Tom Hanks",born:1956}	"Tom Hanks"	"TOM HANKS"	"n/a"	{name -> "Tom Hanks", label -> "Person"}

3 rows

If you're interested in unique results you can use the `DISTINCT` keyword after `RETURN` to indicate that.

Aggregating Information

In many cases you want to aggregate or group the data that you encounter while traversing patterns in your graph. In Cypher aggregation happens in the `RETURN` clause while computing your final results. Many common aggregation functions are supported, e.g. `count`, `sum`, `avg`, `min`, and `max`, but there are several more.

Counting the number of people in your database could be achieved by this:

```
MATCH (:Person)
RETURN count(*) AS people
```

people
3

1 row

Please note that `NULL` values are skipped during aggregation. For aggregating only unique values use `DISTINCT`, like in `count(DISTINCT role)`.

Aggregation in Cypher just works. You specify which result columns you want to aggregate and *Cypher will use all non-aggregated columns as grouping keys*.

Aggregation affects which data is still visible in ordering or later query parts.

To find out how often an actor and director worked together, you'd run this statement:

```
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)<-[:DIRECTED]-(director:Person)
RETURN actor,director,count(*) AS collaborations
```

actor	director	collaborations
Node[5]{name:"Tom Hanks", born:1956}	Node[4]{name:"Robert Zemeckis", born:1951}	1

1 row

Frequently you want to sort and paginate after aggregating a `count(x)`.

Ordering and Pagination

Ordering works like in other query languages, with an `ORDER BY expression [ASC|DESC]` clause. The expression can be any expression discussed before as long as it is computable from the returned information.

So for instance if you return `person.name` you can still `ORDER BY person.age` as both are accessible from the `person` reference. You cannot order by things that you can't infer from the information you return. This is especially important with aggregation and `DISTINCT` return values as both remove the visibility of data that is aggregated.

Pagination is a straightforward use of `SKIP {offset} LIMIT {count}`.

A common pattern is to aggregate for a count (score or frequency), order by it and only return the top-n entries.

For instance to find the most prolific actors you could do:

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
RETURN a,count(*) AS appearances
ORDER BY appearances DESC LIMIT 10;
```

a	appearances
Node[5]{name:"Tom Hanks",born:1956}	2

1 row

Collecting Aggregation

The most helpful aggregation function is `collect`, which, as the name says, collects all aggregated values into a *real* array or list. This comes very handy in many situations as you don't loose the detail information while aggregating.

Collect is well suited for retrieving the typical parent-child structures, where one core entity (parent, root or head) is returned per row with all it's dependent information in associated collections created with `collect`. This means there's no need to repeat the parent information per each child-row or even running 1+n statements to retrieve the parent and its children individually.

To retrieve the cast of each movie in our database you could use this statement:

```
MATCH (m:Movie)<-[:ACTED_IN]-(a:Person)
RETURN m.title AS movie, collect(a.name) AS cast, count(*) AS actors
```

movie	cast	actors
"Forrest Gump"	["Tom Hanks"]	1
"Cloud Atlas"	["Tom Hanks"]	1

2 rows

The lists created by collect can either be used from the client consuming the Cypher results or directly within a statement with any of the collection functions or predicates.

3.5. How to Compose Large Statements

Combine statements with UNION

A Cypher statement is usually quite compact. Expressing references between nodes as visual patterns makes them easy to understand.

If you want to combine the results of two statements that have the same result structure, you can use `UNION [ALL]`.

For instance if you want to list both actors and directors without using the alternative relationship-type syntax `()-[:ACTED_IN|:DIRECTED]->()` you can do this:

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
RETURN p,type(r) AS rel,m
UNION
MATCH (p:Person)-[r:DIRECTED]->(m:Movie)
RETURN p,type(r) AS rel,m
```

p	rel	m
(empty result)		

0 row

Use WITH to Chain Statements

In Cypher it's possible to chain fragments of statements together, much like you would do within a data-flow pipeline. Each fragment works on the output from the previous one and its results can feed into the next one.

You use the `WITH` clause to combine the individual parts and declare which data flows from one to the other. `WITH` is very much like `RETURN` with the difference that it doesn't finish a query but prepares the input for the next part. You can use the same expressions, aggregations, ordering and pagination as in the `RETURN` clause.

The only difference is that you *have to* alias all columns as they would otherwise not be accessible with an identifier. Every column that you don't declare in your `WITH` clause is not available in subsequent query parts.

Tip
If you want to filter by an aggregated value in SQL or simlilar languages you would have to use `HAVING`. That's a single purpose clause for filtering aggregated information. In Cypher, `WHERE` can be used in both cases.

3.6. Utilizing Data Structures

Cypher can create and consume more complex data structures out of the box. As already mentioned you can create literal lists (`[1,2,3]`) and maps (`{name: value}`) within a statement.

There is a number of functions that work with lists, from simple ones like `length(list)` that returns the size of a list to

```
MATCH (m:Movie)<-[:ACTED_IN]-(a:Person)
RETURN m.title AS movie, collect(a.name)[0..5] AS five_of_cast
```

movie	five_of_cast
(empty result)	

0 row

You can also access individual elements or slices of a list quickly with `list[1]` or `list[5..-5]`. Other functions to access parts of a list are `head(list)`, `tail(list)` and `last(list)`.

List Predicates

When using lists and arrays in comparisons you can use predicates like `value IN list` or `any(x IN list WHERE x = value)`. There are list predicates to satisfy conditions for `all`, `any`, `none` and `single` elements.

```
MATCH path =(:Person)-->(:Movie)<--(:Person)
WHERE ALL (r IN rels(path) WHERE type(r)= 'ACTED_IN') AND ANY (n IN nodes(path) WHERE n.name = 'Clint
  Eastwood')
RETURN path
```

path
(empty result)

0 row

List Processing

Oftentimes you want to process lists to `filter`, aggregate (`reduce`) or transform (`extract`) their values. Those transformations can be done within Cypher or in the calling code. This kind of list-processing can reduce the amount of data handled and returned, so it might make sense to do it within the Cypher statement.

A simple, non-graph example would be:

```
WITH range(1,10) AS numbers
WITH extract(n IN numbers | n*n) AS squares
WITH filter(n IN squares WHERE n > 25) AS large_squares
RETURN reduce(a = 0, n IN large_squares | a + n) AS sum_large_squares
```

sum_large_squares
330

1 row

In a graph-query you can filter or aggregate collected values instead or work on array properties.

```
MATCH (m:Movie)<-[r:ACTED_IN]-(a:Person)
WITH m.title AS movie, collect({ name: a.name, roles: r.roles }) AS cast
RETURN movie, extract(c2 IN filter(c1 IN cast WHERE c1.name =~ "T.*")| c2.roles)
```

movie	extract(c2 IN filter(c1 IN cast WHERE c1.name =~ "T.*")	c2.roles)
(empty result)		

0 row

Cypher offers to create and consume more complex data structures out of the box. As already mentioned you can create literal lists (`[1,2,3]`) and maps (`{name: value}`) within your statement.

There is a number of functions to work with lists, from simple ones like `length(list)` that returns the size of a list to

```
MATCH (m:Movie)<-[:ACTED_IN]-(a:Person)
RETURN m.title AS movie, collect(a.name)[0..5] AS five_of_cast
```

movie	five_of_cast
(empty result)	

0 row

You can also access individual elements or slices of a list quickly with `list[1]` or `list[5..-5]`. Other functions to access parts of a list are `head(list)`, `tail(list)` and `last(list)`.

Unwind Lists

Sometimes you have collected information into a list, but want to use each element individually as a row. For instance, you might want to further match patterns in the graph. Or you passed in a collection of values but now want to create or match a node or relationship for each element. Then you can use the `UNWIND` clause to unroll a list into a sequence of rows again.

For instance, a query to find the top 5-co-actors and then follow their movies and again list the cast for each of those movies:

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(colleague:Person)
WITH colleague, count(*) AS frequency, collect(DISTINCT m) AS movies
ORDER BY frequency DESC LIMIT 5 UNWIND movies AS m
MATCH (m)<-[:ACTED_IN]-(a)
RETURN m.title AS movie, collect(a.name) AS cast
```

movie	cast
(empty result)	

0 row

3.7. Labels, Constraints and Indexes

Labels are a convenient way to group nodes together. They are used to restrict queries, define constraints and create indexes.

Using Contraints

You can also specify unique constraints that guarantee uniqueness of a certain property on nodes with a specific label.

These constraints are also used by the `MERGE` clause to make certain that a node only exists once.

The following will give an example of how to use labels and add contraints and indexes to them. Let's start out adding a constraint — in this case we decided that all `Movie` node `title`s should be unique.

```
CREATE CONSTRAINT ON (movie:Movie) ASSERT movie.title IS UNIQUE
```

Note that adding the unique constraint will add an index on that property, so we won't do that separately. If we drop a constraint, and still want an index on the same property, we have to create such an index.

Constraints can be added after a label is already in use, but that requires that the existing data complies with the constraints.

Lookup indexes

For a graph query to run fast, you don't need indexes, you only need them to find your starting points. The main reason for using indexes in a graph database is to find the starting points in the graph as fast as possible. After the initial lookups you rely on in-graph structures and the first class citizenship of relationships in the graph database to achieve high performance.

In this case we want an index to speed up finding actors by name in the database:

```
CREATE INDEX ON :Actor(name)
```

Indexes can be added at any time. Note that it will take some time for an index to come online when there's existing data.

Now, let's add some data.

```
CREATE (actor:Actor { name:"Tom Hanks" }),(movie:Movie { title:'Sleepless IN Seattle' }),
  (actor)-[:ACTED_IN]->(movie);
```

Normally you don't specify indexes when querying for data. They will be used automatically. This means we can simply look up the Tom Hanks node, and the index will kick in behind the scenes to boost performance.

```
MATCH (actor:Actor { name: "Tom Hanks" })
RETURN actor;
```

Labels

Now let's say we want to add another label for a node. Here's how to do that:

```
MATCH (actor:Actor { name: "Tom Hanks" })
SET actor :American;
```

To remove a label from nodes, this is what to do:

```
MATCH (actor:Actor { name: "Tom Hanks" })
REMOVE actor:American;
```

Related Content

For more information on labels and related topics, see:

- the section called “Labels” [7]
- Chapter 14, *Schema* [235]
- Section 14.2, “Constraints” [238]
- Section 14.1, “Indexes” [236]
- Section 10.8, “Using” [149]
- Section 12.3, “Set” [191]
- Section 12.5, “Remove” [197]

3.8. Loading Data

As you've seen you can not only query data expressively but also create data with Cypher statements.

Naturally in most cases you wouldn't want to write or generate huge statements to generate your data but instead use an existing data source that you pass into your statement and which is used to drive the graph generation process.

That process not only includes creating completely new data but also integrating with existing structures and updating your graph.

Parameters

In general we recommend passing in varying literal values from the outside as named parameters. This allows Cypher to reuse existing execution plans for the statements.

Of course you can also pass in parameters for data to be imported. Those can be scalar values, maps, lists or even lists of maps.

In your Cypher statement you can then iterate over those values (e.g. with `UNWIND`) to create your graph structures.

For instance to create a movie graph from JSON data structures pulled from an API you could use:

```
{
  "movies" : [ {
    "title" : "Stardust",
    "released" : 2007,
    "cast" : [ {
      "actor" : {
        "name" : "Robert de Niro",
        "born" : 1943
      },
      "characters" : [ "Captain Shakespeare" ]
    }, {
      "actor" : {
        "name" : "Michelle Pfeiffer",
        "born" : 1958
      },
      "characters" : [ "Lamia" ]
    } ]
  } ]
}
```

```
UNWIND {movies} as movie
MERGE (m:Movie {title:movie.title}) ON CREATE SET m.released = movie.released
FOREACH (role IN movie.cast |
   MERGE (a:Person {name:role.actor.name}) ON CREATE SET a.born = role.actor.born
   MERGE (a)-[:ACTED_IN {roles:role.characters}]->(m)
)
```

Importing CSV

Cypher provides an elegant built-in way to import tabular CSV data into graph structures.

The `LOAD CSV` clause parses a local or remote file into a stream of rows which represent maps (with headers) or lists. Then you can use whatever Cypher operations you want to apply to either create nodes or relationships or to merge with existing graph structures.

As CSV files usually represent either node- or relationship-lists, you run multiple passes to create nodes and relationships separately.

movies.csv

```
id,title,country,year
```

```
1,Wall Street,USA,1987
2,The American President,USA,1995
3,The Shawshank Redemption,USA,1994
```

```
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.2.1/csv/intro/movies.csv" AS line
CREATE (m:Movie { id:line.id,title:line.title, released:toInt(line.year)});
```

persons.csv

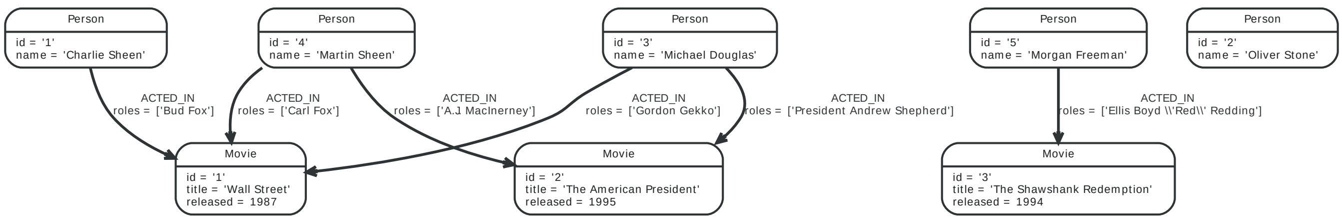
```
id,name
1,Charlie Sheen
2,Oliver Stone
3,Michael Douglas
4,Martin Sheen
5,Morgan Freeman
```

```
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.2.1/csv/intro/persons.csv" AS line
MERGE (a:Person { id:line.id })
ON CREATE SET a.name=line.name;
```

roles.csv

```
personId,movieId,role
1,1,Bud Fox
4,1,Carl Fox
3,1,Gordon Gekko
4,2,A.J. MacInerney
3,2,President Andrew Shepherd
5,3,Ellis Boyd 'Red' Redding
```

```
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.2.1/csv/intro/roles.csv" AS line
MATCH (m:Movie { id:line.movieId })
MATCH (a:Person { id:line.personId })
CREATE (a)-[:ACTED_IN { roles: [line.role]}]->(m);
```



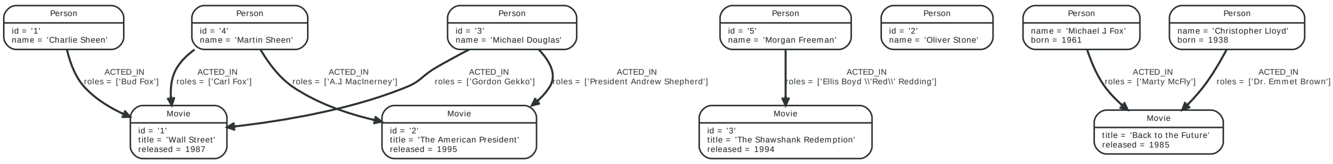
If your file contains denormalized data, you can either run the same file with multiple passes and simple operations as shown above or you might have to use MERGE to create entities uniquely.

For our use-case we can import the data using a CSV structure like this:

movie_actor_roles.csv

```
title;released;actor;born;characters
Back to the Future;1985;Michael J. Fox;1961;Marty McFly
Back to the Future;1985;Christopher Lloyd;1938;Dr. Emmet Brown
```

```
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.2.1/csv/intro/movie_actor_roles.csv" AS line
  FIELDTERMINATOR ";"
MERGE (m:Movie { title:line.title })
ON CREATE SET m.released = toInt(line.released)
MERGE (a:Person { name:line.actor })
ON CREATE SET a.born = toInt(line.born)
MERGE (a)-[:ACTED_IN { roles:split(line.characters,",")}]->(m)
```



If you import a large amount of data (more than 10000 rows), it is recommended to prefix your `LOAD CSV` clause with a `PERIODIC COMMIT` hint. This allows Neo4j to regularly commit the import transactions to avoid memory churn for large transaction-states.

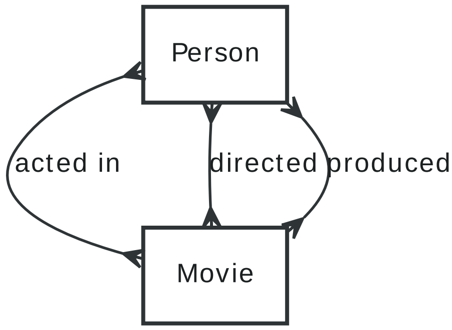
3.9. Cypher vs. SQL

If you have used `SQL` and want to learn Cypher, this chapter is for you! We won't dig very deep into either of the languages, but focus on bridging the gap.

Data Model

For our example, we will use data about persons who *act in, direct, produce* movies.

Here's an entity-relationship model for the example:



We have `Person` and `Movie` entities, which are related in three different ways, each of which have many-to-many cardinality.

In a RDBMS we would use tables for the entities as well as for the associative entities (join tables) needed. In this case we decided to go with the following tables: `movie`, `person`, `acted_in`, `directed`, `produced`. You'll find the SQL for this below.

In Neo4j, the basic data units are nodes and relationships. Both can have properties, which correspond to attributes in a RDBMS.

Nodes can be grouped by putting labels on them. In the example, we will use the labels `Movie` and `Person`.

When using Neo4j, related entities can be represented directly by using relationships. There's no need to deal with foreign keys to handle the relationships, the database will take care of such mechanics. Also, the relationships always have full referential integrity. There's no constraints to enable for this, as it's not optional; it's really part of the underlying data model. Relationships always have a type, and we will differentiate the different kinds of relationships by using the types `ACTED_IN`, `DIRECTED`, `PRODUCED`.

Sample Data

First off, let's see how to set up our example data in a RDBMS. We'll start out creating a few tables and then go on to populate them.

```
CREATE TABLE movie (
  id INTEGER,
  title VARCHAR(100),
  released INTEGER,
  tagline VARCHAR(100)
);
CREATE TABLE person (
  id INTEGER,
  name VARCHAR(100),
  born INTEGER
);
CREATE TABLE acted_in (
  role varchar(100),
  person_id INTEGER,
  movie_id INTEGER
);
CREATE TABLE directed (
```

```
  person_id INTEGER,
  movie_id INTEGER
);
CREATE TABLE produced (
  person_id INTEGER,
  movie_id INTEGER
);
```

Populating with data:

```
INSERT INTO movie (id, title, released, tagline)
VALUES (
  (1, 'The Matrix', 1999, 'Welcome to the Real World'),
  (2, 'The Devil''s Advocate', 1997, 'Evil has its winning ways'),
  (3, 'Monster', 2003, 'The first female serial killer of America')
);
INSERT INTO person (id, name, born)
VALUES (
  (1, 'Keanu Reeves', 1964),
  (2, 'Carrie-Anne Moss', 1967),
  (3, 'Laurence Fishburne', 1961),
  (4, 'Hugo Weaving', 1960),
  (5, 'Andy Wachowski', 1967),
  (6, 'Lana Wachowski', 1965),
  (7, 'Joel Silver', 1952),
  (8, 'Charlize Theron', 1975),
  (9, 'Al Pacino', 1940),
  (10, 'Taylor Hackford', 1944)
);
INSERT INTO acted_in (role, person_id, movie_id)
VALUES (
  ('Neo', 1, 1),
  ('Trinity', 2, 1),
  ('Morpheus', 3, 1),
  ('Agent Smith', 4, 1),
  ('Kevin Lomax', 1, 2),
  ('Mary Ann Lomax', 8, 2),
  ('John Milton', 9, 2),
  ('Aileen', 8, 3)
);
INSERT INTO directed (person_id, movie_id)
VALUES (
  (5, 1),
  (6, 1),
  (10, 2)
);
INSERT INTO produced (person_id, movie_id)
VALUES (
  (7, 1),
  (8, 3)
);
```

Doing this in Neo4j will look quite different. To begin with, we won't create any schema up front. We'll come back to schema later, for now it's enough to know that labels can be used right away without declaring them.

In the `CREATE` statements below, we tell Neo4j what data we want to have in the graph. Simply put, the parentheses denote nodes, while the arrows (-->, or in our case with a relationship type included - `[:DIRECTED]->`) denote relationships. For the nodes we set identifiers like `TheMatrix` so we can easily refer to them later on in the statement. Note that the identifiers are scoped to the statement, and not visible to other Cypher statements. We could use identifiers for the relationships as well, but there's no need for that in this case.

```
CREATE (TheMatrix:Movie { title:'The Matrix', released:1999, tagline:'Welcome to the Real World' })
```

```
CREATE (Keanu:Person { name:'Keanu Reeves', born:1964 })
CREATE (Carrie:Person { name:'Carrie-Anne Moss', born:1967 })
CREATE (Laurence:Person { name:'Laurence Fishburne', born:1961 })
CREATE (Hugo:Person { name:'Hugo Weaving', born:1960 })
CREATE (AndyW:Person { name:'Andy Wachowski', born:1967 })
CREATE (LanaW:Person { name:'Lana Wachowski', born:1965 })
CREATE (JoelS:Person { name:'Joel Silver', born:1952 })
CREATE (Keanu)-[:ACTED_IN { roles: ['Neo']}]->(TheMatrix),
  (Carrie)-[:ACTED_IN { roles: ['Trinity']}]->(TheMatrix),
  (Laurence)-[:ACTED_IN { roles: ['Morpheus']}]->(TheMatrix),
  (Hugo)-[:ACTED_IN { roles: ['Agent Smith']}]->(TheMatrix),(AndyW)-[:DIRECTED]->(TheMatrix),
  (LanaW)-[:DIRECTED]->(TheMatrix),(JoelS)-[:PRODUCED]->(TheMatrix)
CREATE (TheDevilsAdvocate:Movie { title:"The Devil's Advocate", released:1997,
  tagline: 'Evil has its winning ways' })
CREATE (Monster:Movie { title: 'Monster', released: 2003,
  tagline: 'The first female serial killer of America' })
CREATE (Charlize:Person { name:'Charlize Theron', born:1975 })
CREATE (Al:Person { name:'Al Pacino', born:1940 })
CREATE (Taylor:Person { name:'Taylor Hackford', born:1944 })
CREATE (Keanu)-[:ACTED_IN { roles: ['Kevin Lomax']}]->(TheDevilsAdvocate),
  (Charlize)-[:ACTED_IN { roles: ['Mary Ann Lomax']}]->(TheDevilsAdvocate),
  (Al)-[:ACTED_IN { roles: ['John Milton']}]->(TheDevilsAdvocate),
  (Taylor)-[:DIRECTED]->(TheDevilsAdvocate),(Charlize)-[:ACTED_IN { roles: ['Aileen']}]->(Monster),
  (Charlize)-[:PRODUCED { roles: ['Aileen']}]->(Monster)
```

Simple read of data

Let's find all entries in the `movie` table and output their `title` attribute in our RDBMS:

```
SELECT movie.title
FROM movie;
```

TITLE
The Matrix
The Devil's Advocate
Monster

3 rows

Using Neo4j, find all nodes labeled `Movie` and output their `title` property:

```
MATCH (movie:Movie)
RETURN movie.title;
```

movie.title
"The Matrix"
"The Devil's Advocate"
"Monster"

3 rows

`MATCH` tells Neo4j to match a pattern in the graph. In this case the pattern is very simple: any node with a `Movie` label on it. We bind the result of the pattern matching to the identifier `movie`, for use in the `RETURN` clause. And as you can see, the `RETURN` keyword of Cypher is similar to `SELECT` in SQL.

Now let's get movies released after 1998.

```
SELECT movie.title
FROM movie
WHERE movie.released > 1998;
```

TITLE
The Matrix
Monster

2 rows

In this case the addition actually looks identical in Cypher.

```
MATCH (movie:Movie)
WHERE movie.released > 1998
RETURN movie.title;
```

movie.title
"The Matrix"
"Monster"

2 rows

Note however that the semantics of `WHERE` in Cypher is somewhat different, see Section 11.3, “Where” [163] for more information.

Join

Let’s list all persons and the movies they acted in.

```
SELECT person.name, movie.title
FROM person
  JOIN acted_in AS acted_in ON acted_in.person_id = person.id
  JOIN movie ON acted_in.movie_id = movie.id;
```

NAME	TITLE
Keanu Reeves	The Matrix
Keanu Reeves	The Devil's Advocate
Carrie-Anne Moss	The Matrix
Laurence Fishburne	The Matrix
Hugo Weaving	The Matrix
Charlize Theron	The Devil's Advocate
Charlize Theron	Monster
Al Pacino	The Devil's Advocate

8 rows

The same using Cypher:

```
MATCH (person:Person)-[:ACTED_IN]->(movie:Movie)
RETURN person.name, movie.title;
```

Here we match a `Person` and a `Movie` node, in case they are connected with an `ACTED_IN` relationship.

person.name	movie.title
"Hugo Weaving"	"The Matrix"
"Laurence Fishburne"	"The Matrix"
"Carrie-Anne Moss"	"The Matrix"
"Keanu Reeves"	"The Matrix"

8 rows

person.name	movie.title
"Al Pacino"	"The Devil's Advocate"
"Charlize Theron"	"The Devil's Advocate"
"Keanu Reeves"	"The Devil's Advocate"
"Charlize Theron"	"Monster"

8 rows

To make things slightly more complex, let's search for the co-actors of Keanu Reeves. In SQL we use a self join on the `person` table and join on the `acted_in` table once for Keanu, and once for the co-actors.

```
SELECT DISTINCT co_actor.name
FROM person AS keanu
  JOIN acted_in AS acted_in1 ON acted_in1.person_id = keanu.id
  JOIN acted_in AS acted_in2 ON acted_in2.movie_id = acted_in1.movie_id
  JOIN person AS co_actor
    ON acted_in2.person_id = co_actor.id AND co_actor.id <> keanu.id
WHERE keanu.name = 'Keanu Reeves';
```

NAME
Al Pacino
Carrie-Anne Moss
Charlize Theron
Hugo Weaving
Laurence Fishburne

5 rows

In Cypher, we use a pattern with two paths that target the same `Movie` node.

```
MATCH (keanu:Person)-[:ACTED_IN]->(movie:Movie),(coActor:Person)-[:ACTED_IN]->(movie)
WHERE keanu.name = 'Keanu Reeves'
RETURN DISTINCT coActor.name;
```

You may have noticed that we used the `co_actor.id <> keanu.id` predicate in SQL only. This is because Neo4j will only match on the `ACTED_IN` relationship once in the same pattern. If this is not what we want, we can split the pattern up by using two `MATCH` clauses like this:

```
MATCH (keanu:Person)-[:ACTED_IN]->(movie:Movie)
MATCH (coActor:Person)-[:ACTED_IN]->(movie)
WHERE keanu.name = 'Keanu Reeves'
RETURN DISTINCT coActor.name;
```

This time Keanu Reeves is included in the result as well:

coActor.name
"Al Pacino"
"Charlize Theron"
"Keanu Reeves"
"Hugo Weaving"
"Laurence Fishburne"
"Carrie-Anne Moss"

6 rows

Next, let's find out who has both acted in and produced movies.

```
SELECT person.name
FROM person
WHERE person.id IN (SELECT person_id FROM acted_in)
  AND person.id IN (SELECT person_id FROM produced)
```

NAME
Charlize Theron

1 rows

In Cypher, we use patterns as predicates in this case. That is, we require the relationships to exist, but don't care about the connected nodes; thus the empty parentheses.

```
MATCH (person:Person)
WHERE (person)-[:ACTED_IN]->() AND (person)-[:PRODUCED]->()
RETURN person.name
```

Aggregation

Now let's find out a bit about the directors in movies that Keanu Reeves acted in. We want to know how many of those movies each of them directed.

```
SELECT director.name, count(*)
FROM person keanu
  JOIN acted_in ON keanu.id = acted_in.person_id
  JOIN directed ON acted_in.movie_id = directed.movie_id
  JOIN person AS director ON directed.person_id = director.id
WHERE keanu.name = 'Keanu Reeves'
GROUP BY director.name
ORDER BY count(*) DESC
```

NAME	C2
Andy Wachowski	1
Lana Wachowski	1
Taylor Hackford	1

3 rows

Here's how we'll do the same in Cypher:

```
MATCH (keanu:Person { name: 'Keanu Reeves' })-[:ACTED_IN]->(movie:Movie),
  (director:Person)-[:DIRECTED]->(movie)
RETURN director.name, count(*)
ORDER BY count(*) DESC
```

As you can see there is no `GROUP BY` in the Cypher equivalent. Instead, Neo4j will automatically figure out the grouping key.

Chapter 4. Use Cypher in an application

The most direct way to use Cypher programmatically is to execute a HTTP POST operation against the transactional Cypher endpoint. You can send a large number of statements with parameters to the server with each request. For immediate execution you can use the `/db/data/transaction/commit` endpoint with a JSON payload like this:

```
curl -i -H accept:application/json -H content-type:application/json -XPOST http://localhost:7474/db/data/transaction/commit \
  -d '{"statements":[{"statement":"CREATE (p:Person {name:{name},born:{born}}) RETURN p","parameters":{"name":"Keanu
 Reeves","born":1964}}]}'
```

The above command results in:

```
{"results":[{"columns":["p"],"data":[{"row":[{"name":"Keanu Reeves","born":1964}]}]}],"errors":[]}
```

You can add as many "statement" objects in the "statements" list as you want.

For larger use-cases that span multiple requests but whose read-write-read-write operations should be executed within the same transactional scope you'd use the `/db/data/transaction` endpoint. This will give you a transaction URL as the `Location` header, which you can continue to write to and read from. At the end you either commit the whole transaction by POSTing to the (also returned) commit URL or by issuing a DELETE request against the transaction URL.

```
curl -i -H accept:application/json -H content-type:application/json -XPOST http://localhost:7474/db/data/transaction \
  -d '{"statements":[{"statement":"CREATE (p:Person {name:{name},born:{born}}) RETURN p","parameters":{"name":"Clint
 Eastwood","born":1930}}]}'
```

The above command results in:

```
HTTP/1.1 201 Created
Location: http://localhost:7474/db/data/transaction/261

{"commit":"http://localhost:7474/db/data/transaction/261/commit","transaction":{"expires":"Wed, 03 Sep 2014 23:26:51
 +0000"},"errors":[],
 "results":[{"columns":["p"],"data":[{"row":[{"name":"Clint Eastwood","born":1930}]}]}]}
```

See Section 21.1, "Transactional Cypher HTTP endpoint" [286] for more information.

Chapter 5. Basic Data Modeling Examples

The following chapters contain simple examples to get you started thinking about data modeling with graphs. If you are looking for more advanced examples you can head straight to Chapter 6, *Advanced Data Modeling Examples* [61].

The examples use Cypher queries a lot, read Part III, “Cypher Query Language” [101] for more information.

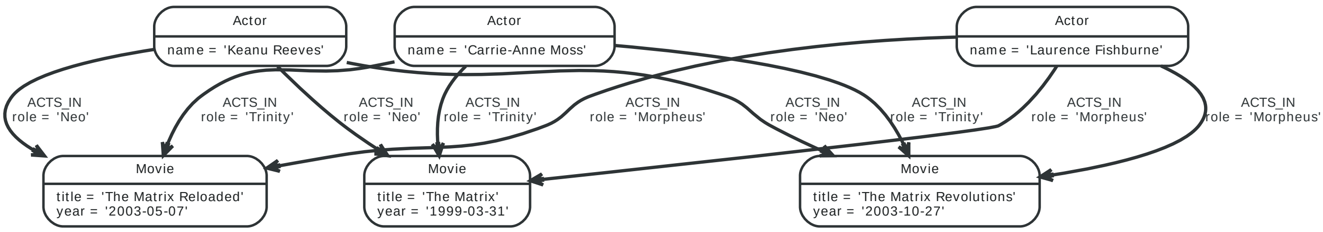
5.1. Movie Database

Our example graph consists of movies with title and year and actors with a name. Actors have `ACTS_IN` relationships to movies, which represents the role they played. This relationship also has a role attribute.

We'll go with three movies and three actors:

```
CREATE (matrix1:Movie { title : 'The Matrix', year : '1999-03-31' })
CREATE (matrix2:Movie { title : 'The Matrix Reloaded', year : '2003-05-07' })
CREATE (matrix3:Movie { title : 'The Matrix Revolutions', year : '2003-10-27' })
CREATE (keanu:Actor { name:'Keanu Reeves' })
CREATE (laurence:Actor { name:'Laurence Fishburne' })
CREATE (carrieanne:Actor { name:'Carrie-Anne Moss' })
CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix1)
CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix2)
CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix3)
CREATE (laurence)-[:ACTS_IN { role : 'Morpheus' }]->(matrix1)
CREATE (laurence)-[:ACTS_IN { role : 'Morpheus' }]->(matrix2)
CREATE (laurence)-[:ACTS_IN { role : 'Morpheus' }]->(matrix3)
CREATE (carrieanne)-[:ACTS_IN { role : 'Trinity' }]->(matrix1)
CREATE (carrieanne)-[:ACTS_IN { role : 'Trinity' }]->(matrix2)
CREATE (carrieanne)-[:ACTS_IN { role : 'Trinity' }]->(matrix3)
```

This gives us the following graph to play with:



Let's check how many nodes we have now:

```
MATCH (n)
RETURN "Hello Graph with " + count(*)+ " Nodes!" AS welcome;
```

Return a single node, by name:

```
MATCH (movie:Movie { title: 'The Matrix' })
RETURN movie;
```

Return the title and date of the matrix node:

```
MATCH (movie:Movie { title: 'The Matrix' })
RETURN movie.title, movie.year;
```

Which results in:

movie.title	movie.year
"The Matrix"	"1999-03-31"

1 row

Show all actors:

```
MATCH (actor:Actor)
RETURN actor;
```

Return just the name, and order them by name:

```
MATCH (actor:Actor)
RETURN actor.name
ORDER BY actor.name;
```

Count the actors:

```
MATCH (actor:Actor)
RETURN count(*);
```

Get only the actors whose names end with “s”:

```
MATCH (actor:Actor)
WHERE actor.name =~ ".*s$"
RETURN actor.name;
```

Here’s some exploratory queries for unknown datasets. *Don’t do this on live production databases!*

Count nodes:

```
MATCH (n)
RETURN count(*);
```

Count relationship types:

```
MATCH (n)-[r]->()
RETURN type(r), count(*);
```

type(r)	count(*)
"ACTS_IN"	9

1 row

List all nodes and their relationships:

```
MATCH (n)-[r]->(m)
RETURN n AS FROM , r AS `->`, m AS to;
```

from	->	to
Node[3]{name:"Keanu Reeves"}	:ACTS_IN[2]{role:"Neo"}	Node[2]{title:"The Matrix Revolutions",year:"2003-10-27"}
Node[3]{name:"Keanu Reeves"}	:ACTS_IN[1]{role:"Neo"}	Node[1]{title:"The Matrix Reloaded",year:"2003-05-07"}
Node[3]{name:"Keanu Reeves"}	:ACTS_IN[0]{role:"Neo"}	Node[0]{title:"The Matrix", year:"1999-03-31"}
Node[4]{name:"Laurence Fishburne"}	:ACTS_IN[5]{role:"Morpheus"}	Node[2]{title:"The Matrix Revolutions",year:"2003-10-27"}
Node[4]{name:"Laurence Fishburne"}	:ACTS_IN[4]{role:"Morpheus"}	Node[1]{title:"The Matrix Reloaded",year:"2003-05-07"}
Node[4]{name:"Laurence Fishburne"}	:ACTS_IN[3]{role:"Morpheus"}	Node[0]{title:"The Matrix", year:"1999-03-31"}
Node[5]{name:"Carrie-Anne Moss"}	:ACTS_IN[8]{role:"Trinity"}	Node[2]{title:"The Matrix Revolutions",year:"2003-10-27"}
Node[5]{name:"Carrie-Anne Moss"}	:ACTS_IN[7]{role:"Trinity"}	Node[1]{title:"The Matrix Reloaded",year:"2003-05-07"}

9 rows

from	->	to
Node[5]{name:"Carrie-Anne Moss"}	:ACTS_IN[6]{role:"Trinity"}	Node[0]{title:"The Matrix", year:"1999-03-31"}

9 rows

5.2. Social Movie Database

Our example graph consists of movies with title and year and actors with a name. Actors have `ACTS_IN` relationships to movies, which represents the role they played. This relationship also has a role attribute.

So far, we queried the movie data; now let's *update the graph* too.

```
CREATE (matrix1:Movie { title : 'The Matrix', year : '1999-03-31' })
CREATE (matrix2:Movie { title : 'The Matrix Reloaded', year : '2003-05-07' })
CREATE (matrix3:Movie { title : 'The Matrix Revolutions', year : '2003-10-27' })
CREATE (keanu:Actor { name:'Keanu Reeves' })
CREATE (laurence:Actor { name:'Laurence Fishburne' })
CREATE (carrieanne:Actor { name:'Carrie-Anne Moss' })
CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix1)
CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix2)
CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix3)
CREATE (laurence)-[:ACTS_IN { role : 'Morpheus' }]->(matrix1)
CREATE (laurence)-[:ACTS_IN { role : 'Morpheus' }]->(matrix2)
CREATE (laurence)-[:ACTS_IN { role : 'Morpheus' }]->(matrix3)
CREATE (carrieanne)-[:ACTS_IN { role : 'Trinity' }]->(matrix1)
CREATE (carrieanne)-[:ACTS_IN { role : 'Trinity' }]->(matrix2)
CREATE (carrieanne)-[:ACTS_IN { role : 'Trinity' }]->(matrix3)
```

We will add ourselves, friends and movie ratings.

Here's how to add a node for yourself and return it, let's say your name is "Me":

```
CREATE (me:User { name: "Me" })
RETURN me;
```

me
Node[6]{name:"Me"}

1 row
Nodes created: 1
Properties set: 1
Labels added: 1

Let's check if the node is there:

```
MATCH (me:User { name: "Me" })
RETURN me.name;
```

Add a movie rating:

```
MATCH (me:User { name: "Me" }),(movie:Movie { title: "The Matrix" })
CREATE (me)-[:RATED { stars : 5, comment : "I love that movie!" }]->(movie);
```

Which movies did I rate?

```
MATCH (me:User { name: "Me" }),(me)-[rating:RATED]->(movie)
RETURN movie.title, rating.stars, rating.comment;
```

movie.title	rating.stars	rating.comment
"The Matrix"	5	"I love that movie!"

1 row

We need a friend!

```
CREATE (friend:User { name: "A Friend" })
RETURN friend;
```

Add our friendship idempotently, so we can re-run the query without adding it several times. We return the relationship to check that it has not been created several times.

```
MATCH (me:User { name: "Me" }),(friend:User { name: "A Friend" })
CREATE UNIQUE (me)-[friendship:FRIEND]->(friend)
RETURN friendship;
```

You can rerun the query, see that it doesn't change anything the second time!

Let's update our friendship with a `since` property:

```
MATCH (me:User { name: "Me" })-[friendship:FRIEND]->(friend:User { name: "A Friend" })
SET friendship.since='forever'
RETURN friendship;
```

Let's pretend us being our friend and wanting to see which movies our friends have rated.

```
MATCH (me:User { name: "A Friend" })-[:FRIEND]-(friend)-[rating:RATED]->(movie)
RETURN movie.title, avg(rating.stars) AS stars, collect(rating.comment) AS comments, count(*);
```

movie.title	stars	comments	count(*)
"The Matrix"	5.0	["I love that movie!"]	1

1 row

That's too little data, let's add some more friends and friendships.

```
MATCH (me:User { name: "Me" })
FOREACH (i IN range(1,10)| CREATE (friend:User { name: "Friend " + i }),(me)-[:FRIEND]->(friend));
```

Show all our friends:

```
MATCH (me:User { name: "Me" })-[r:FRIEND]->(friend)
RETURN type(r) AS friendship, friend.name;
```

friendship	friend.name
"FRIEND"	"A Friend"
"FRIEND"	"Friend 7"
"FRIEND"	"Friend 6"
"FRIEND"	"Friend 9"
"FRIEND"	"Friend 8"
"FRIEND"	"Friend 10"
"FRIEND"	"Friend 1"
"FRIEND"	"Friend 2"
"FRIEND"	"Friend 3"
"FRIEND"	"Friend 4"
"FRIEND"	"Friend 5"

11 rows

5.3. Finding Paths

Our example graph consists of movies with title and year and actors with a name. Actors have ACTS_IN relationships to movies, which represents the role they played. This relationship also has a role attribute.

We queried and updated the data so far, now let's *find interesting constellations, a.k.a. paths*.

```
CREATE (matrix1:Movie { title : 'The Matrix', year : '1999-03-31' })
CREATE (matrix2:Movie { title : 'The Matrix Reloaded', year : '2003-05-07' })
CREATE (matrix3:Movie { title : 'The Matrix Revolutions', year : '2003-10-27' })
CREATE (keanu:Actor { name:'Keanu Reeves' })
CREATE (laurence:Actor { name:'Laurence Fishburne' })
CREATE (carrieanne:Actor { name:'Carrie-Anne Moss' })
CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix1)
CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix2)
CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix3)
CREATE (laurence)-[:ACTS_IN { role : 'Morpheus' }]->(matrix1)
CREATE (laurence)-[:ACTS_IN { role : 'Morpheus' }]->(matrix2)
CREATE (laurence)-[:ACTS_IN { role : 'Morpheus' }]->(matrix3)
CREATE (carrieanne)-[:ACTS_IN { role : 'Trinity' }]->(matrix1)
CREATE (carrieanne)-[:ACTS_IN { role : 'Trinity' }]->(matrix2)
CREATE (carrieanne)-[:ACTS_IN { role : 'Trinity' }]->(matrix3)
```

All other movies that actors in "The Matrix" acted in ordered by occurrence:

```
MATCH (:Movie { title: "The Matrix" })<-[:ACTS_IN]-(actor)-[:ACTS_IN]->(movie)
RETURN movie.title, count(*)
ORDER BY count(*) DESC ;
```

movie.title	count(*)
"The Matrix Revolutions"	3
"The Matrix Reloaded"	3

2 rows

Let's see who acted in each of these movies:

```
MATCH (:Movie { title: "The Matrix" })<-[:ACTS_IN]-(actor)-[:ACTS_IN]->(movie)
RETURN movie.title, collect(actor.name), count(*) AS count
ORDER BY count DESC ;
```

movie.title	collect(actor.name)	count
"The Matrix Revolutions"	["Carrie-Anne Moss","Laurence Fishburne","Keanu Reeves"]	3
"The Matrix Reloaded"	["Carrie-Anne Moss","Laurence Fishburne","Keanu Reeves"]	3

2 rows

What about co-acting, that is actors that acted together:

```
MATCH (:Movie { title: "The Matrix"
  })<-[:ACTS_IN]-(actor)-[:ACTS_IN]->(movie)<-[:ACTS_IN]-(colleague)
RETURN actor.name, collect(DISTINCT colleague.name);
```

actor.name	collect(distinct colleague.name)
"Carrie-Anne Moss"	["Laurence Fishburne","Keanu Reeves"]
"Keanu Reeves"	["Carrie-Anne Moss","Laurence Fishburne"]

3 rows

actor.name	collect(distinct colleague.name)
"Laurence Fishburne"	["Carrie-Anne Moss","Keanu Reeves"]

3 rows

Who of those other actors acted most often with anyone from the matrix cast?

```
MATCH (:Movie { title: "The Matrix"
  })<-[:ACTS_IN]-(actor)-[:ACTS_IN]->(movie)<-[:ACTS_IN]-(colleague)
RETURN colleague.name, count(*)
ORDER BY count(*) DESC LIMIT 10;
```

colleague.name	count(*)
"Carrie-Anne Moss"	4
"Keanu Reeves"	4
"Laurence Fishburne"	4

3 rows

Starting with paths, a path is a sequence of nodes and relationships from a start node to an end node.

We know that Trinity loves Neo, but how many paths exist between the two actors? We'll limit the path length of the pattern as it exhaustively searches the graph otherwise. This is done by using `*0..5` in the pattern relationship.

```
MATCH p =(:Actor { name: "Keanu Reeves" })-[:ACTS_IN*0..5]-(:Actor { name: "Carrie-Anne Moss" })
RETURN p, length(p)
LIMIT 10;
```

p	length(p)
[Node[3]{name:"Keanu Reeves"},:ACTS_IN[0]{role:"Neo"},Node[0]{title:"The Matrix", year:"1999-03-31"},:ACTS_IN[6]{role:"Trinity"}, Node[5]{name:"Carrie-Anne Moss"}]	2
[Node[3]{name:"Keanu Reeves"},:ACTS_IN[1]{role:"Neo"},Node[1]{title:"The Matrix Reloaded", year:"2003-05-07"},:ACTS_IN[4]{role:"Morpheus"}, Node[4]{name:"Laurence Fishburne"},:ACTS_IN[3]{role:"Morpheus"},Node[0]{title:"The Matrix", year:"1999-03-31"},:ACTS_IN[6]{role:"Trinity"}, Node[5]{name:"Carrie-Anne Moss"}]	4
[Node[3]{name:"Keanu Reeves"},:ACTS_IN[2]{role:"Neo"},Node[2]{title:"The Matrix Revolutions",year:"2003-10-27"},:ACTS_IN[5]{role:"Morpheus"},Node[4]{name:"Laurence Fishburne"},:ACTS_IN[3]{role:"Morpheus"}, Node[0]{title:"The Matrix", year:"1999-03-31"},:ACTS_IN[6]{role:"Trinity"}, Node[5]{name:"Carrie-Anne Moss"}]	4
[Node[3]{name:"Keanu Reeves"},:ACTS_IN[1]{role:"Neo"},Node[1]{title:"The Matrix Reloaded", year:"2003-05-07"},:ACTS_IN[7]{role:"Trinity"}, Node[5]{name:"Carrie-Anne Moss"}]	2
[Node[3]{name:"Keanu Reeves"},:ACTS_IN[0]{role:"Neo"},Node[0]{title:"The Matrix",	4

9 rows

p	length(p)
year:"1999-03-31"},:ACTS_IN[3]{role:"Morpheus"}, Node[4]{name:"Laurence Fishburne"},:ACTS_IN[4] {role:"Morpheus"},Node[1]{title:"The Matrix Reloaded",year:"2003-05-07"},:ACTS_IN[7] {role:"Trinity"},Node[5]{name:"Carrie-Anne Moss"}]	
[Node[3]{name:"Keanu Reeves"},:ACTS_IN[2] {role:"Neo"},Node[2]{title:"The Matrix Revolutions",year:"2003-10-27"},:ACTS_IN[5] {role:"Morpheus"},Node[4]{name:"Laurence Fishburne"},:ACTS_IN[4]{role:"Morpheus"}, Node[1]{title:"The Matrix Reloaded", year:"2003-05-07"},:ACTS_IN[7]{role:"Trinity"}, Node[5]{name:"Carrie-Anne Moss"}]	4
[Node[3]{name:"Keanu Reeves"},:ACTS_IN[2] {role:"Neo"},Node[2]{title:"The Matrix Revolutions",year:"2003-10-27"},:ACTS_IN[8] {role:"Trinity"},Node[5]{name:"Carrie-Anne Moss"}]	2
[Node[3]{name:"Keanu Reeves"},:ACTS_IN[0] {role:"Neo"},Node[0]{title:"The Matrix", year:"1999-03-31"},:ACTS_IN[3]{role:"Morpheus"}, Node[4]{name:"Laurence Fishburne"},:ACTS_IN[5] {role:"Morpheus"},Node[2]{title:"The Matrix Revolutions",year:"2003-10-27"},:ACTS_IN[8] {role:"Trinity"},Node[5]{name:"Carrie-Anne Moss"}]	4
[Node[3]{name:"Keanu Reeves"},:ACTS_IN[1] {role:"Neo"},Node[1]{title:"The Matrix Reloaded", year:"2003-05-07"},:ACTS_IN[4]{role:"Morpheus"}, Node[4]{name:"Laurence Fishburne"},:ACTS_IN[5] {role:"Morpheus"},Node[2]{title:"The Matrix Revolutions",year:"2003-10-27"},:ACTS_IN[8] {role:"Trinity"},Node[5]{name:"Carrie-Anne Moss"}]	4

9 rows

But that's a lot of data, we just want to look at the names and titles of the nodes of the path.

```
MATCH p =(:Actor { name: "Keanu Reeves" })-[:ACTS_IN*0..5]-(:Actor { name: "Carrie-Anne Moss" })
RETURN extract(n IN nodes(p)| coalesce(n.title,n.name)) AS `names AND titles`, length(p)
ORDER BY length(p)
LIMIT 10;
```

names and titles	length(p)
["Keanu Reeves","The Matrix","Carrie-Anne Moss"]	2
["Keanu Reeves","The Matrix Reloaded","Carrie-Anne Moss"]	2
["Keanu Reeves","The Matrix Revolutions","Carrie-Anne Moss"]	2

9 rows

names and titles	length(p)
["Keanu Reeves","The Matrix Reloaded","Laurence Fishburne","The Matrix","Carrie-Anne Moss"]	4
["Keanu Reeves","The Matrix Revolutions","Laurence Fishburne","The Matrix","Carrie-Anne Moss"]	4
["Keanu Reeves","The Matrix","Laurence Fishburne","The Matrix Reloaded","Carrie-Anne Moss"]	4
["Keanu Reeves","The Matrix Revolutions","Laurence Fishburne","The Matrix Reloaded","Carrie-Anne Moss"]	4
["Keanu Reeves","The Matrix","Laurence Fishburne","The Matrix Revolutions","Carrie-Anne Moss"]	4
["Keanu Reeves","The Matrix Reloaded","Laurence Fishburne","The Matrix Revolutions","Carrie-Anne Moss"]	4

9 rows

5.4. Linked Lists

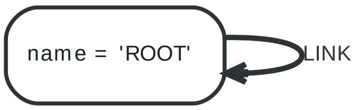
A powerful feature of using a graph database, is that you can create your own in-graph data structures — for example a linked list.

This data structure uses a single node as the list reference. The reference has an outgoing relationship to the head of the list, and an incoming relationship from the last element of the list. If the list is empty, the reference will point to itself.

To make it clear what happens, we will show how the graph looks after each query.

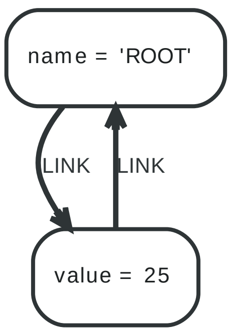
To initialize an empty linked list, we simply create a node, and make it link to itself. Unlike the actual list elements, it doesn't have a `value` property.

```
CREATE (root { name: 'ROOT' })-[:LINK]->(root)
RETURN root
```



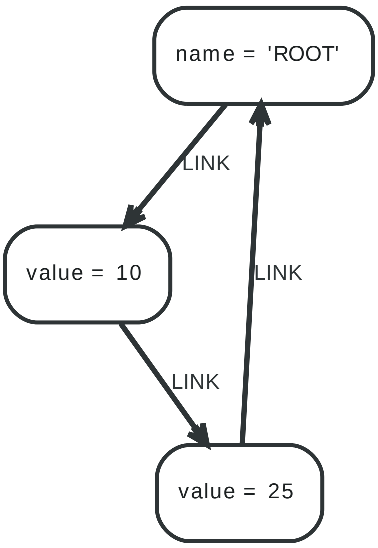
Adding values is done by finding the relationship where the new value should be placed in, and replacing it with a new node, and two relationships to it. We also have to handle the fact that the `before` and `after` nodes could be the same as the `root` node. The case where `before`, `after` and the `root` node are all the same, makes it necessary to use `CREATE UNIQUE` to not create two new value nodes by mistake.

```
MATCH (root)-[:LINK*0..]->(before),(after)-[:LINK*0..]->(root),(before)-[old:LINK]->(after)
WHERE root.name = 'ROOT' AND (before.value < 25 OR before = root) AND (25 < after.value OR after =
  root)
CREATE UNIQUE (before)-[:LINK]->({ value:25 })-[:LINK]->(after)
DELETE old
```



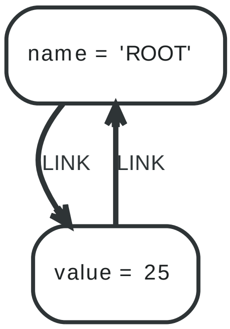
Let's add one more value:

```
MATCH (root)-[:LINK*0..]->(before),(after)-[:LINK*0..]->(root),(before)-[old:LINK]->(after)
WHERE root.name = 'ROOT' AND (before.value < 10 OR before = root) AND (10 < after.value OR after =
  root)
CREATE UNIQUE (before)-[:LINK]->({ value:10 })-[:LINK]->(after)
DELETE old
```



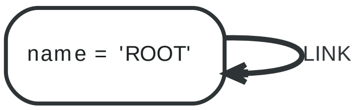
Deleting a value, conversely, is done by finding the node with the value, and the two relationships going in and out from it, and replacing the relationships with a new one.

```
MATCH (root)-[:LINK*0..]->(before),(before)-[delBefore:LINK]->(del)-[delAfter:LINK]->(after),
  (after)-[:LINK*0..]->(root)
WHERE root.name = 'ROOT' AND del.value = 10
CREATE UNIQUE (before)-[:LINK]->(after)
DELETE del, delBefore, delAfter
```



Deleting the last value node is what requires us to use `CREATE UNIQUE` when replacing the relationships. Otherwise, we would end up with two relationships from the `root` node to itself, as both `before` and `after` nodes are equal to the `root` node, meaning the pattern would match twice.

```
MATCH (root)-[:LINK*0..]->(before),(before)-[delBefore:LINK]->(del)-[delAfter:LINK]->(after),
  (after)-[:LINK*0..]->(root)
WHERE root.name = 'ROOT' AND del.value = 25
CREATE UNIQUE (before)-[:LINK]->(after)
DELETE del, delBefore, delAfter
```

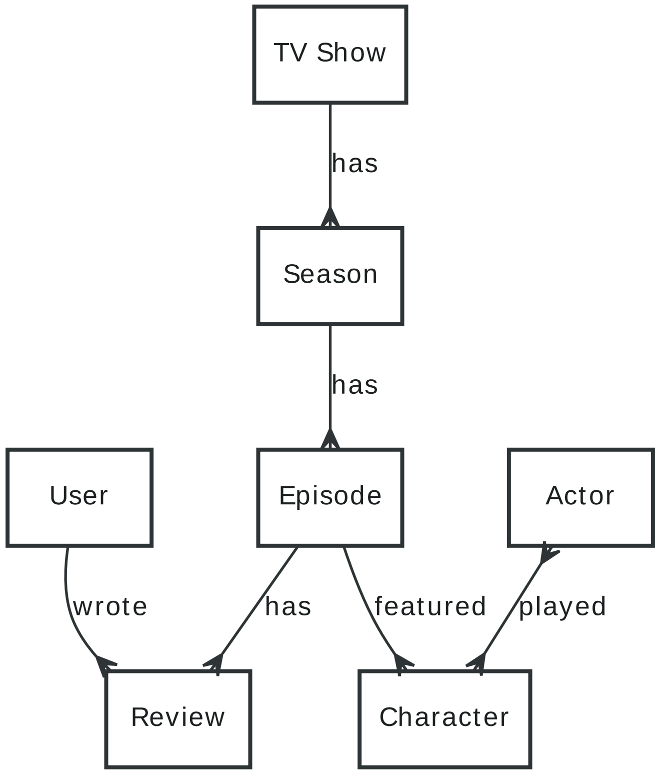


5.5. TV Shows

This example show how TV Shows with Seasons, Episodes, Characters, Actors, Users and Reviews can be modeled in a graph database.

Data Model

Let’s start out with an entity-relationship model of the domain at hand:



To implement this in Neo4j we’ll use the following relationship types:

Relationship Type	Description
HAS_SEASON	Connects a show with its seasons.
HAS_EPISODE	Connects a season with its episodes.
FEATURED_CHARACTER	Connects an episode with its characters.
PLAYED_CHARACTER	Connects actors with characters. Note that an actor can play multiple characters in an episode, and that the same character can be played by multiple actors as well.
HAS_REVIEW	Connects an episode with its reviews.
WROTE_REVIEW	Connects users with reviews they contributed.

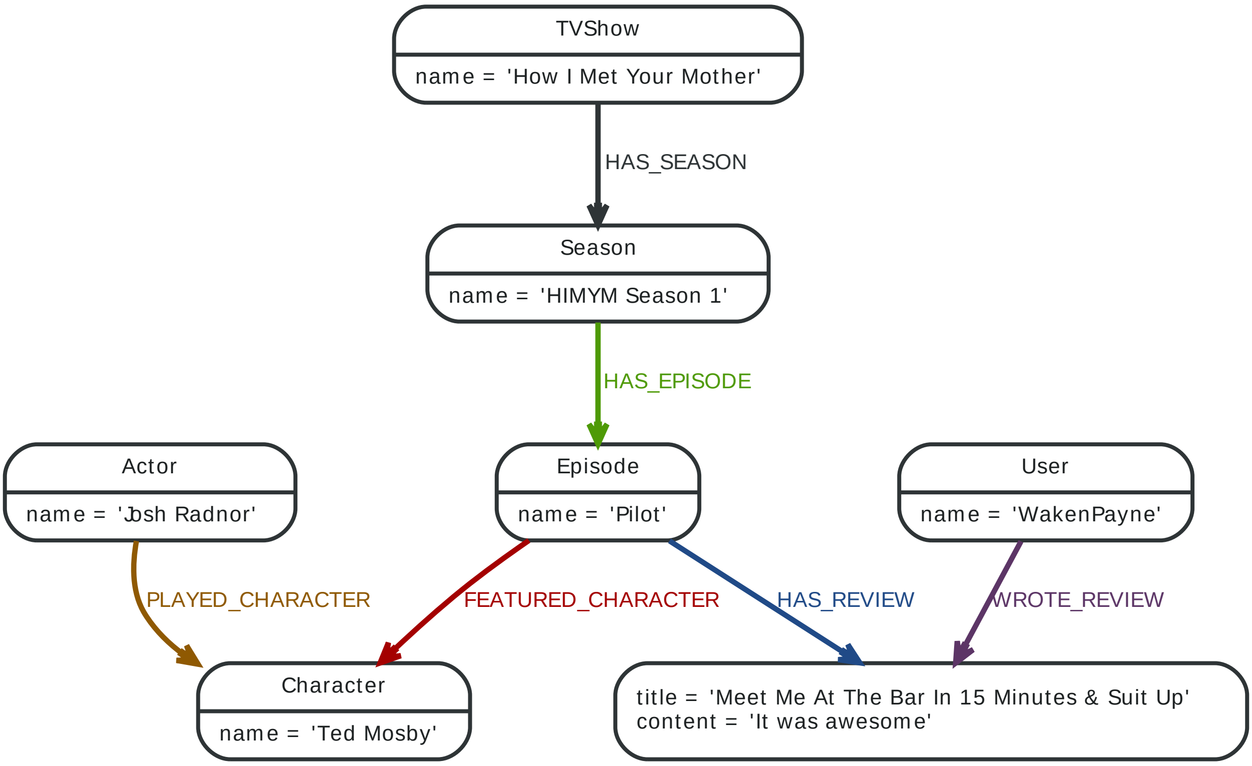
Sample Data

Let’s create some data and see how the domain plays out in practice:

```
CREATE (himym:TVShow { name: "How I Met Your Mother" })
CREATE (himym_s1:Season { name: "HIMYM Season 1" })
CREATE (himym_s1_e1:Episode { name: "Pilot" })
CREATE (ted:Character { name: "Ted Mosby" })
```

```
CREATE (joshRadnor:Actor { name: "Josh Radnor" })
CREATE UNIQUE (joshRadnor)-[:PLAYED_CHARACTER]->(ted)
CREATE UNIQUE (himym)-[:HAS_SEASON]->(himym_s1)
CREATE UNIQUE (himym_s1)-[:HAS_EPISODE]->(himym_s1_e1)
CREATE UNIQUE (himym_s1_e1)-[:FEATURED_CHARACTER]->(ted)
CREATE (himym_s1_e1_review1 { title: "Meet Me At The Bar In 15 Minutes & Suit Up",
  content: "It was awesome" })
CREATE (wakenPayne:User { name: "WakenPayne" })
CREATE (wakenPayne)-[:WROTE_REVIEW]->(himym_s1_e1_review1)<-[:HAS_REVIEW]-(himym_s1_e1)
```

This is how the data looks in the database:



Note that even though we could have modeled the reviews as relationships with title and content properties on them, we made them nodes instead. We gain a lot of flexibility in this way, for example if we want to connect comments to each review.

Now let's add more data:

```
MATCH (himym:TVShow { name: "How I Met Your Mother" }),(himym_s1:Season),
  (himym_s1_e1:Episode { name: "Pilot" }),
  (himym)-[:HAS_SEASON]->(himym_s1)-[:HAS_EPISODE]->(himym_s1_e1)
CREATE (marshall:Character { name: "Marshall Eriksen" })
CREATE (robin:Character { name: "Robin Scherbatsky" })
CREATE (barney:Character { name: "Barney Stinson" })
CREATE (lily:Character { name: "Lily Aldrin" })
CREATE (jasonSegel:Actor { name: "Jason Segel" })
CREATE (cobieSmulders:Actor { name: "Cobie Smulders" })
CREATE (neilPatrickHarris:Actor { name: "Neil Patrick Harris" })
CREATE (alysonHannigan:Actor { name: "Alyson Hannigan" })
CREATE UNIQUE (jasonSegel)-[:PLAYED_CHARACTER]->(marshall)
CREATE UNIQUE (cobieSmulders)-[:PLAYED_CHARACTER]->(robin)
CREATE UNIQUE (neilPatrickHarris)-[:PLAYED_CHARACTER]->(barney)
CREATE UNIQUE (alysonHannigan)-[:PLAYED_CHARACTER]->(lily)
CREATE UNIQUE (himym_s1_e1)-[:FEATURED_CHARACTER]->(marshall)
CREATE UNIQUE (himym_s1_e1)-[:FEATURED_CHARACTER]->(robin)
CREATE UNIQUE (himym_s1_e1)-[:FEATURED_CHARACTER]->(barney)
CREATE UNIQUE (himym_s1_e1)-[:FEATURED_CHARACTER]->(lily)
```

```
CREATE (himym_s1_e1_review2 { title: "What a great pilot for a show :)",
  content: "The humour is great." })
CREATE (atlasredux:User { name: "atlasredux" })
CREATE (atlasredux)-[:WROTE_REVIEW]->(himym_s1_e1_review2)<-[:HAS_REVIEW]-(himym_s1_e1)
```

Information for a show

For a particular TV show, show all the seasons and all the episodes and all the reviews and all the cast members from that show, that is all of the information connected to that TV show.

```
MATCH (tvShow:TVShow)-[:HAS_SEASON]->(season)-[:HAS_EPISODE]->(episode)
WHERE tvShow.name = "How I Met Your Mother"
RETURN season.name, episode.name
```

season.name	episode.name
"HIMYM Season 1"	"Pilot"

1 row

We could also grab the reviews if there are any by slightly tweaking the query:

```
MATCH (tvShow:TVShow)-[:HAS_SEASON]->(season)-[:HAS_EPISODE]->(episode)
WHERE tvShow.name = "How I Met Your Mother"
WITH season, episode
OPTIONAL MATCH (episode)-[:HAS_REVIEW]->(review)
RETURN season.name, episode.name, review
```

season.name	episode.name	review
"HIMYM Season 1"	"Pilot"	Node[5]{title:"Meet Me At The Bar In 15 Minutes & Suit Up", content:"It was awesome"}
"HIMYM Season 1"	"Pilot"	Node[15]{title:"What a great pilot for a show :)", content:"The humour is great."}

2 rows

Now let's list the characters featured in a show. Note that in this query we only put identifiers on the nodes we actually use later on. The other nodes of the path pattern are designated by `()`.

```
MATCH (tvShow:TVShow)-[:HAS_SEASON]->()-[:HAS_EPISODE]->()-[:FEATURED_CHARACTER]->(character)
WHERE tvShow.name = "How I Met Your Mother"
RETURN DISTINCT character.name
```

character.name
"Ted Mosby"
"Marshall Eriksen"
"Robin Scherbatsky"
"Barney Stinson"
"Lily Aldrin"

5 rows

Now let's look at how to get all cast members of a show.

```
MATCH
  (tvShow:TVShow)-[:HAS_SEASON]->()-[:HAS_EPISODE]->(episode)-[:FEATURED_CHARACTER]->()<-[:PLAYED_CHARACTER]-(actor)
WHERE tvShow.name = "How I Met Your Mother"
RETURN DISTINCT actor.name
```

actor.name
"Josh Radnor"
"Jason Segel"
"Cobie Smulders"
"Neil Patrick Harris"
"Alyson Hannigan"

5 rows

Information for an actor

First let's add another TV show that Josh Radnor appeared in:

```
CREATE (er:TVShow { name: "ER" })
CREATE (er_s7:Season { name: "ER S7" })
CREATE (er_s7_e17:Episode { name: "Peter's Progress" })
CREATE (tedMosby:Character { name: "The Advocate " })
CREATE UNIQUE (er)-[:HAS_SEASON]->(er_s7)
CREATE UNIQUE (er_s7)-[:HAS_EPISODE]->(er_s7_e17)
WITH er_s7_e17
MATCH (actor:Actor),(episode:Episode)
WHERE actor.name = "Josh Radnor" AND episode.name = "Peter's Progress"
WITH actor, episode
CREATE (keith:Character { name: "Keith" })
CREATE UNIQUE (actor)-[:PLAYED_CHARACTER]->(keith)
CREATE UNIQUE (episode)-[:FEATURED_CHARACTER]->(keith)
```

And now we'll create a query to find the episodes that he has appeared in:

```
MATCH (actor:Actor)-[:PLAYED_CHARACTER]->(character)<-[:FEATURED_CHARACTER]-(episode)
WHERE actor.name = "Josh Radnor"
RETURN episode.name AS Episode, character.name AS Character
```

Episode	Character
"Pilot"	"Ted Mosby"
"Peter's Progress"	"Keith"

2 rows

Now let's go for a similar query, but add the season and show to it as well.

```
MATCH (actor:Actor)-[:PLAYED_CHARACTER]->(character)<-[:FEATURED_CHARACTER]-(episode),
  (episode)<-[:HAS_EPISODE]-(season)<-[:HAS_SEASON]-(tvshow)
WHERE actor.name = "Josh Radnor"
RETURN tvshow.name AS Show, season.name AS Season, episode.name AS Episode,
  character.name AS Character
```

Show	Season	Episode	Character
"How I Met Your Mother"	"HIMYM Season 1"	"Pilot"	"Ted Mosby"
"ER"	"ER S7"	"Peter's Progress"	"Keith"

2 rows

Chapter 6. Advanced Data Modeling Examples

The following chapters contain 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 nodes, relationships, graph patterns and data locality in traversals.

The examples use Cypher queries a lot, read Part III, “Cypher Query Language” [101] for more information.

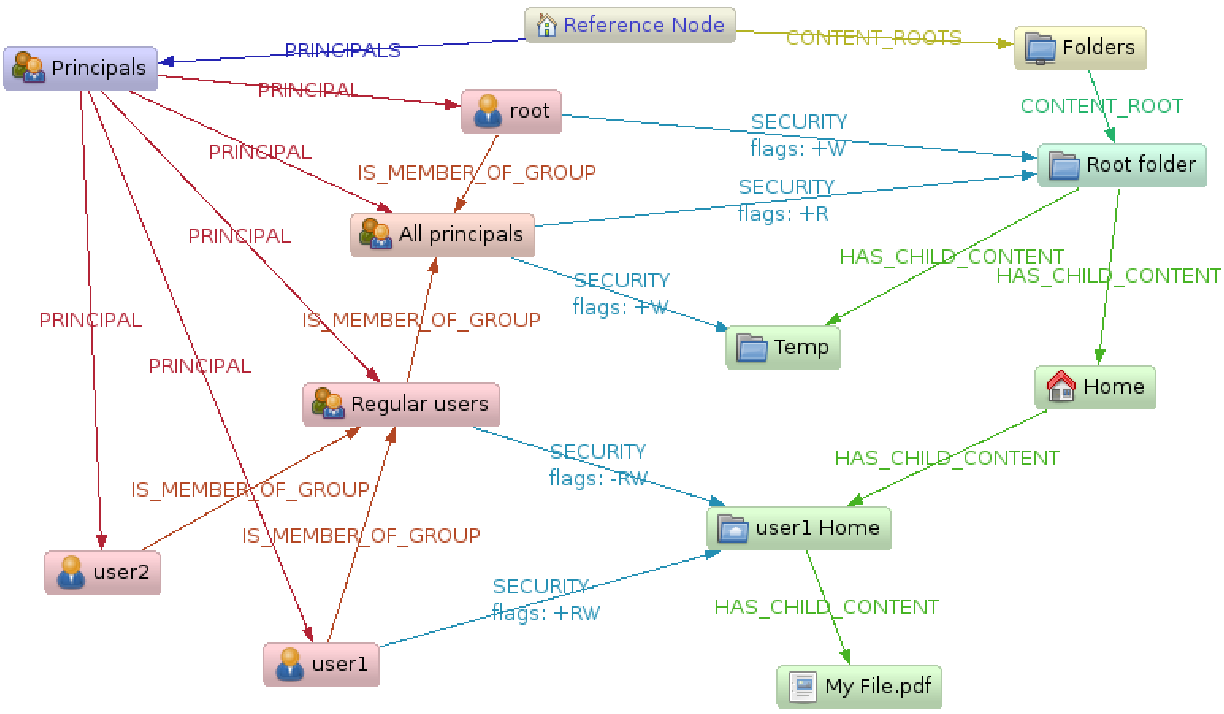
6.1. ACL structures in graphs

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

Generic approach

In many scenarios, an application needs to handle security on some form of managed objects. This example describes one pattern to handle this through the use of a 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 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, without duplicating 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

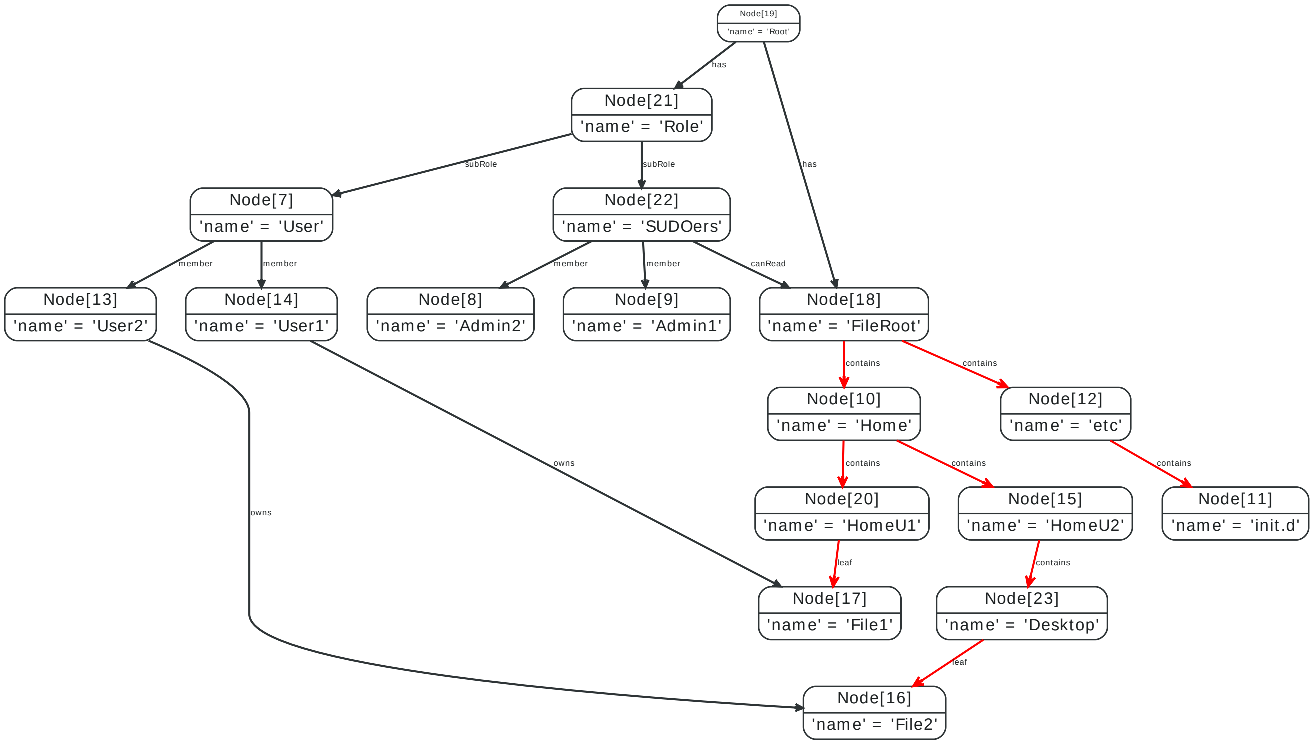
Now, to get the resulting access rights for e.g. "`user 1`" on the "`My File.pdf`" in a Top-Down approach on the model in the graph above would go like:

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`.

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[1].

[1] 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.

```
MATCH ({ name: 'FileRoot' })-[:contains*0..]->(parentDir)-[:leaf]->(file)
RETURN file
```

resulting in:

file
Node[10]{name:"File1"}
Node[9]{name:"File2"}

2 rows

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.

```
MATCH ({ name: 'FileRoot' })-[:contains*0..]->()-[:leaf]->(file)<-[:owns]-(user)
RETURN file, user
```

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

file	user
Node[10]{name:"File1"}	Node[7]{name:"User1"}
Node[9]{name:"File2"}	Node[6]{name:"User2"}

2 rows

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.

```
MATCH (file)<-[:leaf]-()<-[:contains*0..]-(dir)
OPTIONAL MATCH (dir)<-[:canRead]-(role)-[:member]->(readUser)
WHERE file.name =~ 'File.*'
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
"File1"	"HomeU1"	<null>	<null>
"File1"	"Home"	<null>	<null>
"File1"	"FileRoot"	"SUDOers"	"Admin2"
"File1"	"FileRoot"	"SUDOers"	"Admin1"
"File2"	"Desktop"	<null>	<null>
"File2"	"HomeU2"	<null>	<null>
"File2"	"Home"	<null>	<null>
"File2"	"FileRoot"	"SUDOers"	"Admin2"
"File2"	"FileRoot"	"SUDOers"	"Admin1"

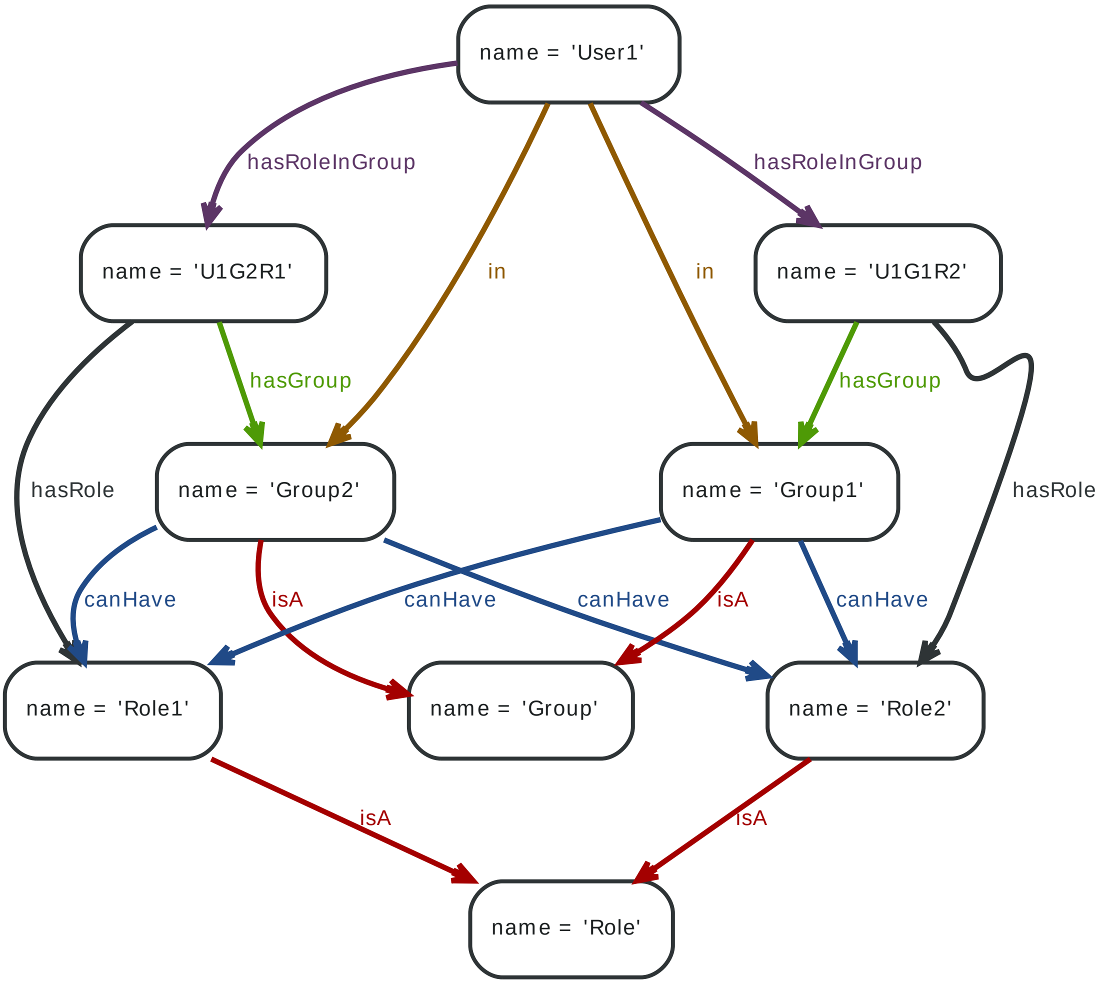
9 rows

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

6.2. Hyperedges

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.

Figure 6.1. Graph



Find Groups

To find out in what roles a user is for a particular groups (here *Group2*), the following query can traverse this HyperEdge node and provide answers.

Query

```
MATCH ({ name: 'User1' })-[:hasRoleInGroup]->(hyperEdge)-[:hasGroup]->({ name: 'Group2' }),
  (hyperEdge)-[:hasRole]->(role)
RETURN role.name
```

The role of `User1` is returned:

Result

role.name
"Role1"
1 row

Find all groups and roles for a user

Here, find all groups and the roles a user has, sorted by the name of the role.

Query

```
MATCH ({ name: 'User1' })-[:hasRoleInGroup]->(hyperEdge)-[:hasGroup]->(group),
  (hyperEdge)-[:hasRole]->(role)
RETURN role.name, group.name
ORDER BY role.name ASC
```

The groups and roles of `User1` are returned:

Result

role.name	group.name
"Role1"	"Group2"
"Role2"	"Group1"

2 rows

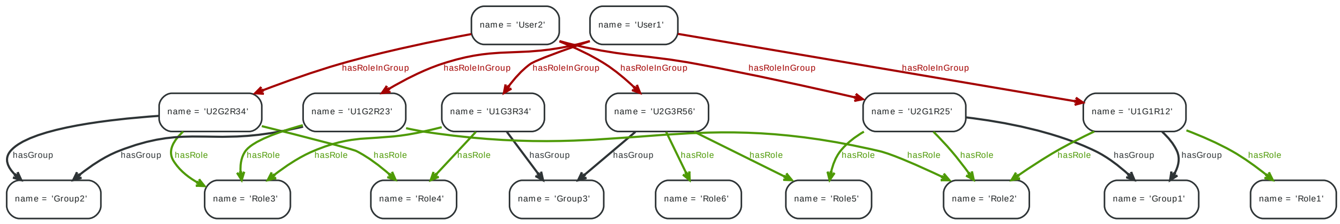
Find common groups based on shared roles

Assume a more complicated graph:

1. Two user nodes `User1`, `User2`.
2. `User1` is in `Group1`, `Group2`, `Group3`.
3. `User1` has `Role1`, `Role2` in `Group1`; `Role2`, `Role3` in `Group2`; `Role3`, `Role4` in `Group3` (hyper edges).
4. `User2` is in `Group1`, `Group2`, `Group3`.
5. `User2` has `Role2`, `Role5` in `Group1`; `Role3`, `Role4` in `Group2`; `Role5`, `Role6` in `Group3` (hyper edges).

The graph for this looks like the following (nodes like `U1G2R23` representing the HyperEdges):

Figure 6.2. Graph



To return `Group1` and `Group2` as `User1` and `User2` share at least one common role in these two groups, the query looks like this:

Query

```
MATCH (u1)-[:hasRoleInGroup]->(hyperEdge1)-[:hasGroup]->(group),(hyperEdge1)-[:hasRole]->(role),
  (u2)-[:hasRoleInGroup]->(hyperEdge2)-[:hasGroup]->(group),(hyperEdge2)-[:hasRole]->(role)
WHERE u1.name = 'User1' AND u2.name = 'User2'
RETURN group.name, count(role)
ORDER BY group.name ASC
```

The groups where `User1` and `User2` share at least one common role:

Result

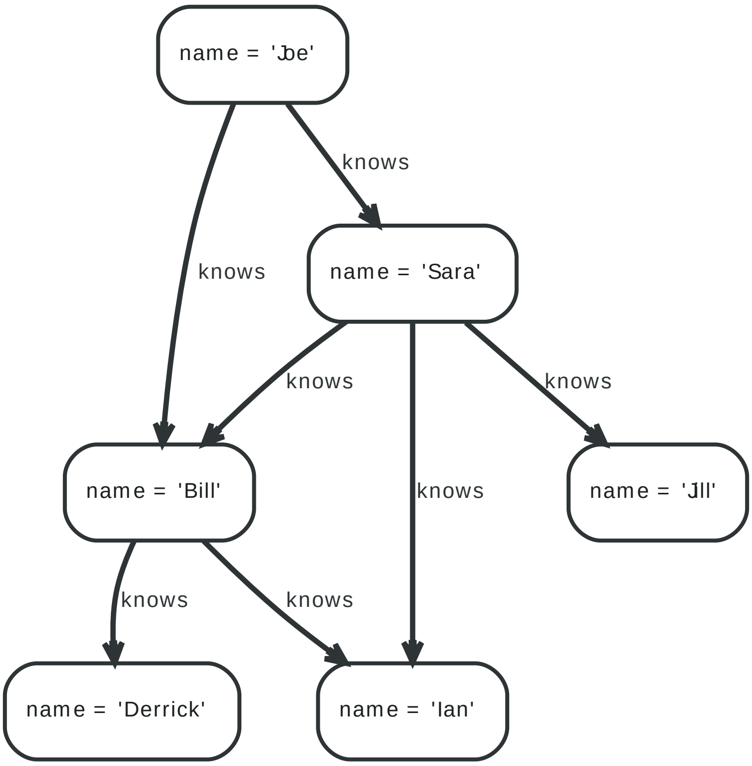
group.name	count(role)
"Group1"	1
"Group2"	1

2 rows

6.3. Basic friend finding based on social neighborhood

Imagine an example graph like the following one:

Figure 6.3. Graph



To find out the friends of Joe's friends that are not already his friends, the query looks like this:

Query

```
MATCH (joe { name: 'Joe' })-[:knows*2..2]-(friend_of_friend)
WHERE NOT (joe)-[:knows]-(friend_of_friend)
RETURN friend_of_friend.name, COUNT(*)
ORDER BY COUNT(*) DESC , friend_of_friend.name
```

This returns a list of friends-of-friends ordered by the number of connections to them, and secondly by their name.

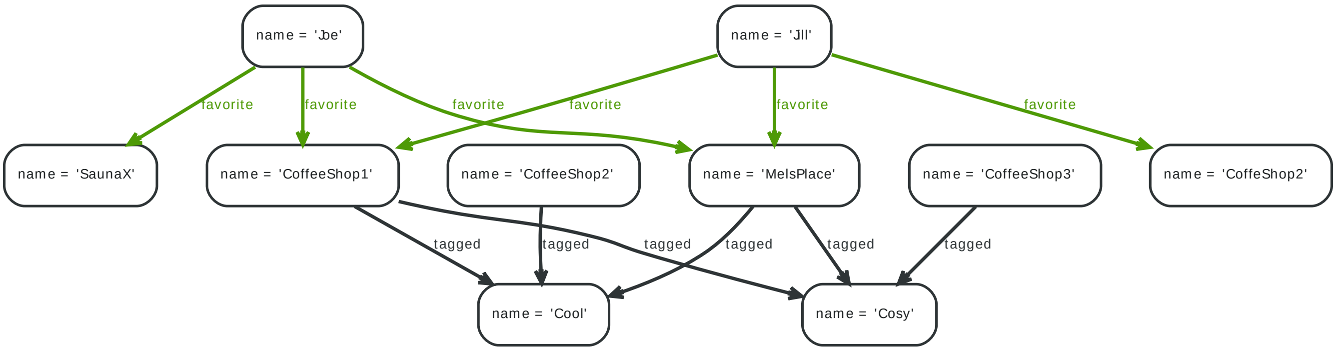
Result

friend_of_friend.name	COUNT(*)
"Ian"	2
"Derrick"	1
"Jill"	1

3 rows

6.4. Co-favorited places

Figure 6.4. 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

```
MATCH (place)<-[:favorite]-(person)-[:favorite]->(stuff)
WHERE place.name = 'CoffeeShop1'
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
"SaunaX"	1

3 rows

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

```
MATCH (place)-[:tagged]->(tag)<-[:tagged]-(otherPlace)
WHERE place.name = 'CoffeeShop1'
RETURN otherPlace.name, collect(tag.name)
ORDER BY length(collect(tag.name)) DESC , otherPlace.name
```

This query returns other places than CoffeeShop1 which share the same tags; they are ranked by the number of tags.

Result

otherPlace.name	collect(tag.name)
"MelsPlace"	["Cosy","Cool"]

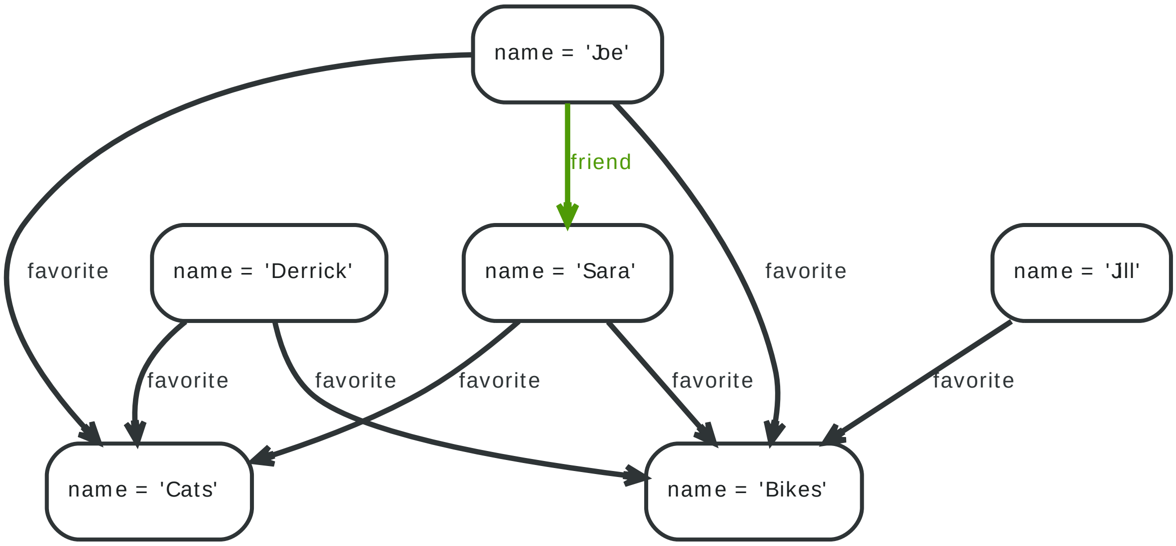
3 rows

otherPlace.name	collect(tag.name)
"CoffeeShop2"	["Cool"]
"CoffeeShop3"	["Cosy"]

3 rows

6.5. Find people based on similar favorites

Figure 6.5. Graph



To find out the possible new friends based on them liking similar things as the asking person, use a query like this:

Query

```
MATCH (me { name: 'Joe' })-[:favorite]->(stuff)<-[:favorite]-(person)
WHERE NOT (me)-[:friend]-(person)
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 is returned.

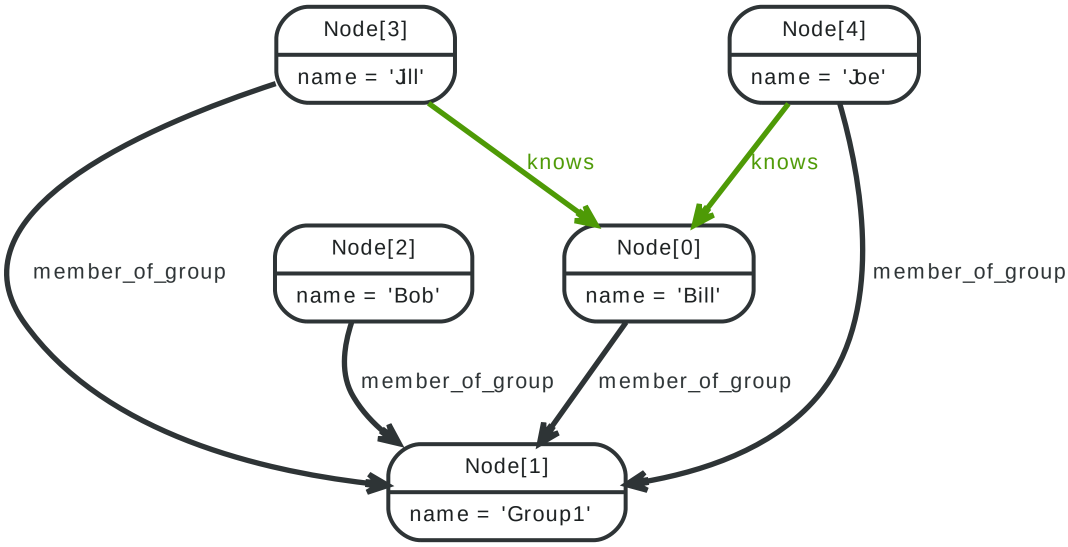
Result

person.name	count(stuff)
"Derrick"	2
"Jill"	1

2 rows

6.6. Find people based on mutual friends and groups

Figure 6.6. Graph



In this scenario, the problem is to determine mutual friends and groups, if any, between persons. If no mutual groups or friends are found, there should be a `0` returned.

Query

```
MATCH (me { name: 'Joe' }),(other)
WHERE other.name IN ['Jill', 'Bob']
OPTIONAL MATCH pGroups=(me)-[:member_of_group]->(mg)<-[:member_of_group]-(other)
OPTIONAL MATCH pMutualFriends=(me)-[:knows]->(mf)<-[:knows]-(other)
RETURN other.name AS name, count(DISTINCT pGroups) AS mutualGroups,
  count(DISTINCT pMutualFriends) AS mutualFriends
ORDER BY mutualFriends DESC
```

The question we are asking is — how many unique paths are there between me and Jill, the paths being common group memberships, and common friends. If the paths are mandatory, no results will be returned if me and Bob lack any common friends, and we don't want that. To make a path optional, you have to make at least one of it's relationships optional. That makes the whole path optional.

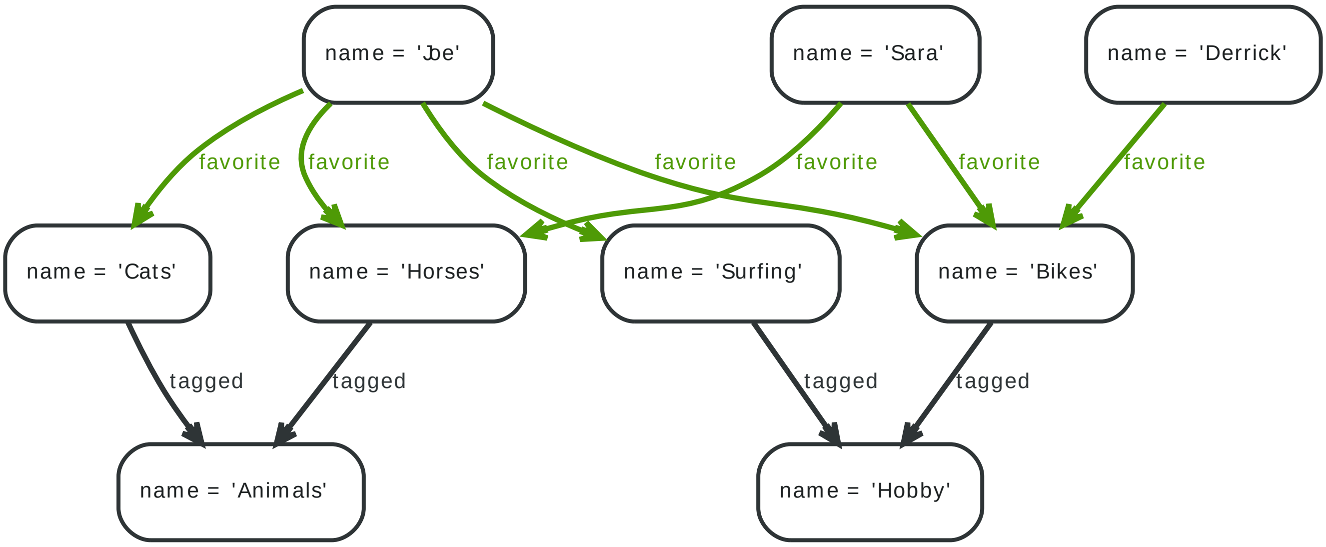
Result

name	mutualGroups	mutualFriends
"Jill"	1	1
"Bob"	1	0

2 rows

6.7. Find friends based on similar tagging

Figure 6.7. Graph



To find people similar to me based on the taggings of their favorited items, one approach could be:

- Determine the tags associated with what I favorite.
- What else is tagged with those tags?
- Who favorites items tagged with the same tags?
- Sort the result by how many of the same things these people like.

Query

```
MATCH
  (me)-[:favorite]->(myFavorites)-[:tagged]->(tag)<-[:tagged]-(theirFavorites)<-[:favorite]-(people)
WHERE me.name = 'Joe' AND NOT me=people
RETURN people.name AS name, count(*) AS similar_favs
ORDER BY similar_favs DESC
```

The query returns the list of possible friends ranked by them liking similar stuff that are not yet friends.

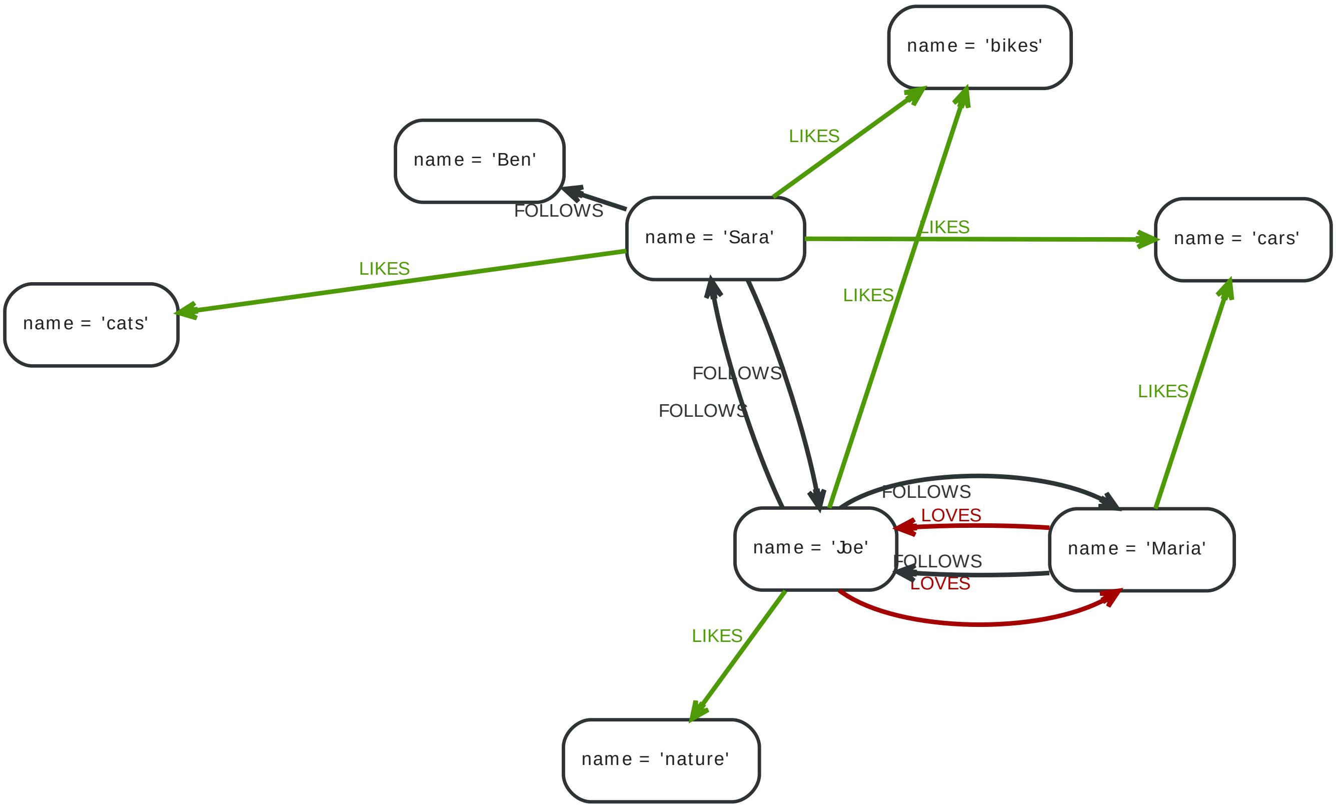
Result

name	similar_favs
"Sara"	2
"Derrick"	1

2 rows

6.8. Multirelational (social) graphs

Figure 6.8. Graph



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

```
MATCH (me { name: 'Joe' })-[r1:FOLLOWS|:LOVES]->(other)-[r2]->(me)
WHERE type(r1)=type(r2)
RETURN other.name, type(r1)
```

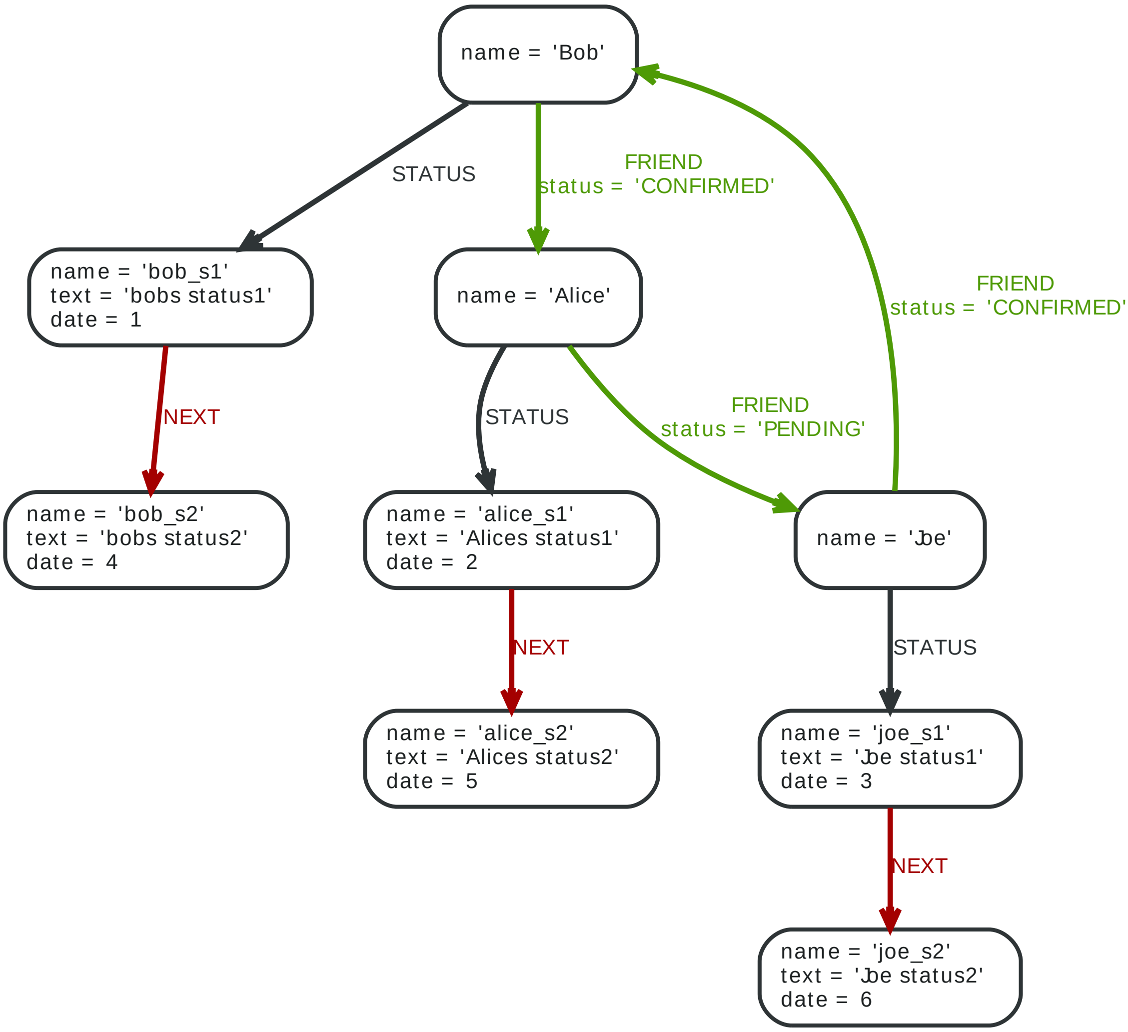
The query returns people that `FOLLOWS` or `LOVES` `Joe` back.

Result

other.name	type(r1)
"Maria"	"FOLLOWS"
"Sara"	"FOLLOWS"
"Maria"	"LOVES"

3 rows

6.9. Implementing newsfeeds in a graph



Implementation of newsfeed or timeline feature is a frequent requirement for social applications. The following exmaples are inspired by Newsfeed feature powered by Neo4j Graph Database[2]. The query asked here is:

Starting at `me`, retrieve the time-ordered status feed of the status updates of me and and all friends that are connected via a `CONFIRMED` `FRIEND` relationship to me.

Query

```
MATCH (me { name: 'Joe' })-[rels:FRIEND*0..1]-(myfriend)
WHERE ALL (r IN rels WHERE r.status = 'CONFIRMED')
WITH myfriend
MATCH (myfriend)-[:STATUS]-(latestupdate)-[:NEXT*0..1]-(statusupdates)
RETURN myfriend.name AS name, statusupdates.date AS date, statusupdates.text AS text
ORDER BY statusupdates.date DESC LIMIT 3
```

To understand the strategy, let's divide the query into five steps:

1. First Get the list of all my friends (along with me) through `FRIEND` relationship (`MATCH (me {name: 'Joe'})-[rels:FRIEND*0..1]-(myfriend)`). Also, the `WHERE` predicate can be added to check whether the friend request is pending or confirmed.

[2] https://web.archive.org/web/20121102191919/http://techfin.in/2012/10/newsfeed-feature-powered-by-neo4j-graph-database/

2. Get the latest status update of my friends through Status relationship (`MATCH myfriend-[:STATUS]-latestupdate`).
3. Get subsequent status updates (along with the latest one) of my friends through `NEXT` relationships (`MATCH (myfriend)-[:STATUS]-(latestupdate)-[:NEXT*0..1]-(statusupdates)`) which will give you the latest and one additional statusupdate; adjust `0..1` to whatever suits your case.
4. Sort the status updates by posted date (`ORDER BY statusupdates.date DESC`).
5. `LIMIT` the number of updates you need in every query (`LIMIT 3`).

Result

name	date	text
"Joe"	6	"Joe status2"
"Bob"	4	"bobs status2"
"Joe"	3	"Joe status1"

3 rows

Here, the example shows how to add a new status update into the existing data for a user.

Query

```
MATCH (me)
WHERE me.name='Bob'
OPTIONAL MATCH (me)-[r:STATUS]-(secondlatestupdate)
DELETE r
CREATE (me)-[:STATUS]->(latest_update { text:'Status',date:123 })
WITH latest_update, collect(secondlatestupdate) AS seconds
FOREACH (x IN seconds | CREATE (latest_update)-[:NEXT]->(x))
RETURN latest_update.text AS new_status
```

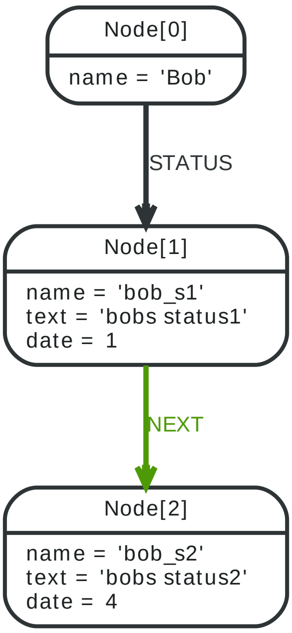
Dividing the query into steps, this query resembles adding new item in middle of a doubly linked list:

1. Get the latest update (if it exists) of the user through the `STATUS` relationship (`OPTIONAL MATCH (me)-[r:STATUS]-(secondlatestupdate)`).
2. Delete the `STATUS` relationship between `user` and `secondlatestupdate` (if it exists), as this would become the second latest update now and only the latest update would be added through a `STATUS` relationship; all earlier updates would be connected to their subsequent updates through a `NEXT` relationship. (`DELETE r`).
3. Now, create the new `statusupdate` node (with text and date as properties) and connect this with the user through a `STATUS` relationship (`CREATE (me)-[:STATUS]->(latest_update { text:'Status',date:123 })`).
4. Pipe over `statusupdate` or an empty collection to the next query part (`WITH latest_update, collect(secondlatestupdate) AS seconds`).
5. Now, create a `NEXT` relationship between the latest status update and the second latest status update (if it exists) (`FOREACH(x in seconds | CREATE (latest_update)-[:NEXT]->(x))`).

Result

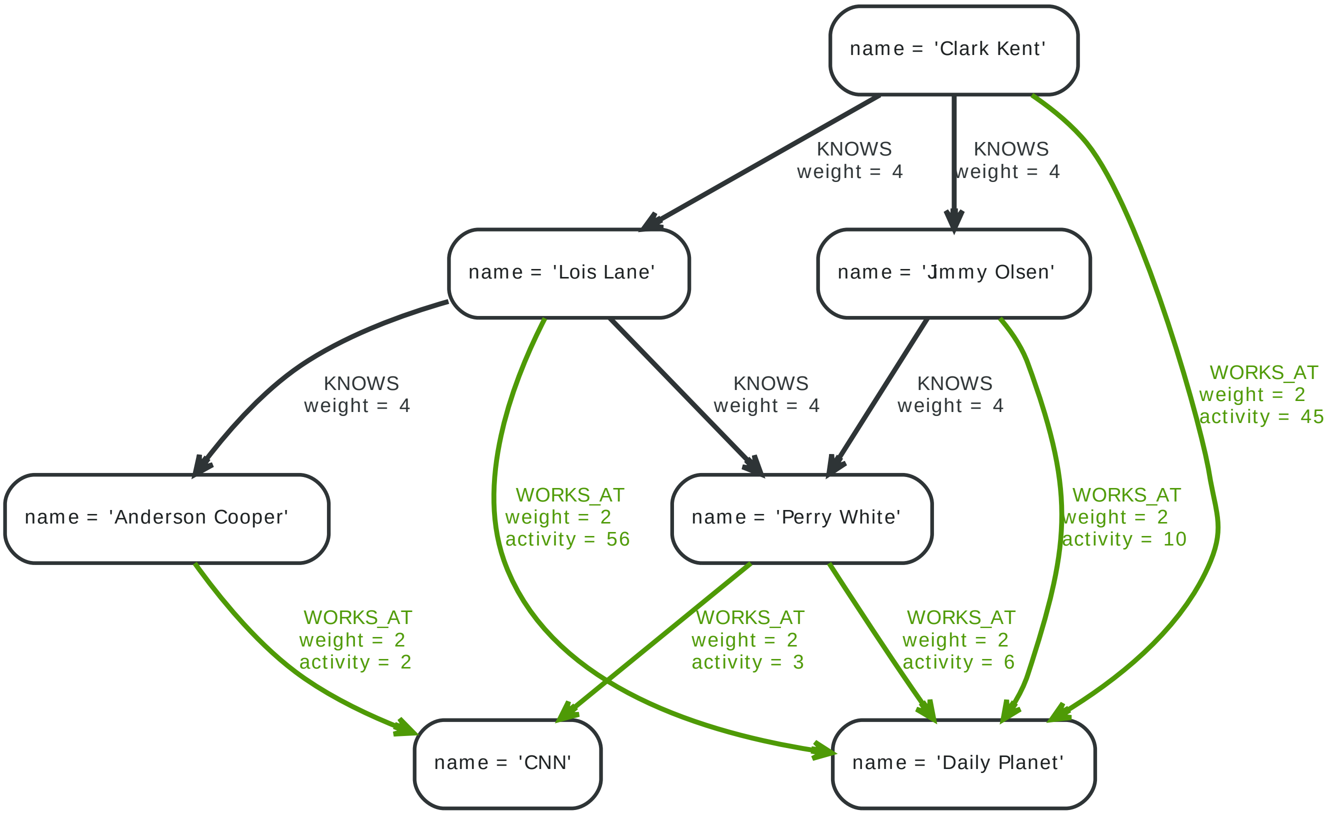
new_status
"Status"

1 row
Nodes created: 1
Relationships created: 2
Properties set: 2
Relationships deleted: 1


Node[0]
name = 'Bob'
STATUS
Node[1]
name = 'bob_s1'
text = 'bobs status1'
date = 1
NEXT
Node[2]
name = 'bob_s2'
text = 'bobs status2'
date = 4

6.10. Boosting recommendation results

Figure 6.9. Graph



This query finds the recommended friends for the origin that are working at the same place as the origin, or know a person that the origin knows, also, the origin should not already know the target. This recommendation is weighted for the weight of the relationship `r2`, and boosted with a factor of 2, if there is an `activity`-property on that relationship

Query

```
MATCH (origin)-[r1:KNOWS|WORKS_AT]-(c)-[r2:KNOWS|WORKS_AT]-(candidate)
WHERE origin.name = "Clark Kent" AND type(r1)=type(r2) AND NOT (origin)-[:KNOWS]-(candidate)
RETURN origin.name AS origin, candidate.name AS candidate, SUM(ROUND(r2.weight
  +(COALESCE(r2.activity,
  0)* 2))) AS boost
ORDER BY boost DESC LIMIT 10
```

This returns the recommended friends for the origin nodes and their recommendation score.

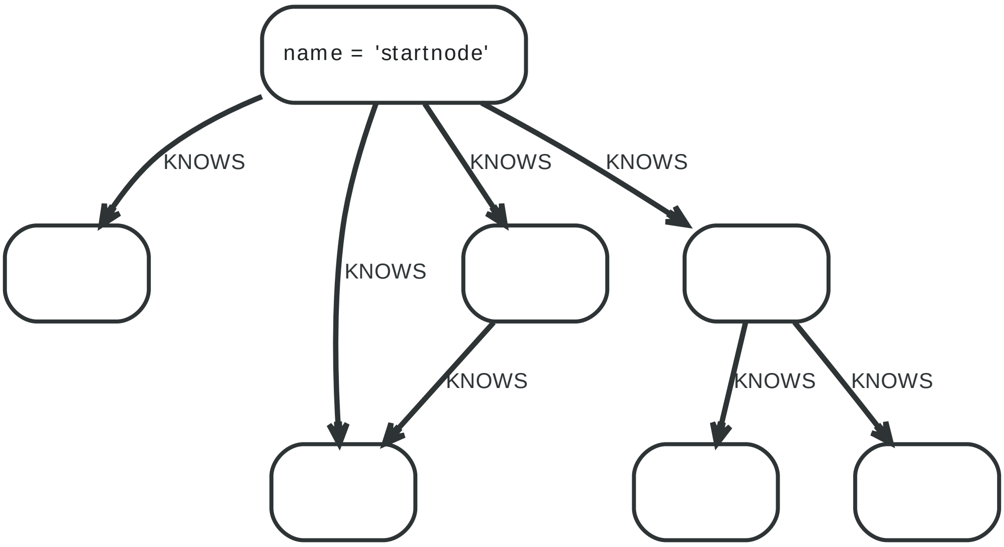
Result

origin	candidate	boost
"Clark Kent"	"Perry White"	22.0
"Clark Kent"	"Anderson Cooper"	4.0

2 rows

6.11. Calculating the clustering coefficient of a network

Figure 6.10. Graph



In this example, adapted from Niko Gamulins blog post on Neo4j for Social Network Analysis[3], the graph in question is showing the 2-hop relationships of a sample person as nodes with `KNOWS` relationships.

The clustering coefficient[4] of a selected node is defined as the probability that two randomly selected neighbors are connected to each other. With the number of neighbors as `n` and the number of mutual connections between the neighbors `r` the calculation is:

The number of possible connections between two neighbors is `n!/(2!(n-2)!) = 4!/(2!(4-2)!) = 24/4 = 6`, where `n` is the number of neighbors `n = 4` and the actual number `r` of connections is `1`. Therefore the clustering coefficient of node 1 is `1/6`.

`n` and `r` are quite simple to retrieve via the following query:

Query

```
MATCH (a { name: "startnode" })--(b)
WITH a, count(DISTINCT b) AS n
MATCH (a)--()-[r]-()--(a)
RETURN n, count(DISTINCT r) AS r
```

This returns `n` and `r` for the above calculations.

Result

n	r
4	1

1 row

3 http://mypetprojects.blogspot.se/2012/06/social-network-analysis-with-neo4j.html
4 http://en.wikipedia.org/wiki/Clustering_coefficient

6.12. Pretty graphs

This section is showing how to create some of the named pretty graphs on Wikipedia[5].

Star graph

The graph is created by first creating a center node, and then once per element in the range, creates a leaf node and connects it to the center.

Query

```
CREATE (center)
FOREACH (x IN range(1,6)| CREATE (leaf),(center)-[:X]->(leaf))
RETURN id(center) AS id;
```

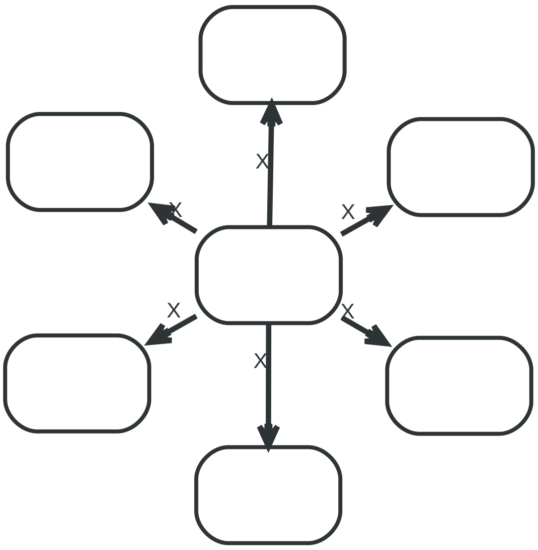
The query returns the id of the center node.

Result

id
0

1 row
Nodes created: 7
Relationships created: 6

Figure 6.11. Graph



Wheel graph

This graph is created in a number of steps:

- Create a center node.
- Once per element in the range, create a leaf and connect it to the center.
- Connect neighboring leafs.
- Find the minimum and maximum leaf and connect these.
- Return the id of the center node.

Query

```
CREATE (center)
```

[5] http://en.wikipedia.org/wiki/Gallery_of_named_graphs

```
FOREACH (x IN range(1,6)| CREATE (leaf { count:x }),(center)-[:X]->(leaf))
WITH center
MATCH (large_leaf)<--(center)-->(small_leaf)
WHERE large_leaf.count = small_leaf.count + 1
CREATE (small_leaf)-[:X]->(large_leaf)
WITH center, min(small_leaf.count) AS min, max(large_leaf.count) AS max
MATCH (first_leaf)<--(center)-->(last_leaf)
WHERE first_leaf.count = min AND last_leaf.count = max
CREATE (last_leaf)-[:X]->(first_leaf)
RETURN id(center) AS id
```

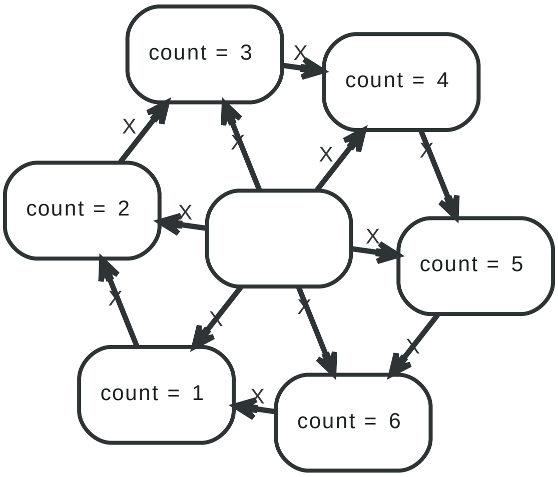
The query returns the id of the center node.

Result

id
0

1 row
Nodes created: 7
Relationships created: 12
Properties set: 6

Figure 6.12. Graph



Complete graph

To create this graph, we first create 6 nodes and label them with the Leaf label. We then match all the unique pairs of nodes, and create a relationship between them.

Query

```
FOREACH (x IN range(1,6)| CREATE (leaf:Leaf { count : x }))
WITH *
MATCH (leaf1:Leaf),(leaf2:Leaf)
WHERE id(leaf1)< id(leaf2)
CREATE (leaf1)-[:X]->(leaf2);
```

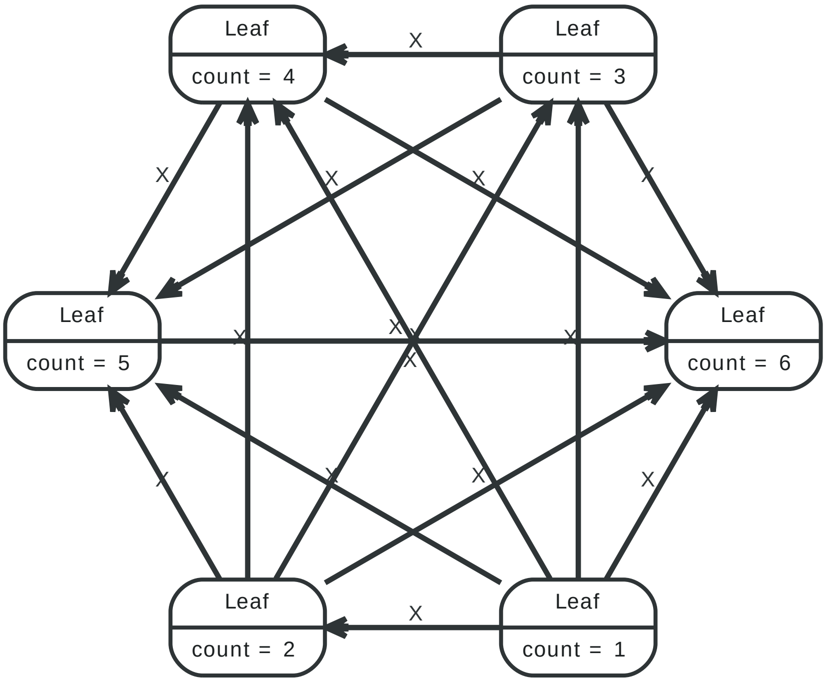
Nothing is returned by this query.

Result

(empty result)

Nodes created: 6
Relationships created: 15
Properties set: 6
Labels added: 6

Figure 6.13. Graph



Friendship graph

This query first creates a center node, and then once per element in the range, creates a cycle graph and connects it to the center

Query

```
CREATE (center)
FOREACH (x IN range(1,3)| CREATE (leaf1),(leaf2),(center)-[:X]->(leaf1),(center)-[:X]->(leaf2),
  (leaf1)-[:X]->(leaf2))
RETURN ID(center) AS id
```

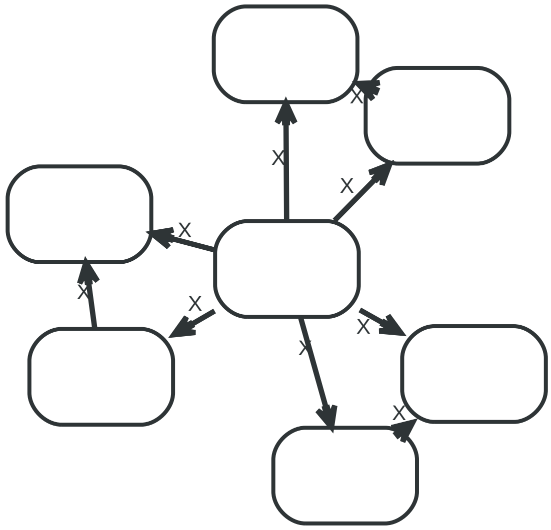
The id of the center node is returned by the query.

Result

id
0

1 row
Nodes created: 7
Relationships created: 9

Figure 6.14. Graph



6.13. A multilevel indexing structure (path tree)

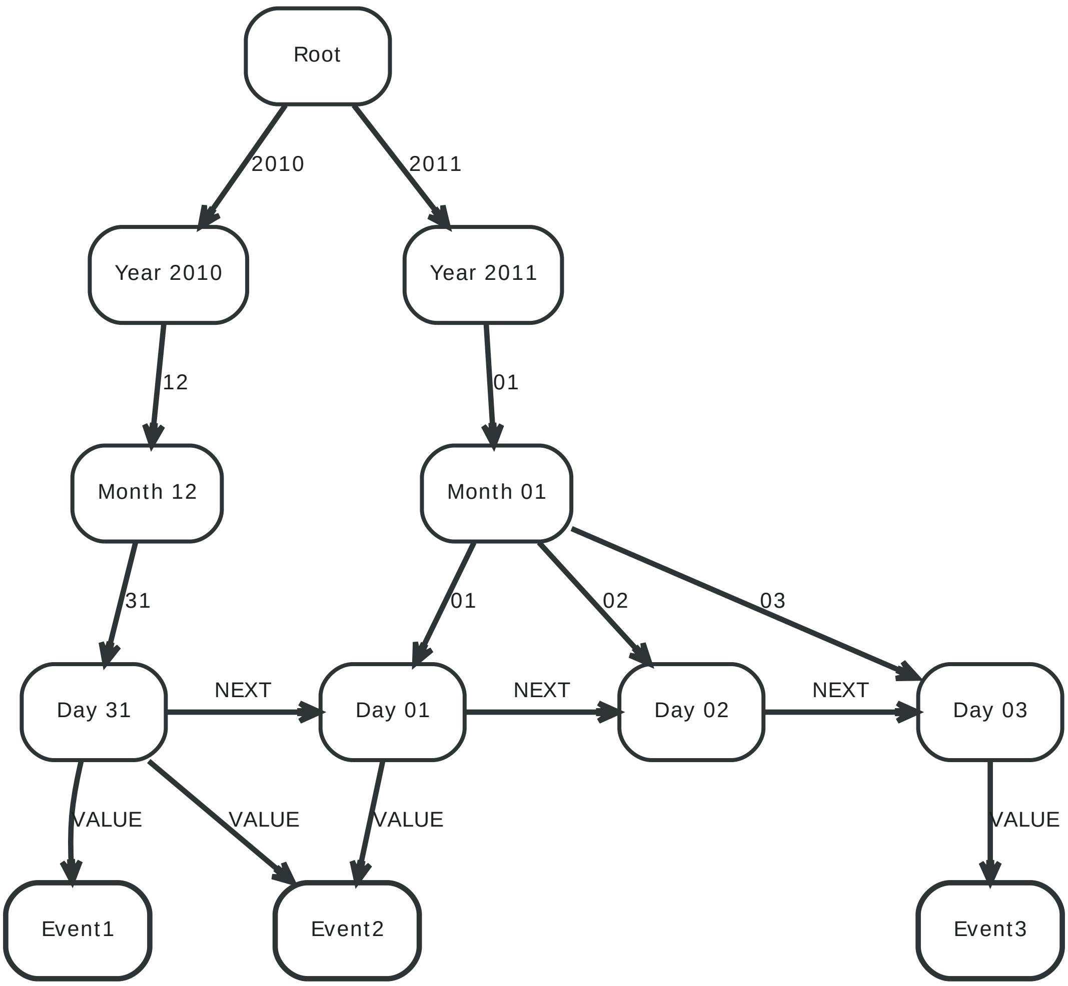
In this example, a multi-level tree structure is used to index event nodes (here `Event1`, `Event2` and `Event3`, in this case with a YEAR-MONTH-DAY granularity, making this a timeline indexing structure. However, this approach should work for a wide range of multi-level ranges.

The structure follows a couple of rules:

- Events can be indexed multiple times by connecting the indexing structure leafs with the events via a `VALUE` relationship.
- The querying is done in a path-range fashion. That is, the start- and end path from the indexing root to the start and end leafs in the tree are calculated
- Using Cypher, the queries following different strategies can be expressed as path sections and put together using one single query.

The graph below depicts a structure with 3 Events being attached to an index structure at different leafs.

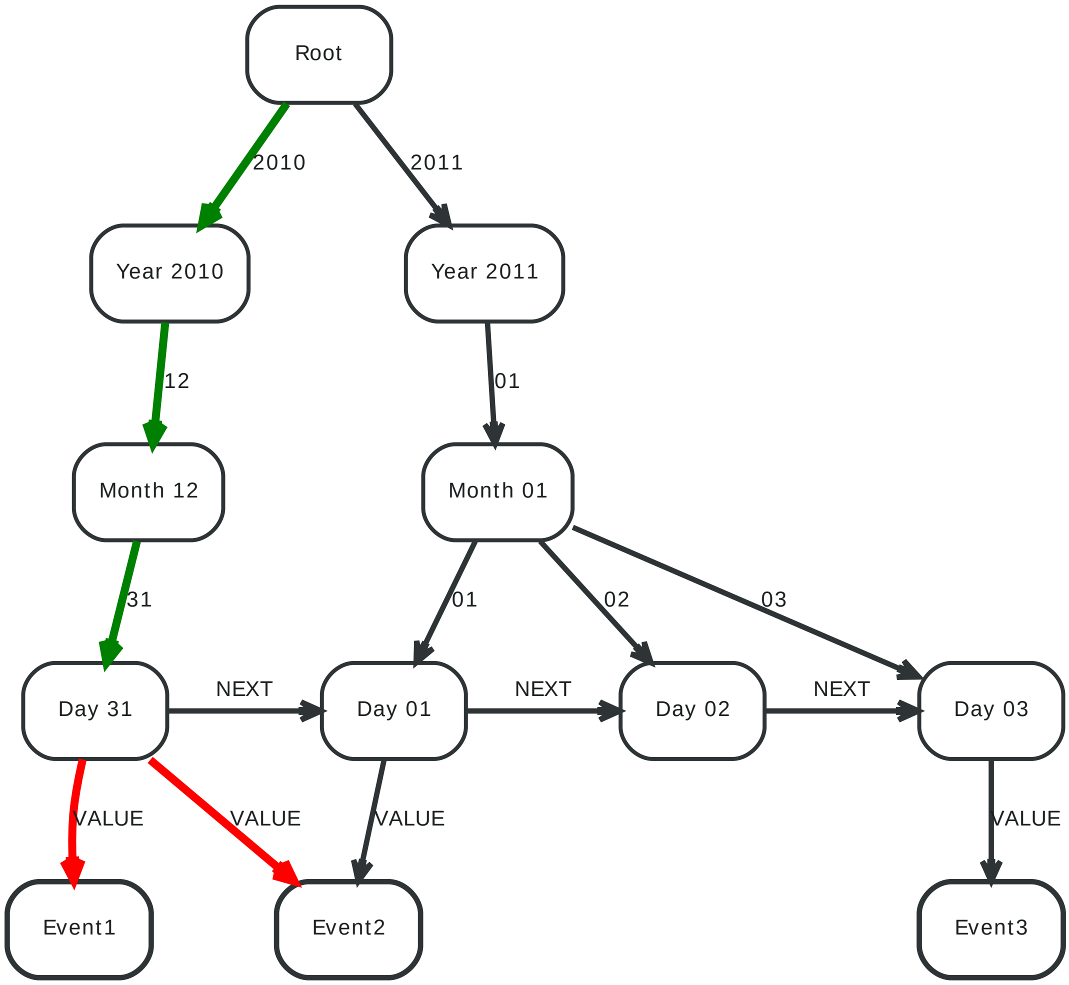
Figure 6.15. Graph



Return zero range

Here, only the events indexed under one leaf (2010-12-31) are returned. The query only needs one path segment `rootPath` (color `Green`) through the index.

Figure 6.16. Graph



Query

```
MATCH rootPath=(root)-[:`2010`]->()-[:`12`]->()-[:`31`]->(leaf),(leaf)-[:VALUE]->(event)
WHERE root.name = 'Root'
RETURN event.name
ORDER BY event.name ASC
```

Returning all events on the date 2010-12-31, in this case `Event1` and `Event2`

Result

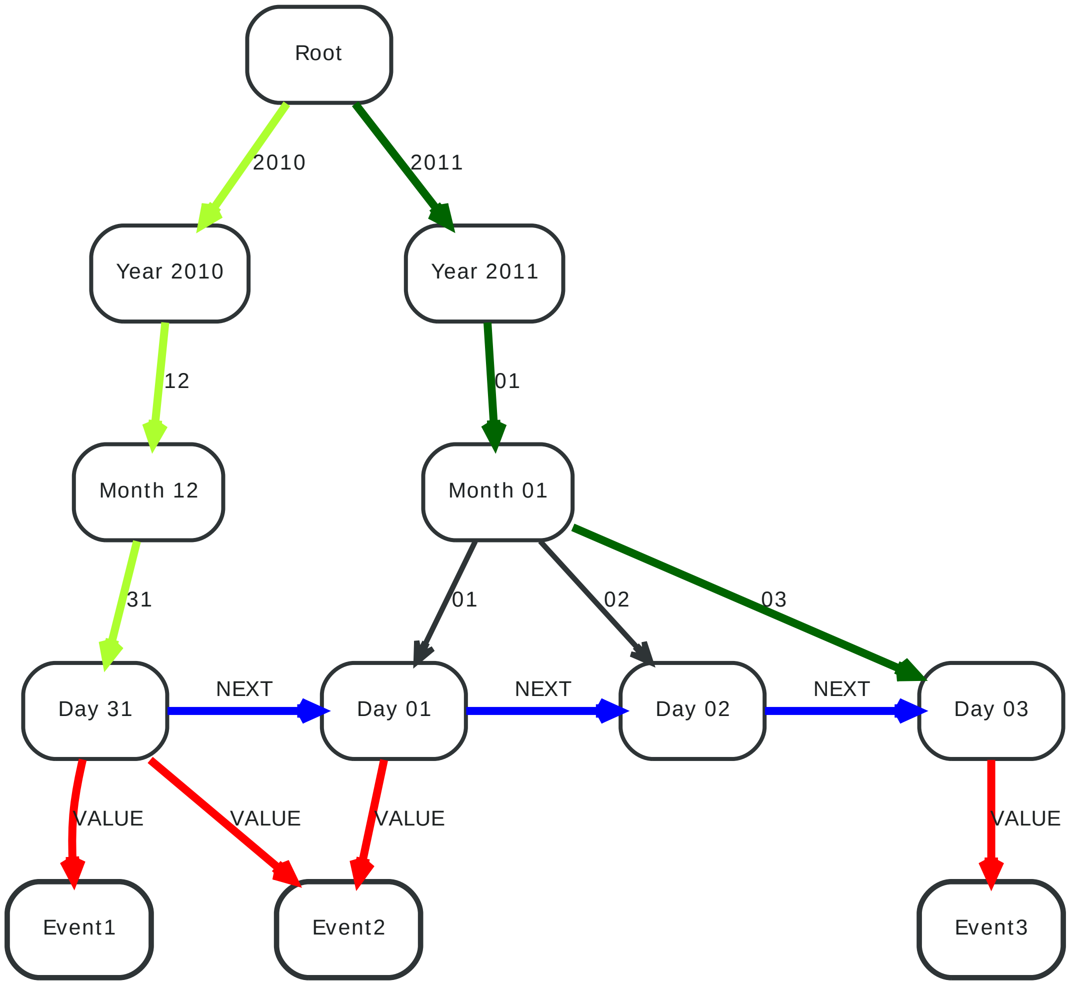
event.name
"Event1"
"Event2"

2 rows

Return the full range

In this case, the range goes from the first to the last leaf of the index tree. Here, `startPath` (color `Greenyellow`) and `endPath` (color `Green`) span up the range, `valuePath` (color `Blue`) is then connecting the leafs, and the values can be read from the `middle` node, hanging off the `values` (color `Red`) path.

Figure 6.17. Graph



Query

```
MATCH startPath=(root)-[:`2010`]->()-[:`12`]->()-[:`31`]->(startLeaf),
  endPath=(root)-[:`2011`]->()-[:`01`]->()-[:`03`]->(endLeaf),
  valuePath=(startLeaf)-[:NEXT*0..]->(middle)-[:NEXT*0..]->(endLeaf),
  vals=(middle)-[:VALUE]->(event)
WHERE root.name = 'Root'
RETURN event.name
ORDER BY event.name ASC
```

Returning all events between 2010-12-31 and 2011-01-03, in this case all events.

Result

event.name
"Event1"
"Event2"
"Event2"
"Event3"

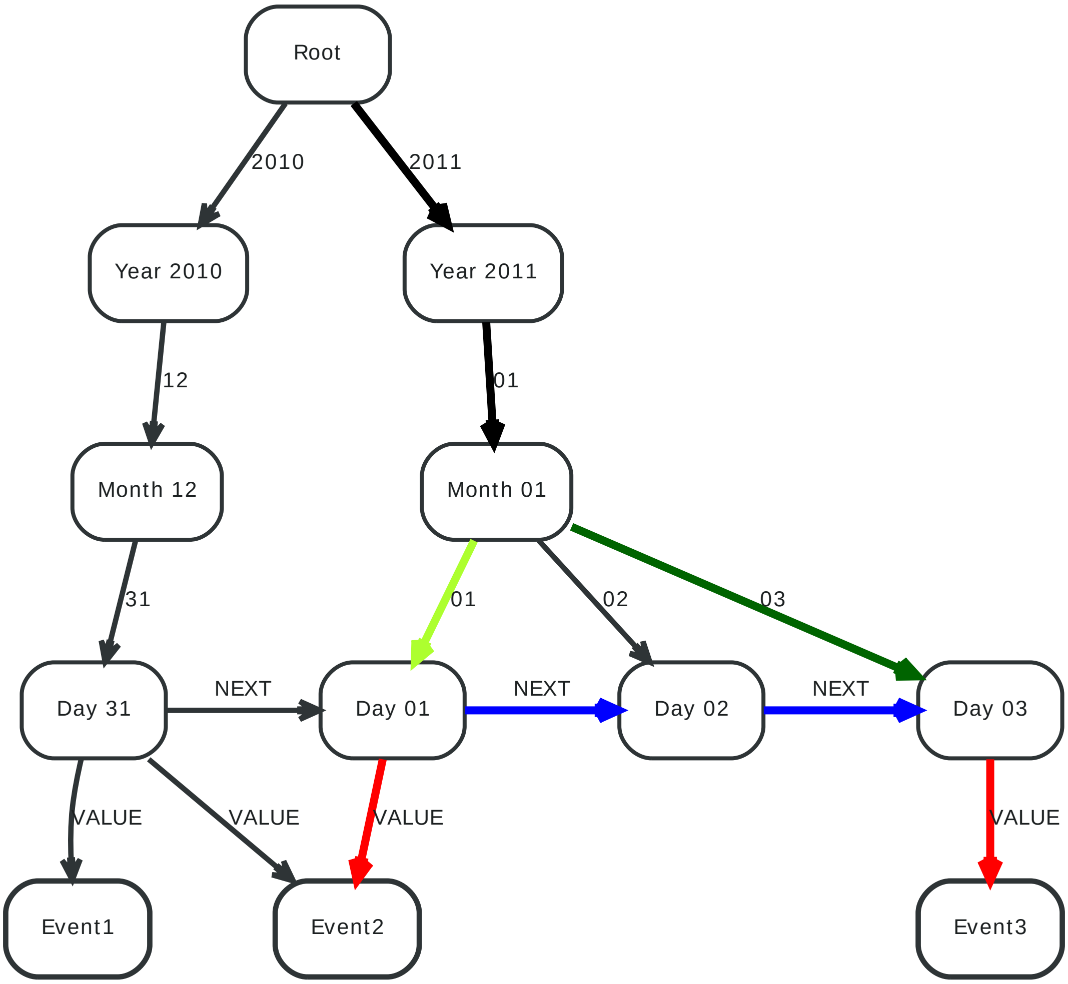
4 rows

Return partly shared path ranges

Here, the query range results in partly shared paths when querying the index, making the introduction of and common path segment `commonPath` (color `Black`) necessary, before spanning up `startPath` (color

Greenyellow) and endPath (color Darkgreen) . After that, valuePath (color Blue) connects the leafs and the indexed values are returned off values (color Red) path.

Figure 6.18. Graph



Query

```
MATCH commonPath=(root)-[:`2011`]->()-[:`01`]->(commonRootEnd),
  startPath=(commonRootEnd)-[:`01`]->(startLeaf), endPath=(commonRootEnd)-[:`03`]->(endLeaf),
  valuePath=(startLeaf)-[:NEXT*0..]->(middle)-[:NEXT*0..]->(endLeaf),
  vals=(middle)-[:VALUE]->(event)
WHERE root.name = 'Root'
RETURN event.name
ORDER BY event.name ASC
```

Returning all events between 2011-01-01 and 2011-01-03, in this case Event2 and Event3.

Result

event.name
"Event2"
"Event3"

2 rows

6.14. Complex similarity computations

Calculate similarities by complex calculations

Here, a similarity between two players in a game is calculated by the number of times they have eaten the same food.

Query

```
MATCH (me { name: 'me' })-[r1:ATE]->(food)<-[r2:ATE]-(you)
WITH me,count(DISTINCT r1) AS H1,count(DISTINCT r2) AS H2,you
MATCH (me)-[r1:ATE]->(food)<-[r2:ATE]-(you)
RETURN sum((1-ABS(r1.times/H1-r2.times/H2))*(r1.times+r2.times)/(H1+H2)) AS similarity
```

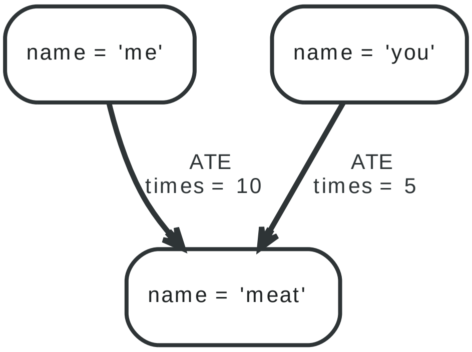
The two players and their similarity measure.

Result

similarity
-30.0

1 row

Figure 6.19. Graph



6.15. The Graphity activity stream model

Find Activity Streams in a network without scaling penalty

This is an approach for scaling the retrieval of activity streams in a friend graph put forward by Rene Pickard as Graphity[6]. In short, a linked list is created for every persons friends in the order that the last activities of these friends have occured. When new activities occur for a friend, all the ordered friend lists that this friend is part of are reordered, transferring computing load to the time of new event updates instead of activity stream reads.

Tip
This approach of course makes excessive use of relationship types. This needs to be taken into consideration when designing a production system with this approach. See Section 17.5, "Capacity" [271] for the maximum number of relationship types.

To find the activity stream for a person, just follow the linked list of the friend list, and retrieve the needed amount of activities form the respective activity list of the friends.

Query

```
MATCH p=(me { name: 'Jane' })-[:jane_knows*]->(friend),(friend)-[:has]->(status)
RETURN me.name, friend.name, status.name, length(p)
ORDER BY length(p)
```

The returns the activity stream for Jane.

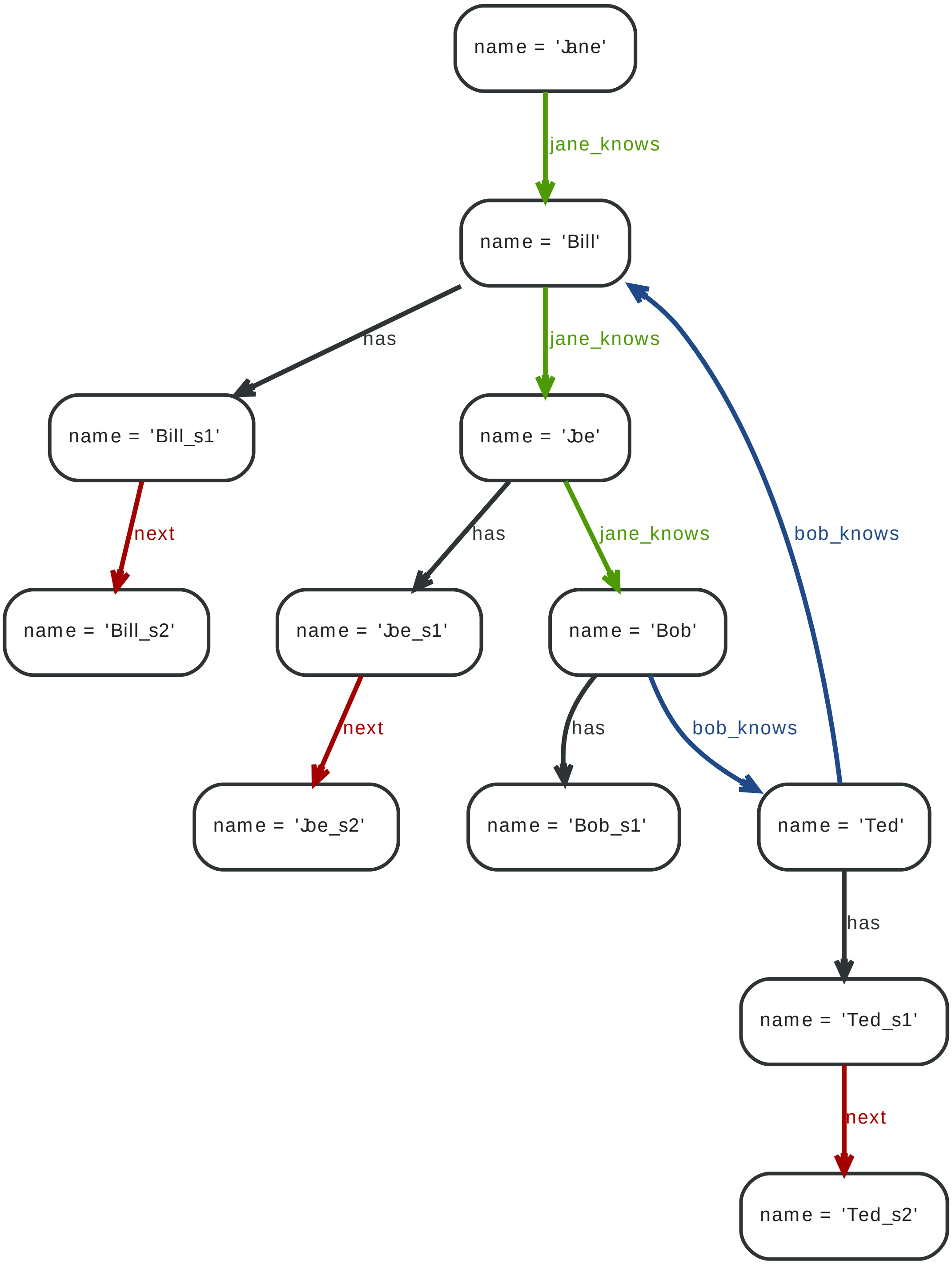
Result

me.name	friend.name	status.name	length(p)
"Jane"	"Bill"	"Bill_s1"	1
"Jane"	"Joe"	"Joe_s1"	2
"Jane"	"Bob"	"Bob_s1"	3

3 rows

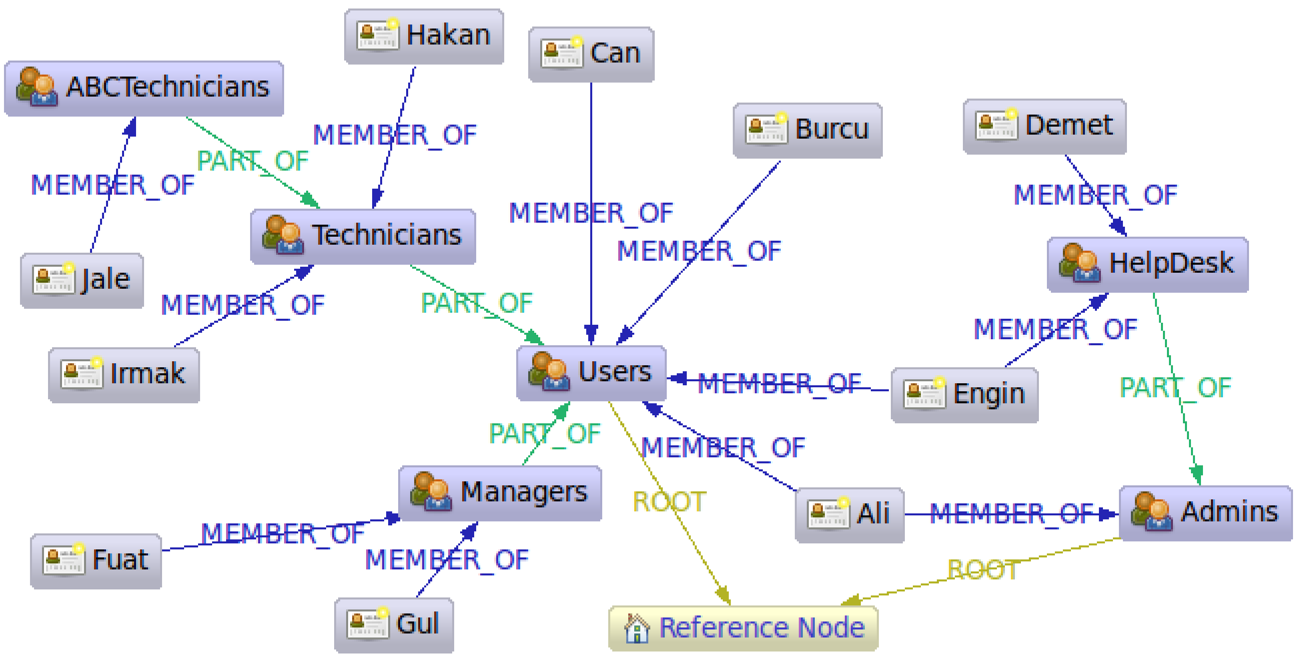
6 http://www.rene-pickhardt.de/graphity-an-efficient-graph-model-for-retrieving-the-top-k-news-feeds-for-users-in-social-networks/

Figure 6.20. Graph



6.16. 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 kind of 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[7] by Kemal Erdogan[8]. The article discusses how to store directed acyclic graphs[9] (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", as they are both admins and users and thus reachable through these group nodes. 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` (green edges) relationships to model the group hierarchy and `MEMBER_OF` (blue 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 the semantics you want.

Lets now have a look at how to retrieve information from the graph. The the queries are done using Cypher, the Java code is using the Neo4j Traversal API (see Section 36.2, "Traversal Framework Java API" [626], which is part of Part VIII, "Advanced Usage" [574]).

7 http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o
8 http://www.codeproject.com/script/Articles/MemberArticles.aspx?amid=274518
9 http://en.wikipedia.org/wiki/Directed_acyclic_graph

Get the admins

In Cypher, we could get the admins like this:

```
MATCH ({ name: 'Admins' })<-[:PART_OF*0..]-(group)<-[:MEMBER_OF]-(user)
RETURN user.name, group.name
```

resulting in:

user.name	group.name
"Ali"	"Admins"
"Demet"	"HelpDesk"
"Engin"	"HelpDesk"

3 rows

And here's the code when using the Java Traversal API:

```
Node admins = getNodeByName( "Admins" );
TraversalDescription traversalDescription = db.traversalDescription()
        .breadthFirst()
        .evaluator( Evaluators.excludeStartPosition() )
        .relationships( RoleRels.PART_OF, Direction.INCOMING )
        .relationships( RoleRels.MEMBER_OF, Direction.INCOMING );
Traverser traverser = traversalDescription.traverse( admins );
```

resulting in the output

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

The result is collected from the traverser using this code:

```
String output = "";
for ( Path path : traverser )
{
    Node node = path.endNode();
    output += "Found: " + node.getProperty( NAME ) + " at depth: "
              + ( path.length() - 1 ) + "\n";
}
```

Get the group memberships of a user

In Cypher:

```
MATCH ({ name: 'Jale' })-[:MEMBER_OF]->()-[:PART_OF*0..]->(group)
RETURN group.name
```

group.name
"ABCTechnicians"
"Technicians"
"Users"

3 rows

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

```
Node jale = getNodeByName( "Jale" );
```

```
traversalDescription = db.traversalDescription()
        .depthFirst()
        .evaluator( Evaluators.excludeStartPosition() )
        .relationships( RoleRels.MEMBER_OF, Direction.OUTGOING )
        .relationships( RoleRels.PART_OF, Direction.OUTGOING );
traverser = traversalDescription.traverse( jale );
```

resulting in:

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

Get all groups

In Cypher:

```
MATCH ({ name: 'Reference_Node' })<-[:ROOT]->()<-[:PART_OF*0..]-(group)
RETURN group.name
```

group.name
"Users"
"Managers"
"Technicians"
"ABCTechnicians"
"Admins"
"HelpDesk"

6 rows

In Java:

```
Node referenceNode = getNodeByName( "Reference_Node") ;
traversalDescription = db.traversalDescription()
        .breadthFirst()
        .evaluator( Evaluators.excludeStartPosition() )
        .relationships( RoleRels.ROOT, Direction.INCOMING )
        .relationships( RoleRels.PART_OF, Direction.INCOMING );
traverser = traversalDescription.traverse( referenceNode );
```

resulting in:

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

Get all members of all groups

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

In Cypher, this looks like:

```
MATCH ({ name: 'Reference_Node' })<-[:ROOT]->(root), p=(root)<-[PART_OF*0..]-()<-[:MEMBER_OF]-(user)
RETURN user.name, min(length(p))
ORDER BY min(length(p)), user.name
```

and results in the following output:

user.name	min(length(p))
"Ali"	1
"Ali"	1
"Burcu"	1
"Can"	1
"Engin"	1
"Demet"	2
"Engin"	2
"Fuat"	2
"Gul"	2
"Hakan"	2
"Irmak"	2
"Jale"	3

12 rows

in Java:

```
traversalDescription = db.traversalDescription()
        .breadthFirst()
        .evaluator(
                Evaluators.includeWhereLastRelationshipTypeIs( RoleRels.MEMBER_OF ) );
traverser = traversalDescription.traverse( referenceNode );
```

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

As seen above, querying even more complex scenarios can be done using comparatively short constructs in Cypher or Java.

Chapter 7. Languages

Please see http://neo4j.com/developer/language-guides/ for the current set of drivers!

There's an included Java example which shows a "low-level" approach to using the Neo4j REST API from Java.

7.1. How to use the REST API from Java

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[1] client components, which are easily downloaded[2] via Maven.

Start the server

Before we can perform any actions on the server, we need to start it as per Section 23.2, "Server Installation" [426]. Next up, we'll check the connection to the server:

```
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 connect to the server, then please have a look at Part V, "Operations" [422].

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.

Sending Cypher

Using the REST API, we can send Cypher queries to the server. This is the main way to use Neo4j. It allows control of the transactional boundaries as needed.

Let's try to use this to list all the nodes in the database which have a `name` property.

```
final String txUri = SERVER_ROOT_URI + "transaction/commit";
WebResource resource = Client.create().resource( txUri );

String payload = "{\"statements\" : [ {\"statement\" : \"" +query + "\"} ]}";
ClientResponse response = resource
        .accept( MediaType.APPLICATION_JSON )
        .type( MediaType.APPLICATION_JSON )
        .entity( payload )
        .post( ClientResponse.class );

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

response.close();
```

For more details, see Section 21.1, "Transactional Cypher HTTP endpoint" [286].

[1] http://jersey.java.net/
[2] https://jersey.java.net/nonav/documentation/1.9/user-guide.html#chapter_deps

Fine-grained REST API calls

For exploratory and special purposes, there is a fine grained REST API, see Chapter 21, *REST API* [285]. The following sections highlight some of the basic operations.

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.

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.

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.

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 `204 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.

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"`.

```
// TraversalDefinition turns into JSON to send to the Server
TraversalDefinition t = new TraversalDefinition();
t.setOrder( TraversalDefinition.DEPTH_FIRST );
t.setUniqueness( TraversalDefinition.NODE );
t.setMaxDepth( 10 );
t.setReturnFilter( TraversalDefinition.ALL );
t.setRelationships( new Relation( "singer", Relation.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"
} ]
```


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.

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.

Appendix: the code

- CreateSimpleGraph.java[3]
- Relation.java[4]
- TraversalDefinition.java[5]

[3] https://github.com/neo4j/neo4j/blob/2.2.1/community/server-examples/src/main/java/org/neo4j/examples/server/CreateSimpleGraph.java
[4] https://github.com/neo4j/neo4j/blob/2.2.1/community/server-examples/src/main/java/org/neo4j/examples/server/Relation.java
[5] https://github.com/neo4j/neo4j/blob/2.2.1/community/server-examples/src/main/java/org/neo4j/examples/server/TraversalDefinition.java

Part III. Cypher Query Language

The Cypher part is the authoritative source for details on the Cypher Query Language. For a short introduction, see Section 8.1, "What is Cypher?" [105]. To take your first steps with Cypher, see Chapter 3, *Introduction to Cypher* [16]. For the terminology used, see Terminology [654].

Chapter 8. Introduction

To get an overview of Cypher, continue reading Section 8.1, “What is Cypher?” [105]. The rest of this chapter deals with the context of Cypher statements, like for example transaction management and how to use parameters. For the Cypher language reference itself see other chapters at Part III, “Cypher Query Language” [101]. To take your first steps with Cypher, see Chapter 3, *Introduction to Cypher* [16]. For the terminology used, see Terminology [654].

8.1. What is Cypher?

Introduction

Cypher is a declarative graph query language that allows for expressive and efficient querying and updating of the graph store. Cypher is a relatively simple but still very powerful language. Very complicated database queries can easily be expressed through Cypher. This allows you to focus on your domain instead of getting lost in database access.

Cypher is designed to be a humane query language, suitable for both developers and (importantly, we think) operations professionals. Our guiding goal is to make the simple things easy, and the complex things possible. Its constructs are based on English prose and neat iconography which helps to make queries more self-explanatory. We have tried to optimize the language for reading and not for writing.

Being a declarative language, Cypher focuses on the clarity of expressing *what* to retrieve from a graph, not on *how* to retrieve it. This is in contrast to imperative languages like Java, scripting languages like Gremlin[1], and the JRuby Neo4j bindings[2]. This approach makes query optimization an implementation detail instead of burdening the user with it and requiring her to update all traversals just because the physical database structure has changed (new indexes etc.).

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[3]. Pattern matching borrows expression approaches from SPARQL[4]. Some of the collection semantics have been borrowed from languages such as Haskell and Python.

Structure

Cypher borrows its structure from SQL — queries are built up using various clauses.

Clauses are chained together, and the they feed intermediate result sets between each other. For example, the matching identifiers from one `MATCH` clause will be the context that the next clause exists in.

The query language is comprised of several distinct clauses. You can read more details about them later in the manual.

Here are a few clauses used to read from the graph:

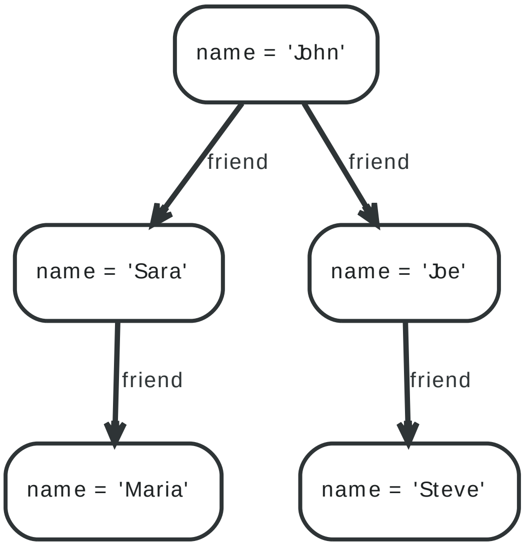
- `MATCH`: The graph pattern to match. This is the most common way to get data from the graph.
- `WHERE`: Not a clause in it's own right, but rather part of `MATCH`, `OPTIONAL MATCH` and `WITH`. Adds constraints to a pattern, or filters the intermediate result passing through `WITH`.
- `RETURN`: What to return.

Let's see `MATCH` and `RETURN` in action.

Imagine an example graph like the following one:

1 http://gremlin.tinkerpop.com
2 https://github.com/neo4jrb/neo4j/
3 http://en.wikipedia.org/wiki/SQL
4 http://en.wikipedia.org/wiki/SPARQL

Figure 8.1. Example Graph



For example, here is a query which finds a user called John and John's friends (though not his direct friends) before returning both John and any friends-of-friends that are found.

```
MATCH (john {name: 'John'})-[:friend]->()-[:friend]->(fof)
RETURN john.name, fof.name
```

Resulting in:

john.name	fof.name
"John"	"Maria"
"John"	"Steve"

2 rows

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

We take a list of user names and find all nodes with names from this list, match their friends and return only those followed users who have a `name` property starting with `S`.

```
MATCH (user)-[:friend]->(follower)
WHERE user.name IN ['Joe', 'John', 'Sara', 'Maria', 'Steve'] AND follower.name =~ 'S.*'
RETURN user.name, follower.name
```

Resulting in:

user.name	follower.name
"John"	"Sara"
"Joe"	"Steve"

2 rows

And here are examples of clauses that are used to update the graph:

- `CREATE` (and `DELETE`): Create (and delete) nodes and relationships.
- `SET` (and `REMOVE`): Set values to properties and add labels on nodes using `SET` and use `REMOVE` to remove them.
- `MERGE`: Match existing or create new nodes and patterns. This is especially useful together with uniqueness constraints.

For more Cypher examples, see Chapter 5, *Basic Data Modeling Examples* [45] as well as the rest of the Cypher part with details on the language. To use Cypher from Java, see Section 35.14, "Execute

Cypher Queries from Java” [620]. To take your first steps with Cypher, see Chapter 3, *Introduction to Cypher* [16].

8.2. Updating the graph

Cypher can be used for both querying and updating your graph.

The Structure of Updating Queries

Quick info

- A Cypher query part can't both match and update the graph at the same time.
- Every part can either read and match on the graph, or make updates on it.

If you read from the graph, and then update the graph, your query implicitly has two parts — the reading is the first part, and the writing is the second. If your query is read-only, Cypher will be lazy, and not actually match the pattern until you ask for the results. In an updating query, the semantics are that *all* the reading will be done before any writing actually happens. First reading, and then writing, is the only pattern where the query parts are implicit — any other order and you have to be explicit about your query parts. The parts are separated using the `WITH` statement. `WITH` is like the event horizon — it's a barrier between a plan and the finished execution of that plan.

When you want to filter using aggregated data, you have to chain together two reading query parts — the first one does the aggregating, and the second query filters on the results coming from the first one.

```
MATCH (n {name: 'John'})-[:FRIEND]-friend
WITH n, count(friend) as friendsCount
WHERE friendsCount > 3
RETURN n, friendsCount
```

Using `WITH`, you specify how you want the aggregation to happen, and that the aggregation has to be finished before Cypher can start filtering.

You can chain together as many query parts as you have JVM heap for.

Returning data

Any query can return data. If your query only reads, it has to return data — it serves no purpose if it doesn't, and it is not a valid Cypher query. Queries that update the graph don't have to return anything, but they can.

After all the parts of the query comes one final `RETURN` clause. `RETURN` is not part of any query part — it is a period symbol at the end of a query. The `RETURN` clause has three sub-clauses that come with it `SKIP`/`LIMIT` and `ORDER BY`.

If you return graph elements from a query that has just deleted them — beware, you are holding a pointer that is no longer valid. Operations on that node might fail mysteriously and unpredictably.

8.3. Transactions

Any query that updates the graph will run in a transaction. An updating query will always either fully succeed, or not succeed at all.

Cypher will either create a new transaction or run inside an existing one:

- If no transaction exists in the running context Cypher will create one and commit it once the query finishes.
- In case there already exists a transaction in the running context, the query will run inside it, and nothing will be persisted to disk until that transaction is successfully committed.

This can be used to have multiple queries be committed as a single transaction:

1. Open a transaction,
2. run multiple updating Cypher queries,
3. and commit all of them in one go.

Note that a query will hold the changes in memory until the whole query has finished executing. A large query will consequently need a JVM with lots of heap space.

For using transactions over the REST API, see Section 21.1, “Transactional Cypher HTTP endpoint” [286].

When writing server extensions or using Neo4j embedded, remember that all iterators returned from an execution result should be either fully exhausted or closed to ensure that the resources bound to them will be properly released. Resources include transactions started by the query, so failing to do so may, for example, lead to deadlocks or other weird behavior.

8.4. Uniqueness

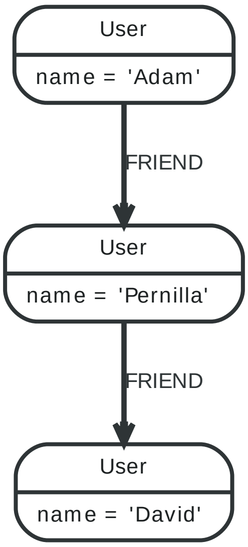
While pattern matching, Neo4j makes sure to not include matches where the same graph relationship is found multiple times in a single pattern. In most use cases, this is a sensible thing to do.

Example: looking for a user's friends of friends should not return said user.

Let's create a few nodes and relationships:

```
CREATE (adam:User { name: 'Adam' }),(pernilla:User { name: 'Pernilla' }),(david:User { name: 'David'
  }),
  (adam)-[:FRIEND]->(pernilla),(pernilla)-[:FRIEND]->(david)
```

Which gives us the following graph:



Now let's look for friends of friends of Adam:

```
MATCH (user:User { name: 'Adam' })-[r1:FRIEND]-()-[r2:FRIEND]-(friend_of_a_friend)
RETURN friend_of_a_friend.name AS fofName
```

fofName
"David"

1 row

In this query, Cypher makes sure to not return matches where the pattern relationships `r1` and `r2` point to the same graph relationship.

This is however not always desired. If the query should return the user, it is possible to spread the matching over multiple `MATCH` clauses, like so:

```
MATCH (user:User { name: 'Adam' })-[r1:FRIEND]-(friend)
MATCH (friend)-[r2:FRIEND]-(friend_of_a_friend)
RETURN friend_of_a_friend.name AS fofName
```

fofName
"David"
"Adam"

2 rows

Note that while the following query looks similar to the previous one, it is actually equivalent to the one before.

```
MATCH (user:User { name: 'Adam' })-[r1:FRIEND]-(friend),(friend)-[r2:FRIEND]-(friend_of_a_friend)
RETURN friend_of_a_friend.name AS fofName
```

Here, the `MATCH` clause has a single pattern with two paths, while the previous query has two distinct patterns.

fofName
"David"

1 row

8.5. Parameters

Cypher supports querying with parameters. This means developers don't have to resort to string building to create a query. In addition to that, it also makes caching of execution plans much easier for Cypher.

Parameters can be used for literals and expressions in the `WHERE` clause, for the index value in the `START` clause, index queries, and finally for node/relationship ids. Parameters can not be used as for property names, relationship types and labels, since these patterns are part of the query structure that is compiled into a query plan.

Accepted names for parameters are letters and numbers, and any combination of these.

For details on using parameters via the Neo4j REST API, see Section 21.1, "Transactional Cypher HTTP endpoint" [286]. For details on parameters when using the Neo4j embedded Java API, see Section 35.15, "Query Parameters" [622].

Below follows a comprehensive set of examples of parameter usage. The parameters are given as JSON here. Exactly how to submit them depends on the driver in use.

String literal

Parameters

```
{
  "name" : "Johan"
}
```

Query

```
MATCH (n)
WHERE n.name = { name }
RETURN n
```

You can use parameters in this syntax as well:

Parameters

```
{
  "name" : "Johan"
}
```

Query

```
MATCH (n { name: { name }})
RETURN n
```

Regular expression

Parameters

```
{
  "regex" : ".*h.*"
}
```

Query

```
MATCH (n)
WHERE n.name =~ { regex }
RETURN n.name
```

Create node with properties

Parameters

```
{
  "props" : {
    "position" : "Developer",
    "name" : "Andres"
  }
}
```

Query

```
CREATE ({ props })
```

Create multiple nodes with properties

Parameters

```
{
  "props" : [ {
    "position" : "Developer",
    "awesome" : true,
    "name" : "Andres"
  }, {
    "position" : "Developer",
    "name" : "Michael",
    "children" : 3
  } ]
}
```

Query

```
CREATE (n:Person { props })
RETURN n
```

Setting all properties on node

Note that this will replace all the current properties.

Parameters

```
{
  "props" : {
    "position" : "Developer",
    "name" : "Andres"
  }
}
```

Query

```
MATCH (n)
WHERE n.name='Michaela'
SET n = { props }
```

SKIP and LIMIT

Parameters

```
{
  "s" : 1,
  "l" : 1
}
```

Query

```
MATCH (n)
RETURN n.name
```

```
SKIP { s }
LIMIT { l }
```

Node id

Parameters

```
{
  "id" : 0
}
```

Query

```
MATCH n
WHERE id(n)= { id }
RETURN n.name
```

Multiple node ids

Parameters

```
{
  "ids" : [ 0, 1, 2 ]
}
```

Query

```
MATCH n
WHERE id(n) IN { ids }
RETURN n.name
```

Index value (legacy indexes)

Parameters

```
{
  "value" : "Michaela"
}
```

Query

```
START n=node:people(name = { value })
RETURN n
```

Index query (legacy indexes)

Parameters

```
{
  "query" : "name:Andreas"
}
```

Query

```
START n=node:people({ query })
RETURN n
```

8.6. Compatibility

Cypher is still changing rather rapidly. Parts of the changes are internal — we add new pattern matchers, aggregators and optimizations or write new query planners, which hopefully makes your queries run faster.

Other changes are directly visible to our users — the syntax is still changing. New concepts are being added and old ones changed to fit into new possibilities. To guard you from having to keep up with our syntax changes, Cypher allows you to use an older parser, but still gain speed from new optimizations.

There are two ways you can select which parser to use. You can configure your database with the configuration parameter `cypher_parser_version`, and enter which parser you'd like to use (see the section called “Supported Language Versions” [115]). Any Cypher query that doesn't explicitly say anything else, will get the parser you have configured, or the latest parser if none is configured.

The other way is on a query by query basis. By simply putting `"CYPHER 2.1"` at the beginning, that particular query will be parsed with the 2.1 version of the parser. Below is an example using the `START` clause to access a legacy index:

```
CYPHER 2.1
START n=node:nodes(name = "A")
RETURN n
```

Accessing entities by id via `START`

In versions of Cypher prior to 2.2 it was also possible to access specific nodes or relationships using the `START` clause. In this case you could use a syntax like the following:

```
CYPHER 2.1
START n=node(42)
RETURN n
```

Note
The use of the `START` clause to find nodes by ID was deprecated from Cypher 2.0 onwards and is now entirely disabled in Cypher 2.2 and up. You should instead make use of the `MATCH` clause for starting points. See Section 11.1, “Match” [152] for more information on the correct syntax for this. The `START` clause should only be used when accessing legacy indexes (see Chapter 37, *Legacy Indexing* [632]).

Supported Language Versions

Neo4j 2.2 supports the following versions of the Cypher language:

- Neo4j Cypher 2.2
- Neo4j Cypher 2.1
- Neo4j Cypher 2.0
- Neo4j Cypher 1.9

Tip
Each release of Neo4j supports a limited number of old Cypher Language Versions. When you upgrade to a new release of Neo4j, please make sure that it supports the Cypher language version you need. If not, you may need to modify your queries to work with a newer Cypher language version.

Note
Support for Cypher version 2.0 and Cypher version 2.1 will be removed in Neo4j 2.3

Chapter 9. Syntax

The nitty-gritty details of Cypher syntax.

9.1. Values

Cypher queries the graph by looking at nodes, relationships, properties and query parameters. All values that are handled by Cypher have a distinct type. The supported types of values are:

- Numeric values,
- String values,
- Boolean values,
- Nodes,
- Relationships,
- Paths,
- Maps from Strings to other values,
- Collections of any other type of value.

Most types of values can be constructed in a query using literal expressions (see Section 9.2, “Expressions” [118]). Special care must be taken when using `NULL`, as `NULL` is a value of every type (see Section 9.8, “Working with NULL” [132]). Nodes, relationships, and paths are returned as a result of pattern matching.

Note that labels are not values but are a form of pattern syntax.

9.2. Expressions

Expressions in general

An expression in Cypher can be:

- A decimal (integer or double) literal: `13`, `-40000`, `3.14`, `6.022E23`.
- A hexadecimal integer literal (starting with `0x`): `0x13zf`, `0xFC3A9`, `-0x66eff`.
- An octal integer literal (starting with `0`): `01372`, `01278`, `-05671`.
- A string literal: `"Hello"`, `'World'`.
- A boolean literal: `true`, `false`, `TRUE`, `FALSE`.
- An identifier: `n`, `x`, `rel`, `myFancyIdentifier`, ``` `A name with weird stuff in it[]!` ```.
- A property: `n.prop`, `x.prop`, `rel.thisProperty`, ``` myFancyIdentifier.`(weird property name)` ```.
- A parameter: `{param}`, `{0}`
- A collection of expressions: `["a", "b"]`, `[1,2,3]`, `["a", 2, n.property, {param}]`, `[ ]`.
- A function call: `length(p)`, `nodes(p)`.
- An aggregate function: `avg(x.prop)`, `count(*)`.
- A path-pattern: `(a)-->()<--(b)`.
- An operator application: `1 + 2` and `3 < 4`.
- A predicate expression is an expression that returns true or false: `a.prop = "Hello"`, `length(p) > 10`, `has(a.name)`.
- A `CASE` expression.

Note on string literals

String literals can contain these escape sequences.

Escape sequence	Character
`\t`	Tab
`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\f`	Form feed
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash
`\uxxxx`	Unicode UTF-16 code point (4 hex digits must follow the `\u`)
`\Uxxxxxxxx`	Unicode UTF-32 code point (8 hex digits must follow the `\U`)

Case Expressions

Cypher supports `CASE` expressions, which is a generic conditional expression, similar to if/else statements in other languages. Two variants of `CASE` exist — the simple form and the generic form.

Simple CASE

The expression is calculated, and compared in order with the `WHEN` clauses until a match is found. If no match is found the expression in the `ELSE` clause is used, or `null`, if no `ELSE` case exists.

Syntax:

```
CASE test
WHEN value THEN result
[WHEN ...]
[ELSE default]
END
```

Arguments:

- *test:* A valid expression.
- *value:* An expression whose result will be compared to the `test` expression.
- *result:* This is the result expression used if the value expression matches the `test` expression.
- *default:* The expression to use if no match is found.

Query

```
MATCH n
RETURN
CASE n.eyes
WHEN 'blue'
THEN 1
WHEN 'brown'
THEN 2
ELSE 3 END AS result
```

Result

result
2
1
2
1
3

5 rows

Generic CASE

The predicates are evaluated in order until a true value is found, and the result value is used. If no match is found the expression in the `ELSE` clause is used, or `null`, if no `ELSE` case exists.

Syntax:

```
CASE
WHEN predicate THEN result
[WHEN ...]
[ELSE default]
END
```

Arguments:

- *predicate:* A predicate that is tested to find a valid alternative.
- *result:* This is the result expression used if the predicate matches.
- *default:* The expression to use if no match is found.

Query

```
MATCH n
RETURN
```

```
CASE
WHEN n.eyes = 'blue'
THEN 1
WHEN n.age < 40
THEN 2
ELSE 3 END AS result
```

Result

result
3
1
2
1
3

5 rows

9.3. Identifiers

When you reference parts of a pattern or a query, you do so by naming them. The names you give the different parts are called identifiers.

In this example:

```
MATCH (n)-->(b) RETURN b
```

The identifiers are `n` and `b`.

Identifier names are case sensitive, and can contain underscores and alphanumeric characters (a-z, 0-9), but must always start with a letter. If other characters are needed, you can quote the identifier using backquote (`` ` ``) signs.

The same rules apply to property names.

9.4. Operators

Mathematical operators

The mathematical operators are +, -, *, / and %, ^.

Comparison operators

The comparison operators are =, <>, <, >, <=, >=, IS NULL, and IS NOT NULL. See the section called “Equality and Comparison of Values” [122] on how they behave.

Boolean operators

The boolean operators are AND, OR, XOR, NOT.

String operators

Strings can be concatenated using the + operator.

Collection operators

Collections can be concatenated using the + operator. To check if an element exists in a collection, you can use the IN operator.

Property operators

Note
Since version 2.0, the previously existing property operators ? and ! have been removed. This syntax is no longer supported. Missing properties are now returned as NULL. Please use (NOT(has(<ident>.prop)) OR <ident>.prop=<value>) if you really need the old behavior of the ? operator. — Also, the use of ? for optional relationships has been removed in favor of the newly introduced OPTIONAL MATCH clause.

Equality and Comparison of Values

Equality

Cypher supports comparing values (see Section 9.1, “Values” [117]) by equality using the = and <> operators.

Values of the same type are only equal if they are the same identical value (e.g. 3 = 3 and "x" <> "xy").

Maps are only equal if they map exactly the same keys to equal values and collections are only equal if they contain the same sequence of equal values (e.g. [3, 4] = [1+2, 8/2]).

Values of different types are considered as equal according to the following rules:

- Paths are treated as collections of alternating nodes and relationships and are equal to all collections that contain that very same sequence of nodes and relationships.
- Testing any value against NULL with both the = and the <> operators always is NULL. This includes NULL = NULL and NULL <> NULL. The only way to reliably test if a value v is NULL is by using the special v IS NULL, or v IS NOT NULL equality operators.

All other combinations of types of values cannot be compared with each other. Especially, nodes, relationships, and literal maps are incomparable with each other.

It is an error to compare values that cannot be compared.

Ordering and Comparison of Values

The comparison operators <= (for ascending) and >= (for descending) are used to compare values for ordering. The following points give some details on how the comparison is performed.

- Numerical values are compared for ordering using numerical order (e.g. `3 < 4` is true).
- String values are compared for ordering using lexicographic order (e.g. `"x" < "xy"`).
- Boolean values are compared for ordering such that `false < true`.
- Comparing for ordering when one argument is `NULL` is `NULL` (e.g. `NULL < 3` is `NULL`).
- It is an error to compare other types of values with each other for ordering.

For other comparison operators, see the section called “Comparison operators” [122].

9.5. Comments

To add comments to your queries, use double slash. Examples:

```
MATCH (n) RETURN n //This is an end of line comment
```

```
MATCH (n)
//This is a whole line comment
RETURN n
```

```
MATCH (n) WHERE n.property = "//This is NOT a comment" RETURN n
```

9.6. Patterns

Patterns and pattern-matching are at the very heart of Cypher, so being effective with Cypher requires a good understanding of patterns.

Using patterns, you describe the shape of the data you're looking for. For example, in the `MATCH` clause you describe the shape with a pattern, and Cypher will figure out how to get that data for you.

The pattern describes the data using a form that is very similar to how one typically draws the shape of property graph data on a whiteboard: usually as circles (representing nodes) and arrows between them to represent relationships.

Patterns appear in multiple places in Cypher: in `MATCH`, `CREATE` and `MERGE` clauses, and in pattern expressions. Each of these is described in more details in:

- Section 11.1, “Match” [152]
- Section 11.2, “Optional Match” [161]
- Section 12.1, “Create” [181]
- Section 12.2, “Merge” [185]
- the section called “Using patterns in WHERE” [165]

Patterns for nodes

The very simplest “shape” that can be described in a pattern is a node. A node is described using a pair of parentheses, and is typically given a name. For example:

```
(a)
```

This simple pattern describes a single node, and names that node using the identifier `a`.

Note that the parentheses may be omitted, but only when there are no labels or properties specified for the node pattern.

Patterns for related nodes

More interesting is patterns that describe multiple nodes and relationships between them. Cypher patterns describe relationships by employing an arrow between two nodes. For example:

```
(a)-->(b)
```

This pattern describes a very simple data shape: two nodes, and a single relationship from one to the other. In this example, the two nodes are both named as `a` and `b` respectively, and the relationship is “directed”: it goes from `a` to `b`.

This way of describing nodes and relationships can be extended to cover an arbitrary number of nodes and the relationships between them, for example:

```
(a)-->(b)<--(c)
```

Such a series of connected nodes and relationships is called a "path".

Note that the naming of the nodes in these patterns is only necessary should one need to refer to the same node again, either later in the pattern or elsewhere in the Cypher query. If this is not necessary then the name may be omitted, like so:

```
(a)-->()<--(c)
```

Labels

In addition to simply describing the shape of a node in the pattern, one can also describe attributes. The most simple attribute that can be described in the pattern is a label that the node must have. For example:

```
(a:User)-->(b)
```

One can also describe a node that has multiple labels:

```
(a:User:Admin)-->(b)
```

Specifying properties

Nodes and relationships are the fundamental structures in a graph. Neo4j uses properties on both of these to allow for far richer models.

Properties can be expressed in patterns using a map-construct: curly brackets surrounding a number of key-expression pairs, separated by commas. E.g. a node with two properties on it would look like:

```
(a { name: "Andres", sport: "Brazilian Ju-Jitsu" })
```

A relationship with expectations on it would could look like:

```
(a)-[{blocked: false}]->(b)
```

When properties appear in patterns, they add an additional constraint to the shape of the data. In the case of a `CREATE` clause, the properties will be set in the newly created nodes and relationships. In the case of a `MERGE` clause, the properties will be used as additional constraints on the shape any existing data must have (the specified properties must exactly match any existing data in the graph). If no matching data is found, then `MERGE` behaves like `CREATE` and the properties will be set in the newly created nodes and relationships.

Note that patterns supplied to `CREATE` may use a single parameter to specify properties, e.g: `CREATE (node {paramName})`. This is not possible with patterns used in other clauses, as Cypher needs to know the property names at the time the query is compiled, so that matching can be done effectively.

Describing relationships

The simplest way to describe a relationship is by using the arrow between two nodes, as in the previous examples. Using this technique, you can describe that the relationship should exist and the directionality of it. If you don't care about the direction of the relationship, the arrow head can be omitted, like so:

```
(a)--(b)
```

As with nodes, relationships may also be given names. In this case, a pair of square brackets is used to break up the arrow and the identifier is placed between. For example:

```
(a)-[r]->(b)
```

Much like labels on nodes, relationships can have types. To describe a relationship with a specific type, you can specify this like so:

```
(a)-[r:REL_TYPE]->(b)
```

Unlike labels, relationships can only have one type. But if we'd like to describe some data such that the relationship could have any one of a set of types, then they can all be listed in the pattern, separating them with the pipe symbol `|` like this:

```
(a)-[r:TYPE1|TYPE2]->(b)
```

Note that this form of pattern can only be used to describe existing data (ie. when using a pattern with `MATCH` or as an expression). It will not work with `CREATE` or `MERGE`, since it's not possible to create a relationship with multiple types.

As with nodes, the name of the relationship can always be omitted, in this case like so:

```
(a)-[:REL_TYPE]->(b)
```

Variable length

Rather than describing a long path using a sequence of many node and relationship descriptions in a pattern, many relationships (and the intermediate nodes) can be described by specifying a length in the relationship description of a pattern. For example:

```
(a)-[*2]->(b)
```

This describes a graph of three nodes and two relationship, all in one path (a path of length 2). This is equivalent to:

```
(a)-->()-->(b)
```

A range of lengths can also be specified: such relationship patterns are called “variable length relationships”. For example:

```
(a)-[*3..5]->(b)
```

This is a minimum length of 3, and a maximum of 5. It describes a graph of either 4 nodes and 3 relationships, 5 nodes and 4 relationships or 6 nodes and 5 relationships, all connected together in a single path.

Either bound can be omitted. For example, to describe paths of length 3 or more, use:

```
(a)-[*3..]->(b)
```

And to describe paths of length 5 or less, use:

```
(a)-[*..5]->(b)
```

Both bounds can be omitted, allowing paths of any length to be described:

```
(a)-[*]->(b)
```

As a simple example, let’s take the query below:

Query

```
MATCH (me)-[:KNOWS*1..2]-(remote_friend)
WHERE me.name = "Filipa"
RETURN remote_friend.name
```

Result

remote_friend.name
"Dilshad"
"Anders"

2 rows

This query finds data in the graph which a shape that fits the pattern: specifically a node (with the name property `Filipa`) and then the `KNOWS` related nodes, one or two steps out. This is a typical example of finding first and second degree friends.

Note that variable length relationships can not be used with `CREATE` and `MERGE`.

Assigning to path identifiers

As described above, a series of connected nodes and relationships is called a "path". Cypher allows paths to be named using an identifer, like so:

```
p = (a)-[*3..5]->(b)
```

You can do this in `MATCH`, `CREATE` and `MERGE`, but not when using patterns as expressions.

9.7. Collections

Cypher has good support for collections.

Collections in general

A literal collection is created by using brackets and separating the elements in the collection with commas.

Query

```
RETURN [0,1,2,3,4,5,6,7,8,9] AS collection
```

Result

collection
[0,1,2,3,4,5,6,7,8,9]

1 row

In our examples, we'll use the range function. It gives you a collection containing all numbers between given start and end numbers. Range is inclusive in both ends.

To access individual elements in the collection, we use the square brackets again. This will extract from the start index and up to but not including the end index.

Query

```
RETURN range(0,10)[3]
```

Result

range(0,10)[3]
3

1 row

You can also use negative numbers, to start from the end of the collection instead.

Query

```
RETURN range(0,10)[-3]
```

Result

range(0,10)[-3]
8

1 row

Finally, you can use ranges inside the brackets to return ranges of the collection.

Query

```
RETURN range(0,10)[0..3]
```

Result

range(0,10)[0..3]
[0,1,2]

1 row

Query

```
RETURN range(0,10)[0..-5]
```

Result

range(0,10)[0..-5]
[0,1,2,3,4,5]

1 row

Query

```
RETURN range(0,10)[-5..]
```

Result

range(0,10)[-5..]
[6,7,8,9,10]

1 row

Query

```
RETURN range(0,10)[..4]
```

Result

range(0,10)[..4]
[0,1,2,3]

1 row

> **Note**
> Out-of-bound slices are simply truncated, but out-of-bound single elements return `NULL`.

Query

```
RETURN range(0,10)[15]
```

Result

range(0,10)[15]
<null>

1 row

Query

```
RETURN range(0,10)[5..15]
```

Result

range(0,10)[5..15]
[5,6,7,8,9,10]

1 row

You can get the length of a collection like this:

Query

```
RETURN length(range(0,10)[0..3])
```

Result

length(range(0,10)[0..3])
3

1 row

List comprehension

List comprehension is a syntactic construct available in Cypher for creating a collection based on existing collections. It follows the form of the mathematical set-builder notation (set comprehension) instead of the use of map and filter functions.

Query

```
RETURN [x IN range(0,10) WHERE x % 2 = 0 | x^3] AS result
```

Result

result
[0.0,8.0,64.0,216.0,512.0,1000.0]

1 row

Either the `WHERE` part, or the expression, can be omitted, if you only want to filter or map respectively.

Query

```
RETURN [x IN range(0,10) WHERE x % 2 = 0] AS result
```

Result

result
[0,2,4,6,8,10]

1 row

Query

```
RETURN [x IN range(0,10)| x^3] AS result
```

Result

result
[0.0,1.0,8.0,27.0,64.0,125.0,216.0,343.0,512.0,729.0,1000.0]

1 row

Literal maps

From Cypher, you can also construct maps. Through REST you will get JSON objects; in Java they will be `java.util.Map<String,Object>`.

Query

```
RETURN { key : "Value", collectionKey: [{ inner: "Map1" }, { inner: "Map2" }]}
```

Result

{ key : "Value", collectionKey: [{ inner: "Map1" }, { inner: "Map2" }] }
{key -> "Value", collectionKey -> [{inner -> "Map1"},{inner -> "Map2"}]}

1 row

9.8. Working with NULL

Introduction to NULL in Cypher

In Cypher, `NULL` is used to represent missing or undefined values. Conceptually, `NULL` means "a missing unknown value" and it is treated somewhat differently from other values. For example getting a property from a node that does not have said property produces `NULL`. Most expressions that take `NULL` as input will produce `NULL`. This includes boolean expressions that are used as predicates in the `WHERE` clause. In this case, anything that is not `TRUE` is interpreted as being false.

`NULL` is not equal to `NULL`. Not knowing two values does not imply that they are the same value. So the expression `NULL = NULL` yields `NULL` and not `TRUE`.

Logical operations with NULL

The logical operators (`AND`, `OR`, `XOR`, `IN`, `NOT`) treat `NULL` as the "unknown" value of three-valued logic. Here is the truth table for `AND`, `OR` and `XOR`.

a	**b**	**a** `AND` **b**	**a** `OR` **b**	**a** `XOR` **b**
`FALSE`	`FALSE`	`FALSE`	`FALSE`	`FALSE`
`FALSE`	`NULL`	`FALSE`	`NULL`	`NULL`
`FALSE`	`TRUE`	`FALSE`	`TRUE`	`TRUE`
`TRUE`	`FALSE`	`FALSE`	`TRUE`	`TRUE`
`TRUE`	`NULL`	`NULL`	`TRUE`	`NULL`
`TRUE`	`TRUE`	`TRUE`	`TRUE`	`FALSE`
`NULL`	`FALSE`	`FALSE`	`NULL`	`NULL`
`NULL`	`NULL`	`NULL`	`NULL`	`NULL`
`NULL`	`TRUE`	`NULL`	`TRUE`	`NULL`

The IN operator and NULL

The `IN` operator follows similar logic. If Cypher knows that something exists in a collection, the result will be `TRUE`. Any collection that contains a `NULL` and doesn't have a matching element will return `NULL`. Otherwise, the result will be false. Here is a table with examples:

Expression	**Result**
2 IN [1, 2, 3]	`TRUE`
2 IN [1, `NULL`, 3]	`NULL`
2 IN [1, 2, `NULL`]	`TRUE`
2 IN [1]	`FALSE`
2 IN []	`FALSE`
`NULL` IN [1,2,3]	`NULL`
`NULL` IN [1,`NULL`,3]	`NULL`
`NULL` IN []	`FALSE`

Using `ALL`, `ANY`, `NONE`, and `SINGLE` follows a similar rule. If the result can be calculated definitely, `TRUE` or `FALSE` is returned. Otherwise `NULL` is produced.

Expressions that return NULL

- Getting a missing element from a collection: `[][0]`, `head([])`

- Trying to access a property that does not exist on a node or relationship: `n.missingProperty`
- Comparisons when either side is `NULL`: `1 < NULL`
- Arithmetic expressions containing `NULL`: `1 + NULL`
- Function calls where any arguments are `NULL`: `sin(NULL)`

Chapter 10. General Clauses

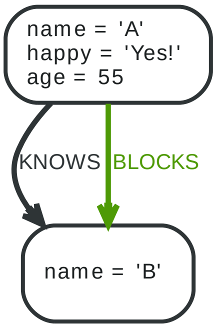
10.1. Return

The RETURN clause defines what to include in the query result set.

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.

Tip
If what you actually want is the value of a property, make sure to not return the full node/ relationship. This will improve performance.

Figure 10.1. Graph



Return nodes

To return a node, list it in the RETURN statement.

Query

```
MATCH (n { name: "B" })
RETURN n
```

The example will return the node.

Result

n
Node[1]{name:"B"}
1 row

Return relationships

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

Query

```
MATCH (n { name: "A" })-[r:KNOWS]->(c)
RETURN r
```

The relationship is returned by the example.

Result

r
:KNOWS[0]{}
1 row

Return property

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

Query

```
MATCH (n { name: "A" })
RETURN n.name
```

The value of the property `name` gets returned.

Result

n.name
"A"

1 row

Return all elements

When you want to return all nodes, relationships and paths found in a query, you can use the `*` symbol.

Query

```
MATCH p=(a { name: "A" })-[r]->(b)
RETURN *
```

This returns the two nodes, the relationship and the path used in the query.

Result

a	b	p	r
Node[0]{name:"A", happy:"Yes!",age:55}	Node[1]{name:"B"}	[Node[0]{name:"A", happy:"Yes!", age:55},:KNOWS[0]{}, Node[1]{name:"B"}]	:KNOWS[0]{}
Node[0]{name:"A", happy:"Yes!",age:55}	Node[1]{name:"B"}	[Node[0]{name:"A", happy:"Yes!", age:55},:BLOCKS[1]{}, Node[1]{name:"B"}]	:BLOCKS[1]{}

2 rows

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

```
MATCH (`This isn't a common identifier`)
WHERE `This isn't a common identifier`.name='A'
RETURN `This isn't a common identifier`.happy
```

The node with name "A" is returned

Result

`This isn't a common identifier`.happy
"Yes!"

1 row

Column alias

If the name of the column should be different from the expression used, you can rename it by using `AS` <new name>.

Query

```
MATCH (a { name: "A" })
RETURN a.age AS SomethingTotallyDifferent
```

Returns the age property of a node, but renames the column.

Result

SomethingTotallyDifferent
55

1 row

Optional properties

If a property might or might not be there, you can still select it as usual. It will be treated as `NULL` if it is missing

Query

```
MATCH (n)
RETURN n.age
```

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

Result

n.age
55
<null>

2 rows

Other expressions

Any expression can be used as a return item — literals, predicates, properties, functions, and everything else.

Query

```
MATCH (a { name: "A" })
RETURN a.age > 30, "I'm a literal",(a)-->()
```

Returns a predicate, a literal and function call with a pattern expression parameter.

Result

a.age > 30	"I'm a literal"	(a)-->()
true	"I'm a literal"	[[Node[0]{name:"A",happy:"Yes!",age:55},:KNOWS[0]{},Node[1]{name:"B"}],[Node[0]{name:"A",happy:"Yes!",age:55},:BLOCKS[1]{},Node[1]{name:"B"}]]

1 row

Unique results

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

Query

```
MATCH (a { name: "A" })-->(b)
RETURN DISTINCT b
```

The node named B is returned by the query, but only once.

Result

b
`Node[1]{name:"B"}`

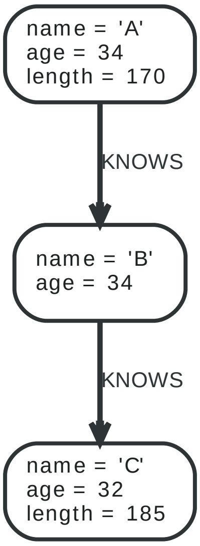
1 row

10.2. Order by

The `ORDER BY` clause specifies that the output should be sorted.

Note that you can not sort on nodes or relationships, just on properties on these. `ORDER BY` relies on comparisons to sort the output, see the section called “Ordering and Comparison of Values” [122].

Figure 10.2. Graph



Order nodes by property

`ORDER BY` is used to sort the output.

Query

```
MATCH (n)
RETURN n
ORDER BY n.name
```

The nodes are returned, sorted by their name.

Result

n
Node[0]{name:"A",age:34,length:170}
Node[1]{name:"B",age:34}
Node[2]{name:"C",age:32,length:185}

3 rows

Order nodes by multiple properties

You can order by multiple properties by stating each identifier in the `ORDER BY` clause. Cypher will sort the result by the first identifier listed, and for equals values, go to the next property in the `ORDER BY` clause, and so on.

Query

```
MATCH (n)
RETURN n
ORDER BY n.age, n.name
```

This returns the nodes, sorted first by their age, and then by their name.

Result

n
Node[2]{name:"C",age:32,length:185}
Node[0]{name:"A",age:34,length:170}
Node[1]{name:"B",age:34}

3 rows

Order nodes in descending order

By adding `DESC[ENDING]` after the identifier to sort on, the sort will be done in reverse order.

Query

```
MATCH (n)
RETURN n
ORDER BY n.name DESC
```

The example returns the nodes, sorted by their name reversely.

Result

n
Node[2]{name:"C",age:32,length:185}
Node[1]{name:"B",age:34}
Node[0]{name:"A",age:34,length:170}

3 rows

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

```
MATCH (n)
RETURN n.length, n
ORDER BY n.length
```

The nodes are returned sorted by the length property, with a node without that property last.

Result

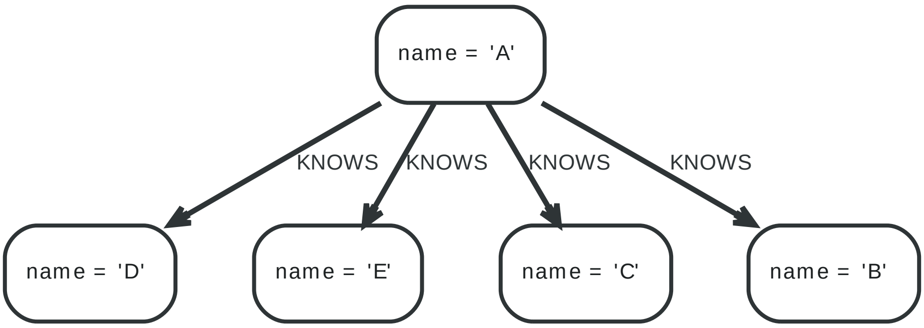
n.length	n
170	Node[0]{name:"A",age:34,length:170}
185	Node[2]{name:"C",age:32,length:185}
<null>	Node[1]{name:"B",age:34}

3 rows

10.3. Limit

`LIMIT` constrains the number of rows in the output.

Figure 10.3. Graph



Return first part

To return a subset of the result, starting from the top, use this syntax:

Query

```
MATCH (n)
RETURN n
ORDER BY n.name
LIMIT 3
```

The top three items are returned by the example query.

Result

n
Node[2]{name:"A"}
Node[3]{name:"B"}
Node[4]{name:"C"}

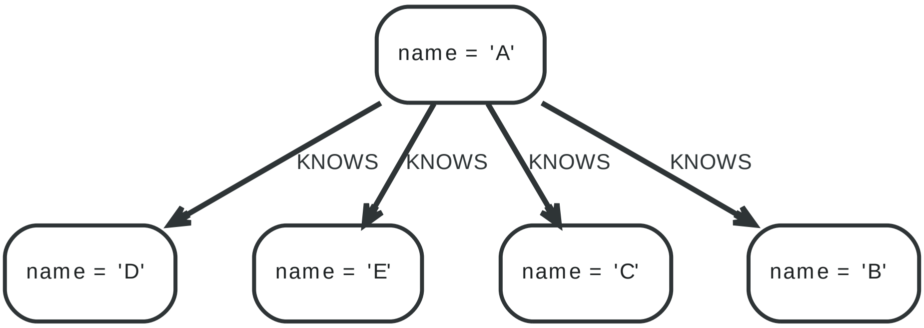
3 rows

10.4. Skip

`SKIP` defines from which row to start including the rows in the output.

By using `SKIP`, the result set will get 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.

Figure 10.4. Graph



Skip first three

To return a subset of the result, starting from the fourth result, use the following syntax:

Query

```
MATCH (n)
RETURN n
ORDER BY n.name
SKIP 3
```

The first three nodes are skipped, and only the last two are returned in the result.

Result

n
Node[0]{name:"D"}
Node[1]{name:"E"}

2 rows

Return middle two

To return a subset of the result, starting from somewhere in the middle, use this syntax:

Query

```
MATCH (n)
RETURN n
ORDER BY n.name
SKIP 1
LIMIT 2
```

Two nodes from the middle are returned.

Result

n
Node[3]{name:"B"}
Node[4]{name:"C"}

2 rows

10.5. With

The `WITH` clause allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.

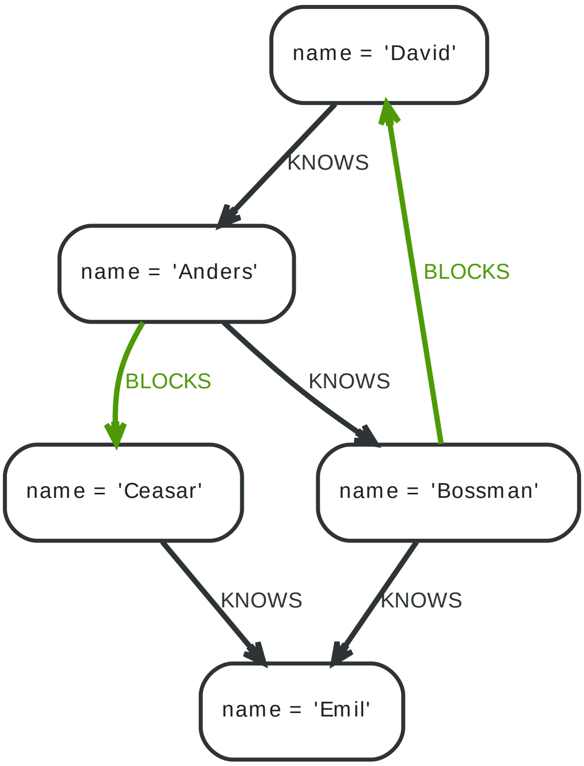
Using `WITH`, you can manipulate the output before it is passed on to the following query parts. The manipulations can be of the shape and/or number of entries in the result set.

One common usage of `WITH` is to limit the number of entries that are then passed on to other `MATCH` clauses. By combining `ORDER BY` and `LIMIT`, it's possible to get the top X entries by some criteria, and then bring in additional data from the graph.

Another use is to filter on aggregated values. `WITH` is used to introduce aggregates which can then by used in predicates in `WHERE`. These aggregate expressions create new bindings in the results. `WITH` can also, like `RETURN`, alias expressions that are introduced into the results using the aliases as binding name.

`WITH` is also used to separate reading from updating of the graph. Every part of a query must be either read-only or write-only. When going from a writing part to a reading part, the switch must be done with a `WITH` clause.

Figure 10.5. Graph



Filter on aggregate function results

Aggregated results have to pass through a `WITH` clause to be able to filter on.

Query

```
MATCH (david { name: "David" })--(otherPerson)-->()
WITH otherPerson, count(*) AS foaf
WHERE foaf > 1
RETURN otherPerson
```

The person connected to David with the at least more than one outgoing relationship will be returned by the query.

Result

otherPerson
Node[2]{name:"Anders"}

1 row

Sort results before using collect on them

You can sort your results before passing them to collect, thus sorting the resulting collection.

Query

```
MATCH (n)
WITH n
ORDER BY n.name DESC LIMIT 3
RETURN collect(n.name)
```

A list of the names of people in reverse order, limited to 3, in a collection.

Result

collect(n.name)
["Emil","David","Ceasar"]

1 row

Limit branching of your path search

You can match paths, limit to a certain number, and then match again using those paths as a base As well as any number of similar limited searches.

Query

```
MATCH (n { name: "Anders" })--(m)
WITH m
ORDER BY m.name DESC LIMIT 1
MATCH (m)--(o)
RETURN o.name
```

Starting at Anders, find all matching nodes, order by name descending and get the top result, then find all the nodes connected to that top result, and return their names.

Result

o.name
"Anders"
"Bossman"

2 rows

10.6. Unwind

`UNWIND` expands a collection into a sequence of rows.

With `UNWIND`, you can transform any collection back into individual rows. These collections can be parameters that were passed in, previously `COLLECT`ed result or other collection expressions.

One common usage of unwind is to create distinct collections. Another is to create data from parameter collections that are provided to the query.

`UNWIND` requires you to specify a new name for the inner values.

Unwind a collection

We want to transform the literal collection into rows named `x` and return them.

Query

```
UNWIND[1,2,3] AS x
RETURN x
```

Each value of the original collection is returned as an individual row.

Result

x
1
2
3
3 rows

Create a distinct collection

We want to transform a collection of duplicates into a set using `DISTINCT`.

Query

```
WITH [1,1,2,2] AS coll UNWIND coll AS x
WITH DISTINCT x
RETURN collect(x) AS SET
```

Each value of the original collection is unwound and passed through `DISTINCT` to create a unique set.

Result

set
[1,2]
1 row

Create nodes from a collection parameter

Create a number of nodes and relationships from a parameter-list without using `FOREACH`.

Parameters

```
{
  "events" : [ {
    "year" : 2014,
    "id" : 1
  }, {
    "year" : 2014,
    "id" : 2
  } ]
```

```
}
```

Query

```
UNWIND { events } AS event
MERGE (y:Year { year:event.year })
MERGE (y)<-[:IN]-(e:Event { id:event.id })
RETURN e.id AS x
ORDER BY x
```

Each value of the original collection is unwound and passed through `MERGE` to find or create the nodes and relationships.

Result

x
1
2

2 rows
Nodes created: 3
Relationships created: 2
Properties set: 3
Labels added: 3

10.7. Union

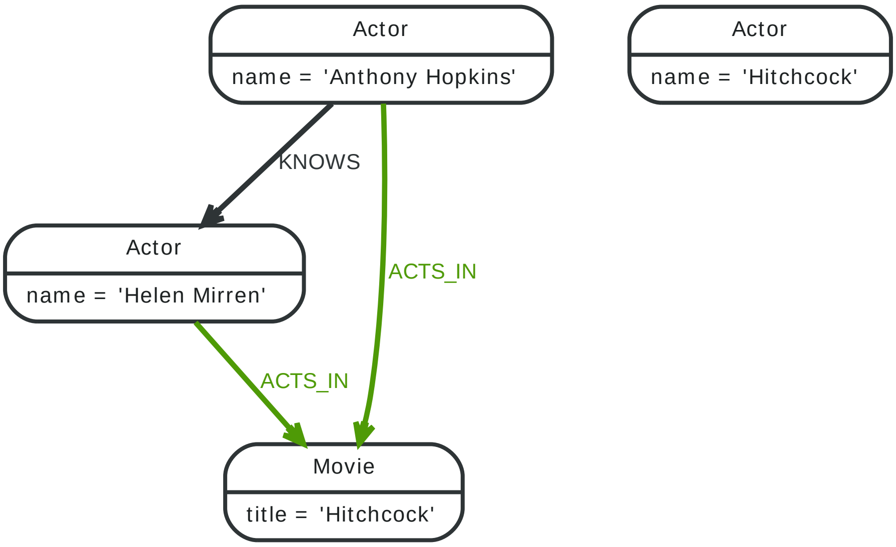
The `UNION` clause is used to combine the result of multiple queries.

It combines the results of two or more queries into a single result set that includes all the rows that belong to all queries in the union.

The number and the names of the columns must be identical in all queries combined by using `UNION`.

To keep all the result rows, use `UNION ALL`. Using just `UNION` will combine and remove duplicates from the result set.

Figure 10.6. Graph



Combine two queries

Combining the results from two queries is done using `UNION ALL`.

Query

```
MATCH (n:Actor)
RETURN n.name AS name
UNION ALL MATCH (n:Movie)
RETURN n.title AS name
```

The combined result is returned, including duplicates.

Result

name
"Anthony Hopkins"
"Helen Mirren"
"Hitchcock"
"Hitchcock"

4 rows

Combine two queries and remove duplicates

By not including `ALL` in the `UNION`, duplicates are removed from the combined result set

Query

```
MATCH (n:Actor)
```

```
RETURN n.name AS name
UNION
MATCH (n:Movie)
RETURN n.title AS name
```

The combined result is returned, without duplicates.

Result

name
"Anthony Hopkins"
"Helen Mirren"
"Hitchcock"

3 rows

10.8. Using

`USING` is used to give Neo4j hints about which index to use, or to scan instead of using an index.

If you do not specify an explicit `START` clause, Cypher needs to infer where in the graph to start your query. This is done by looking at the `MATCH` clause and the `WHERE` conditions and using that information to find a useful index.

This index might not be the best choice though — sometimes multiple indexes could be used, and Neo4j has picked the wrong one (from a performance point of view).

You can force Neo4j to use a specific starting point through the `USING` clause. This is called giving an index hint.

If your query matches large parts of an index, it might be faster to scan the label and filter out nodes that do not match. To do this, you can use `USING SCAN`. It will force Cypher to not use an index that could have been used, and instead do a label scan.

Note
You cannot use index hints if your query has a `START` clause.

Query using an index hint

To query using an index hint, use `USING INDEX`.

Query

```
MATCH (n:Swedish)
USING INDEX n:Swedish(surname)
WHERE n.surname = 'Taylor'
RETURN n
```

The query result is returned as usual.

Result

n
Node[3]{name:"Andres",age:36,awesome:true,surname:"Taylor"}

1 row

Query using multiple index hints

To query using multiple index hints, use `USING INDEX`.

Query

```
MATCH (m:German)-->(n:Swedish)
USING INDEX m:German(surname)
USING INDEX n:Swedish(surname)
WHERE m.surname = 'Plantikow' AND n.surname = 'Taylor'
RETURN m
```

The query result is returned as usual.

Result

m
Node[1]{name:"Stefan",surname:"Plantikow"}

1 row

Hinting a label scan

If the best performance is to be had by scanning all nodes in a label and then filtering on that set, use `USING SCAN`.

Query

```
MATCH (m:German)
USING SCAN m:German
WHERE m.surname = 'Plantikow'
RETURN m
```

This query does its work by finding all `:German` labeled nodes and filtering them by the surname property.

Result

m
Node[1]{name:"Stefan",surname:"Plantikow"}
1 row

Chapter 11. Reading Clauses

The flow of data within a Cypher query is an unordered sequence of maps with key-value pairs — a set of possible bindings between the identifiers in the query and values derived from the database. This set is refined and augmented by subsequent parts of the query.

11.1. Match

The `MATCH` clause is used to search for the pattern described in it.

Introduction

The `MATCH` clause allows you to specify the patterns Neo4j will search for in the database. This is the primary way of getting data into the current set of bindings. It is worth reading up more on the specification of the patterns themselves in Section 9.6, “Patterns” [125].

`MATCH` is often coupled to a `WHERE` part which adds restrictions, or predicates, to the `MATCH` patterns, making them more specific. The predicates are part of the pattern description, not a filter applied after the matching is done. *This means that `WHERE` should always be put together with the `MATCH` clause it belongs to.*

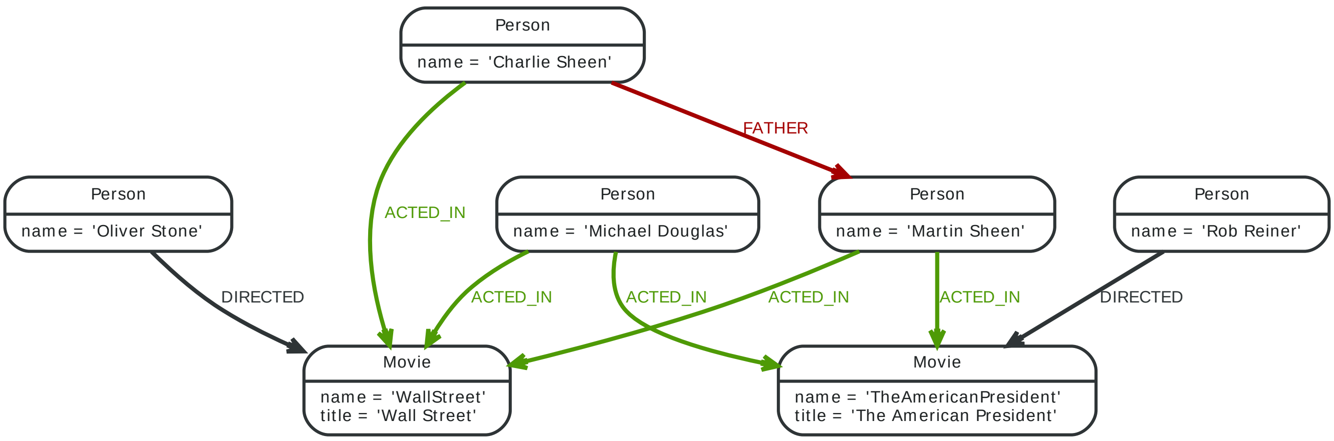
`MATCH` can occur at the beginning of the query or later, possibly after a `WITH`. If it is the first clause, nothing will have been bound yet, and Neo4j will design a search to find the results matching the clause and any associated predicates specified in any `WHERE` part. This could involve a scan of the database, a search for nodes of a certain label, or a search of an index to find starting points for the pattern matching. Nodes and relationships found by this search are available as *bound pattern elements,* and can be used for pattern matching of sub-graphs. They can also be used in any further `MATCH` clauses, where Neo4j will use the known elements, and from there find further unknown elements.

Cypher is declarative, and so usually the query itself does not specify the algorithm to use to perform the search. Neo4j will automatically work out the best approach to finding start nodes and matching patterns. Predicates in `WHERE` parts can be evaluated before pattern matching, during pattern matching, or after finding matches. However, there are cases where you can influence the decisions taken by the query compiler. Read more about indexes in Section 14.1, “Indexes” [236], and more about the specifying index hints to force Neo4j to use a specific index in Section 10.8, “Using” [149].

Tip
To understand more about the patterns used in the `MATCH` clause, read Section 9.6, “Patterns” [125].

The following graph is used for the examples below:

Figure 11.1. Graph



Basic node finding

Get all nodes

By just specifying a pattern with a single node and no labels, all nodes in the graph will be returned.

Query

```
MATCH (n)
RETURN n
```

Returns all the nodes in the database.

Result

n
Node[0]{name:"Oliver Stone"}
Node[1]{name:"Charlie Sheen"}
Node[2]{name:"Martin Sheen"}
Node[3]{name:"TheAmericanPresident",title:"The American President"}
Node[4]{name:"WallStreet",title:"Wall Street"}
Node[5]{name:"Rob Reiner"}
Node[6]{name:"Michael Douglas"}

7 rows

Get all nodes with a label

Getting all nodes with a label on them is done with a single node pattern where the node has a label on it.

Query

```
MATCH (movie:Movie)
RETURN movie
```

Returns all the movies in the database.

Result

movie
Node[3]{name:"TheAmericanPresident",title:"The American President"}
Node[4]{name:"WallStreet",title:"Wall Street"}

2 rows

Related nodes

The symbol -- means *related to,* without regard to type or direction of the relationship.

Query

```
MATCH (director { name:'Oliver Stone' })--(movie)
RETURN movie.title
```

Returns all the movies directed by Oliver Stone.

Result

movie.title
"Wall Street"

1 row

Match with labels

To constrain your pattern with labels on nodes, you add it to your pattern nodes, using the label syntax.

Query

```
MATCH (charlie:Person { name:'Charlie Sheen' })--(movie:Movie)
RETURN movie
```

Return any nodes connected with the `Person` Charlie that are labeled `Movie`.

Result

movie
`Node[4]{name:"WallStreet",title:"Wall Street"}`

1 row

Relationship basics

Outgoing relationships

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

Query

```
MATCH (martin { name:'Martin Sheen' })-->(movie)
RETURN movie.title
```

Returns nodes connected to Martin by outgoing relationships.

Result

movie.title
`"The American President"`
`"Wall Street"`

2 rows

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

```
MATCH (martin { name:'Martin Sheen' })-[r]->(movie)
RETURN r
```

Returns all outgoing relationships from Martin.

Result

r
`:ACTED_IN[3]{}`
`:ACTED_IN[1]{}`

2 rows

Match by relationship type

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

Query

```
MATCH (wallstreet { title:'Wall Street' })<-[:ACTED_IN]-(actor)
RETURN actor
```

Returns nodes that `ACTED_IN` Wall Street.

Result

actor
Node[6]{name:"Michael Douglas"}
Node[2]{name:"Martin Sheen"}
Node[1]{name:"Charlie Sheen"}

3 rows

Match by multiple relationship types

To match on one of multiple types, you can specify this by chaining them together with the pipe symbol `|`.

Query

```
MATCH (wallstreet { title:'Wall Street' })<-[:ACTED_IN|:DIRECTED]-(person)
RETURN person
```

Returns nodes with a `ACTED_IN` or `DIRECTED` relationship to Wall Street.

Result

person
Node[6]{name:"Michael Douglas"}
Node[2]{name:"Martin Sheen"}
Node[1]{name:"Charlie Sheen"}
Node[0]{name:"Oliver Stone"}

4 rows

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

```
MATCH (wallstreet { title:'Wall Street' })<-[r:ACTED_IN]-(actor)
RETURN r
```

Returns nodes that `ACTED_IN` Wall Street.

Result

r
:ACTED_IN[2]{}
:ACTED_IN[1]{}
:ACTED_IN[0]{}

3 rows

Relationships in depth

Note
Inside a single pattern, relationships will only be matched once. You can read more about this in Section 8.4, "Uniqueness" [110].

Relationship types with uncommon characters

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

Query

```
MATCH (n { name:'Rob Reiner' })-[r:`TYPE THAT HAS SPACE IN IT`]->()
RETURN r
```

Returns a relationship of a type with spaces in it.

Result

r
:TYPE THAT HAS SPACE IN IT[8]{}

1 row

Multiple relationships

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

Query

```
MATCH (charlie { name:'Charlie Sheen' })-[:ACTED_IN]->(movie)<-[:DIRECTED]-(director)
RETURN charlie,movie,director
```

Returns the three nodes in the path.

Result

charlie	movie	director
Node[1]{name:"Charlie Sheen"}	Node[4]{name:"WallStreet", title:"Wall Street"}	Node[0]{name:"Oliver Stone"}

1 row

Variable length relationships

Nodes that are a variable number of relationship→node hops away can be found using the following syntax: `-[:TYPE*minHops..maxHops]->`. minHops and maxHops are optional and default to 1 and infinity respectively. When no bounds are given the dots may be omitted.

Query

```
MATCH (martin { name:"Martin Sheen" })-[:ACTED_IN*1..2]-(x)
RETURN x
```

Returns nodes that are 1 or 2 relationships away from Martin.

Result

x
Node[4]{name:"WallStreet",title:"Wall Street"}
Node[1]{name:"Charlie Sheen"}
Node[6]{name:"Michael Douglas"}
Node[3]{name:"TheAmericanPresident",title:"The American President"}
Node[6]{name:"Michael Douglas"}

5 rows

Relationship identifier in variable length relationships

When the connection between two nodes is of variable length, a relationship identifier becomes an collection of relationships.

Query

```
MATCH (actor { name:'Charlie Sheen' })-[r:ACTED_IN*2]-(co_actor)
```

```
RETURN r
```

The query returns a collection of relationships.

Result

r
[:ACTED_IN[0]{},:ACTED_IN[1]{}]
[:ACTED_IN[0]{},:ACTED_IN[2]{}]

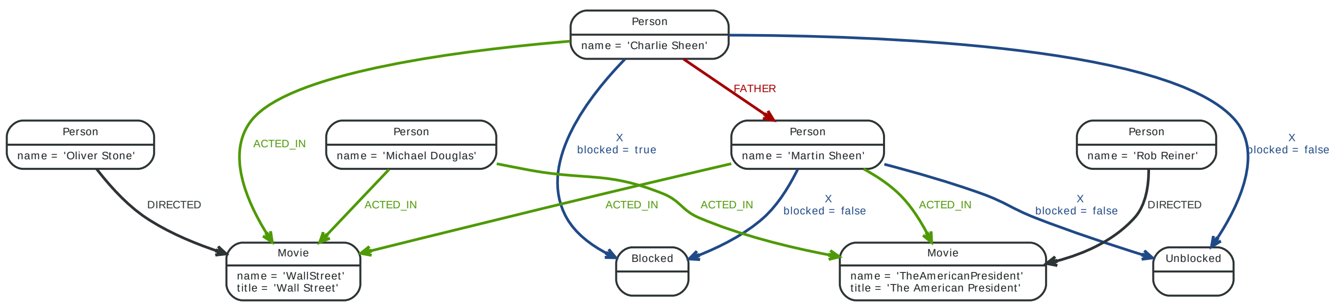
2 rows

Match with properties on a variable length path

A variable length relationship with properties defined on in it means that all relationships in the path must have the property set to the given value. In this query, there are two paths between Charile Sheen and his dad Martin Sheen. One of the includes a “blocked” relationship and the other doesn’t. In this case we first alter the original graph by using the following query to add “blocked” and “unblocked” relationships:

```
MATCH (charlie:Person { name:'Charlie Sheen' }),(martin:Person { name:'Martin Sheen' })
CREATE (charlie)-[:X { blocked:false }]->(:Unblocked)<-[:X { blocked:false }]-(martin)
CREATE (charlie)-[:X { blocked:true }]->(:Blocked)<-[:X { blocked:false }]-(martin);
```

This means that we are starting out with the following graph:



Query

```
MATCH p =(charlie:Person)-[* { blocked:false }]-(martin:Person)
WHERE charlie.name = 'Charlie Sheen' AND martin.name = 'Martin Sheen'
RETURN p
```

Returns the paths between Charlie and Martin Sheen where all relationships have the `blocked` property set to `FALSE`.

Result

p
[Node[1]{name:"Charlie Sheen"},:X[8]{blocked:false},Node[7]{},:X[9]{blocked:false},Node[2]{name:"Martin Sheen"}]

1 row

Zero length paths

Using variable length paths that have the lower bound zero 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. Note that when matching zero length paths the result may contain a match even when matching on a relationship type not in use.

Query

```
MATCH (wallstreet:Movie { title:'Wall Street' })-[*0..1]-(x)
```

```
RETURN x
```

Returns all nodes that are zero or one relationships away from Wall Street.

Result

x
Node[4]{name:"WallStreet",title:"Wall Street"}
Node[0]{name:"Oliver Stone"}
Node[1]{name:"Charlie Sheen"}
Node[2]{name:"Martin Sheen"}
Node[6]{name:"Michael Douglas"}

5 rows

Named path

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

Query

```
MATCH p =(michael { name:'Michael Douglas' })-->()
RETURN p
```

Returns the two paths starting from Michael.

Result

p
[Node[6]{name:"Michael Douglas"},:ACTED_IN[4]{},Node[3]{name:"TheAmericanPresident",title:"The American President"}]
[Node[6]{name:"Michael Douglas"},:ACTED_IN[2]{},Node[4]{name:"WallStreet",title:"Wall Street"}]

2 rows

Matching on a bound relationship

When your pattern contains a bound relationship, and that relationship pattern doesn't specify direction, Cypher will try to match the relationship in both directions.

Query

```
MATCH (a)-[r]-(b)
WHERE id(r)= 0
RETURN a,b
```

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

Result

a	b
Node[1]{name:"Charlie Sheen"}	Node[4]{name:"WallStreet",title:"Wall Street"}
Node[4]{name:"WallStreet",title:"Wall Street"}	Node[1]{name:"Charlie Sheen"}

2 rows

Shortest path

Single shortest path

Finding a single shortest path between two nodes is as easy as using the `shortestPath` function. It's done like this:

Query

```
MATCH (martin:Person { name:"Martin Sheen" }),(oliver:Person { name:"Oliver Stone" }),
  p = shortestPath((martin)-[*..15]-(oliver))
RETURN p
```

This means: find a single shortest path between two nodes, as long as the path is max 15 relationships long. Inside of the parentheses you define 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. You can also mark the path as optional.

Result

p
`[Node[2]{name:"Martin Sheen"},:ACTED_IN[1]{},Node[4]{name:"WallStreet",title:"Wall Street"},:DIRECTED[5]{},Node[0]{name:"Oliver Stone"}]`

1 row

All shortest paths

Finds all the shortest paths between two nodes.

Query

```
MATCH (martin:Person { name:"Martin Sheen" }),(michael:Person { name:"Michael Douglas" }),
  p = allShortestPaths((martin)-[*]-(michael))
RETURN p
```

Finds the two shortest paths between Martin and Michael.

Result

p
`[Node[2]{name:"Martin Sheen"},:ACTED_IN[3]{},Node[3]{name:"TheAmericanPresident",title:"The American President"},:ACTED_IN[4]{},Node[6]{name:"Michael Douglas"}]`
`[Node[2]{name:"Martin Sheen"},:ACTED_IN[1]{},Node[4]{name:"WallStreet",title:"Wall Street"},:ACTED_IN[2]{},Node[6]{name:"Michael Douglas"}]`

2 rows

Get node or relationship by id

Node by id

Search for nodes by id can be done with the *id* function in a predicate.

Note
Neo4j reuses its internal ids when nodes and relationships are deleted. This means that applications using, and relying on internal Neo4j ids, are brittle or at risk of making mistakes. Rather use application generated ids.

Query

```
MATCH n
WHERE id(n)= 1
RETURN n
```

The corresponding node is returned.

Result

n
`Node[1]{name:"Charlie Sheen"}`

1 row

Relationship by id

Search for nodes by id can be done with the *id* function in a predicate.

This is not recommended practice. See the section called “Node by id” [159] for more information on the use of Neo4j ids.

Query

```
MATCH ()-[r]->()
WHERE id(r)= 0
RETURN r
```

The relationship with id `0` is returned.

Result

r
:ACTED_IN[0]{}

1 row

Multiple nodes by id

Multiple nodes are selected by specifying them in an IN clause.

Query

```
MATCH n
WHERE id(n) IN [1, 2, 0]
RETURN n
```

This returns the nodes listed in the `IN` expression.

Result

n
Node[0]{name:"Oliver Stone"}
Node[1]{name:"Charlie Sheen"}
Node[2]{name:"Martin Sheen"}

3 rows

11.2. Optional Match

The `OPTIONAL MATCH` clause is used to search for the pattern described in it, while using `NULL`s for missing parts of the pattern.

Introduction

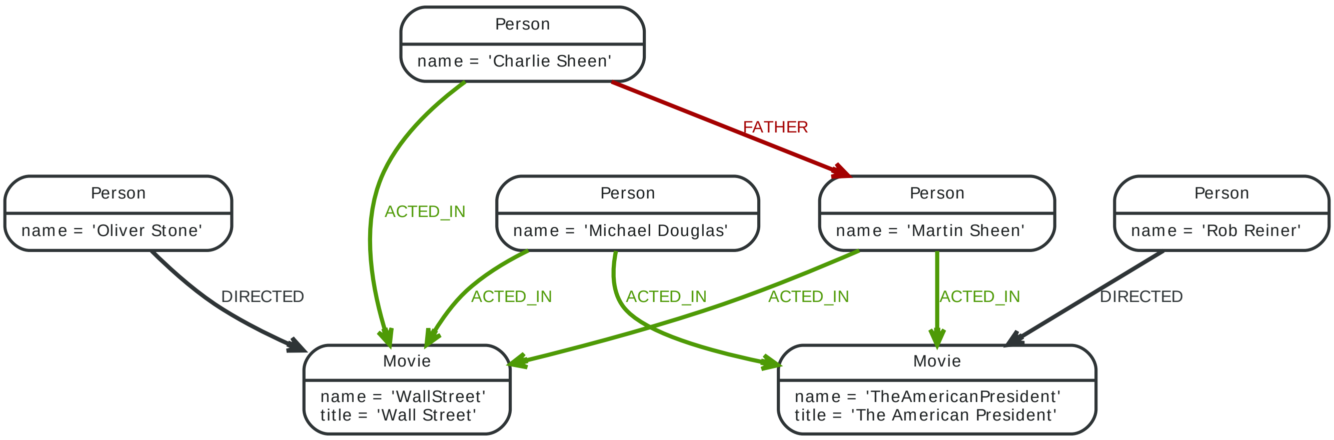
`OPTIONAL MATCH` matches patterns against your graph database, just like `MATCH` does. The difference is that if no matches are found, `OPTIONAL MATCH` will use `NULL`s for missing parts of the pattern. `OPTIONAL MATCH` could be considered the Cypher equivalent of the outer join in SQL.

Either the whole pattern is matched, or nothing is matched. Remember that `WHERE` is part of the pattern description, and the predicates will be considered while looking for matches, not after. This matters especially in the case of multiple (`OPTIONAL`) `MATCH` clauses, where it is crucial to put `WHERE` together with the `MATCH` it belongs to.

Tip
To understand the patterns used in the `OPTIONAL MATCH` clause, read Section 9.6, “Patterns” [125].

The following graph is used for the examples below:

Figure 11.2. Graph



Relationship

If a relationship is optional, use the `OPTIONAL MATCH` clause. This is 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.

Query

```
MATCH (a:Movie { title: 'Wall Street' })
OPTIONAL MATCH (a)-->(x)
RETURN x
```

Returns `NULL`, since the node has no outgoing relationships.

Result

x
<null>

1 row

Properties on optional elements

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

Query

```
MATCH (a:Movie { title: 'Wall Street' })
OPTIONAL MATCH (a)-->(x)
RETURN x, x.name
```

Returns the element x (`NULL` in this query), and `NULL` as its name.

Result

x	x.name
<null>	<null>

1 row

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

```
MATCH (a:Movie { title: 'Wall Street' })
OPTIONAL MATCH (a)-[r:ACTS_IN]->()
RETURN r
```

This returns a node, and `NULL`, since the node has no outgoing `ACTS_IN` relationships.

Result

r
<null>

1 row

11.3. Where

`WHERE` adds constraints to the patterns in an `(OPTIONAL) MATCH` clause or filters the results of a `WITH` clause.

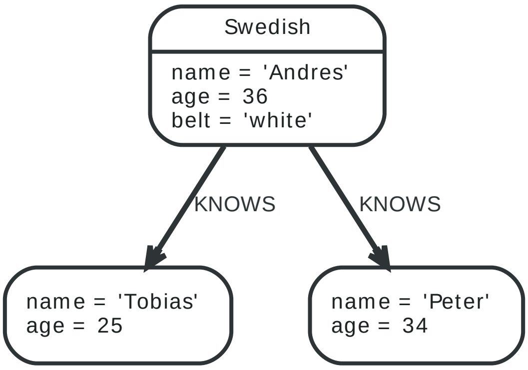
`WHERE` is not a clause in it's own right — rather, it's part of `MATCH`, `OPTIONAL MATCH`, `START` and `WITH`.

In the case of `WITH` and `START`, `WHERE` simply filters the results.

For `MATCH` and `OPTIONAL MATCH` on the other hand, `WHERE` adds constraints to the patterns described. It should not be seen as a filter after the matching is finished.

> **Note**
> In the case of multiple (`OPTIONAL`) `MATCH` clauses, the predicate in `WHERE` is always a part of the patterns in the directly preceding `MATCH`. Both results and performance may be impacted if the `WHERE` is put inside the wrong `MATCH` clause.

Figure 11.3. Graph



Basic usage

Boolean operations

You can use the expected boolean operators `AND` and `OR`, and also the boolean function `NOT`. See Section 9.8, "Working with NULL" [132] for more information on how this works with `NULL`.

Query

```
MATCH (n)
WHERE n.name = 'Peter' XOR (n.age < 30 AND n.name = "Tobias") OR NOT (n.name = "Tobias" OR
  n.name="Peter")
RETURN n
```

This query shows how boolean operators can be used.

Result

n
`Node[0]{name:"Tobias",age:25}`
`Node[1]{name:"Peter",age:34}`
`Node[2]{name:"Andres",age:36,belt:"white"}`
3 rows

Filter on node label

To filter nodes by label, write a label predicate after the `WHERE` keyword using `WHERE n:foo`.

Query

```
MATCH (n)
WHERE n:Swedish
RETURN n
```

The "`Andres`" node will be returned.

Result

n
Node[2]{name:"Andres",age:36,belt:"white"}

1 row

Filter on node property

To filter on a property, write your clause after the `WHERE` keyword. Filtering on relationship properties works just the same way.

Query

```
MATCH (n)
WHERE n.age < 30
RETURN n
```

The "`Tobias`" node will be returned.

Result

n
Node[0]{name:"Tobias",age:25}

1 row

Property exists

To only include nodes/relationships that have a property, use the `HAS()` function and just write out the identifier and the property you expect it to have.

Query

```
MATCH (n)
WHERE HAS (n.belt)
RETURN n
```

The node named "`Andres`" is returned.

Result

n
Node[2]{name:"Andres",age:36,belt:"white"}

1 row

Regular expressions

Regular expressions

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

Query

```
MATCH (n)
WHERE n.name =~ 'Tob.*'
RETURN n
```

The "`Tobias`" node will be returned.

Result

n
Node[0]{name:"Tobias",age:25}

1 row

Escaping in regular expressions

If you need a forward slash inside of your regular expression, escape it. Remember that back slash needs to be escaped in string literals

Query

```
MATCH (n)
WHERE n.name =~ 'Some\/thing'
RETURN n
```

No nodes match this regular expression.

Result

n
(empty result)

0 row

Case insensitive regular expressions

By pre-pending a regular expression with `(?i)`, the whole expression becomes case insensitive.

Query

```
MATCH (n)
WHERE n.name =~ '(?i)ANDR.*'
RETURN n
```

The node with name "`Andres`" is returned.

Result

n
Node[2]{name:"Andres",age:36,belt:"white"}

1 row

Using patterns in WHERE

Filter on patterns

Patterns are expressions in Cypher, expressions that return a collection of paths. Collection expressions are also predicates — an empty collection represents `false`, and a non-empty represents `true`.

So, patterns are not only expressions, they are also predicates. The only limitation to your pattern is that you must be able to express it in a single path. You can not use commas between multiple paths like you do in `MATCH`. You can achieve the same effect by combining multiple patterns with `AND`.

Note that you can not introduce new identifiers here. Although it might look very similar to the `MATCH` patterns, the `WHERE` clause is all about eliminating matched subgraphs. `MATCH (a)-[*]->(b)` is very different from `WHERE (a)-[*]->(b)`; the first will produce a subgraph for every path it can find between `a` and `b`, and the latter will eliminate any matched subgraphs where `a` and `b` do not have a directed relationship chain between them.

Query

```
MATCH (tobias { name: 'Tobias' }),(others)
WHERE others.name IN ['Andres', 'Peter'] AND (tobias)<--(others)
RETURN others
```

Nodes that have an outgoing relationship to the "Tobias" node are returned.

Result

others
Node[2]{name:"Andres",age:36,belt:"white"}

1 row

Filter on patterns using NOT

The NOT function can be used to exclude a pattern.

Query

```
MATCH (persons),(peter { name: 'Peter' })
WHERE NOT (persons)-->(peter)
RETURN persons
```

Nodes that do not have an outgoing relationship to the "Peter" node are returned.

Result

persons
Node[0]{name:"Tobias",age:25}
Node[1]{name:"Peter",age:34}

2 rows

Filter on patterns with properties

You can also add properties to your patterns:

Query

```
MATCH (n)
WHERE (n)-[:KNOWS]-({ name:'Tobias' })
RETURN n
```

Finds all nodes that have a KNOWS relationship to a node with the name Tobias.

Result

n
Node[2]{name:"Andres",age:36,belt:"white"}

1 row

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

```
MATCH (n)-[r]->()
WHERE n.name='Andres' AND type(r)=~ 'K.*'
RETURN r
```

This returns relationships that has a type whose name starts with K.

Result

r
:KNOWS[1]{}

2 rows

r
:KNOWS[0]{}

2 rows

Collections

IN operator

To check if an element exists in a collection, you can use the `IN` operator.

Query

```
MATCH (a)
WHERE a.name IN ["Peter", "Tobias"]
RETURN a
```

This query shows how to check if a property exists in a literal collection.

Result

a
Node[0]{name:"Tobias",age:25}
Node[1]{name:"Peter",age:34}

2 rows

Missing properties and values

Default to false if property is missing

As missing properties evaluate to `NULL`, the comparision in the example will evaluate to `FALSE` for nodes without the `belt` property.

Query

```
MATCH (n)
WHERE n.belt = 'white'
RETURN n
```

Only nodes with the belt property are returned.

Result

n
Node[2]{name:"Andres",age:36,belt:"white"}

1 row

Default to true if property is missing

If you want to compare a property on a graph element, but only if it exists, you can compare the property against both the value you are looking for and `NULL`, like:

Query

```
MATCH (n)
WHERE n.belt = 'white' OR n.belt IS NULL RETURN n
ORDER BY n.name
```

This returns all nodes, even those without the belt property.

Result

n
Node[2]{name:"Andres",age:36,belt:"white"}

3 rows

n
Node[1]{name:"Peter",age:34}
Node[0]{name:"Tobias",age:25}

3 rows

Filter on NULL

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`, although `NOT(IS NULL x)` also works.

Query

```
MATCH (person)
WHERE person.name = 'Peter' AND person.belt IS NULL RETURN person
```

Nodes that have name *Peter* but no belt property are returned.

Result

person
Node[1]{name:"Peter",age:34}

1 row

11.4. Start

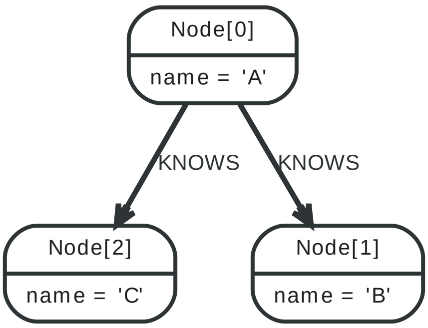
Find starting points through legacy indexes.

> **Important**
> The START clause should only be used when accessing legacy indexes (see Chapter 37, *Legacy Indexing* [632]). In all other cases, use MATCH instead (see Section 11.1, “Match” [152]).

In Cypher, every query describes a pattern, and in that pattern one can have multiple starting points. A starting point is a relationship or a node where a pattern is anchored. Using START you can only introduce starting points by legacy index lookups. Note that trying to use a legacy index that doesn’t exist will generate an error.

This is the graph the examples are using:

Figure 11.4. Graph



Get node or relationship from index

Node by index lookup

When the starting point can be found by using 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 query returns the node indexed with the name "`A`".

Result

n
Node[0]{name:"A"}
1 row

Relationship by index lookup

When the starting point can be found by using index lookups, it can be done like this: `relationship:index-name(key = "value")`.

Query

```
START r=relationship:rels(name = "Andrés")
RETURN r
```

The relationship indexed with the `name` property set to "`Andrés`" is returned by the query.

Result

r
:KNOWS[0]{name:"Andrés"

1 row

Node by index query

When the starting point can be found by more complex Lucene queries, this is the syntax to use: `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 by the query.

Result

n
Node[0]{name:"A"}

1 row

11.5. Aggregation

Introduction

To calculate aggregated data, Cypher offers aggregation, much like SQL's `GROUP BY`.

Aggregate functions take multiple input values and calculate an aggregated value from them. Examples are `avg` that calculates the average of multiple numeric values, or `min` that finds the smallest numeric value in a set of values.

Aggregation can be done over all the matching subgraphs, or it can be further divided by introducing key values. These are non-aggregate expressions, that are used to group the values going into the aggregate functions.

So, if the return statement looks something like this:

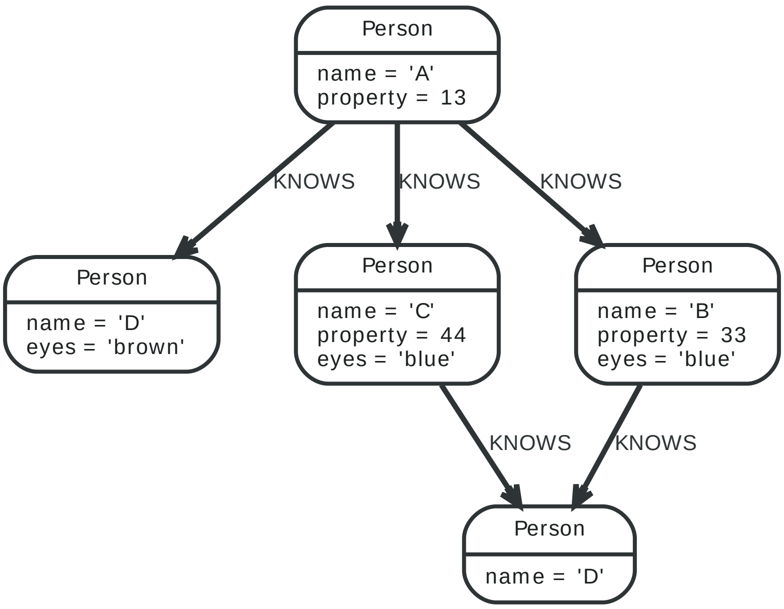
```
RETURN n, count(*)
```

We have two return expressions: `n`, and `count(*)`. The first, `n`, is no aggregate function, and so it will be the grouping key. The latter, `count(*)` is an aggregate expression. So the matching subgraphs will be divided into different buckets, depending on the grouping key. The aggregate function will then run on these buckets, calculating the aggregate values.

If you want to use aggregations to sort your result set, the aggregation must be included in the `RETURN` to be used in your `ORDER BY`.

The last piece of the puzzle is the `DISTINCT` keyword. It is used to make all values unique before running them through an aggregate function.

An example might be helpful. In this case, we are running the query against the following data:



Query

```
MATCH (me:Person)-->(friend:Person)-->(friend_of_friend:Person)
WHERE me.name = 'A'
RETURN count(DISTINCT friend_of_friend), count(friend_of_friend)
```

In this example we are trying to find all our friends of friends, and count them. The first aggregate function, `count(DISTINCT friend_of_friend)`, will only see a `friend_of_friend` once — `DISTINCT` removes the duplicates. The latter aggregate function, `count(friend_of_friend)`, might very well see the same `friend_of_friend` multiple times. In this case, both `B` and `C` know `D` and thus `D` will get counted twice, when not using `DISTINCT`.

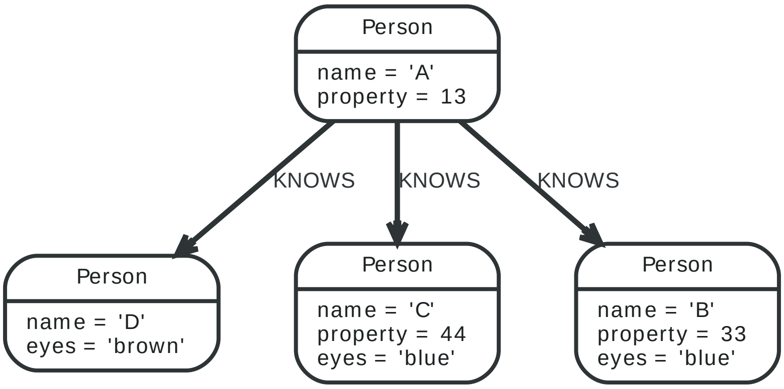
Result

count(distinct friend_of_friend)	count(friend_of_friend)
1	2

1 row

The following examples are assuming the example graph structure below.

Figure 11.5. Graph



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>`.

Count nodes

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

Query

```
MATCH (n { name: 'A' })-->(x)
RETURN n, count(*)
```

This returns the start node and the count of related nodes.

Result

n	count(*)
Node[1]{name:"A",property:13}	3

1 row

Group Count Relationship Types

To count the groups of relationship types, return the types and count them with `count(*)`.

Query

```
MATCH (n { name: 'A' })-[r]->()
RETURN type(r), count(*)
```

The relationship types and their group count is returned by the query.

Result

type(r)	count(*)
"KNOWS"	3

1 row

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

```
MATCH (n { name: 'A' })-->(x)
RETURN count(x)
```

The example query returns the number of connected nodes from the start node.

Result

count(x)
3

1 row

Count non-null values

You can count the non-`NULL` values by using `count(<identifier>)`.

Query

```
MATCH (n:Person)
RETURN count(n.property)
```

The count of related nodes with the `property` property set is returned by the query.

Result

count(n.property)
3

1 row

Statistics

sum

The `sum` aggregation function simply sums all the numeric values it encounters. `NULL`s are silently dropped.

Query

```
MATCH (n:Person)
RETURN sum(n.property)
```

This returns the sum of all the values in the property `property`.

Result

sum(n.property)
90

1 row

avg

`avg` calculates the average of a numeric column.

Query

```
MATCH (n:Person)
RETURN avg(n.property)
```

The average of all the values in the property `property` is returned by the example query.

Result

avg(n.property)
30.0

1 row

percentileDisc

`percentileDisc` calculates the percentile of a given value over a group, with a percentile from 0.0 to 1.0. It uses a rounding method, returning the nearest value to the percentile. For interpolated values, see `percentileCont`.

Query

```
MATCH (n:Person)
RETURN percentileDisc(n.property, 0.5)
```

The 50th percentile of the values in the property `property` is returned by the example query. In this case, 0.5 is the median, or 50th percentile.

Result

percentileDisc(n.property, 0.5)
33

1 row

percentileCont

`percentileCont` calculates the percentile of a given value over a group, with a percentile from 0.0 to 1.0. It uses a linear interpolation method, calculating a weighted average between two values, if the desired percentile lies between them. For nearest values using a rounding method, see `percentileDisc`.

Query

```
MATCH (n:Person)
RETURN percentileCont(n.property, 0.4)
```

The 40th percentile of the values in the property `property` is returned by the example query, calculated with a weighted average.

Result

percentileCont(n.property, 0.4)
29.0

1 row

stdev

`stdev` calculates the standard deviation for a given value over a group. It uses a standard two-pass method, with `N - 1` as the denominator, and should be used when taking a sample of the population for an unbiased estimate. When the standard variation of the entire population is being calculated, `stdevp` should be used.

Query

```
MATCH (n)
WHERE n.name IN ['A', 'B', 'C']
```

```
RETURN stdev(n.property)
```

The standard deviation of the values in the property `property` is returned by the example query.

Result

stdev(n.property)
15.716233645501712
1 row

stdevp

`stdevp` calculates the standard deviation for a given value over a group. It uses a standard two-pass method, with `N` as the denominator, and should be used when calculating the standard deviation for an entire population. When the standard variation of only a sample of the population is being calculated, `stdev` should be used.

Query

```
MATCH (n)
WHERE n.name IN ['A', 'B', 'C']
RETURN stdevp(n.property)
```

The population standard deviation of the values in the property `property` is returned by the example query.

Result

stdevp(n.property)
12.832251036613439
1 row

max

`max` find the largest value in a numeric column.

Query

```
MATCH (n:Person)
RETURN max(n.property)
```

The largest of all the values in the property `property` is returned.

Result

max(n.property)
44
1 row

min

`min` takes a numeric property as input, and returns the smallest value in that column.

Query

```
MATCH (n:Person)
RETURN min(n.property)
```

This returns the smallest of all the values in the property `property`.

Result

min(n.property)
13
1 row

collect

`collect` collects all the values into a list. It will ignore `NULL`s.

Query

```
MATCH (n:Person)
RETURN collect(n.property)
```

Returns a single row, with all the values collected.

Result

collect(n.property)
[13,33,44]

1 row

DISTINCT

All aggregation functions also take the `DISTINCT` modifier, which removes duplicates from the values. So, to count the number of unique eye colors from nodes related to `a`, this query can be used:

Query

```
MATCH (a:Person { name: 'A' })-->(b)
RETURN count(DISTINCT b.eyes)
```

Returns the number of eye colors.

Result

count(distinct b.eyes)
2

1 row

11.6. Load CSV

`LOAD CSV` is used to import data from CSV files.

It is required to specify an identifier for the CSV data using `AS`. See the examples below for further details.

There is also a worked example, see Section 12.8, “Importing CSV files with Cypher” [203].

CSV file format

The CSV file to use with `LOAD CSV` must have the following characteristics:

- the character encoding is UTF-8;
- the end line termination is system dependent, e.g., it is `\n` on unix or `\r\n` on windows;
- the default field terminator is `,`;
- the field terminator character can be change by using the option `FIELDTERMINATOR` available in the `LOAD CSV` command;
- quoted strings are allowed in the CSV file and the quotes are dropped when reading the data;
- the character for string quotation is double quote `"`;
- the escape character is `\`.

Import data from a CSV file

To import data from a CSV file into Neo4j, you can use `LOAD CSV` to get the data into your query. Then you write it to your database using the normal updating clauses of Cypher.

artists.csv

```
"1","ABBA","1992"
"2","Roxette","1986"
"3","Europe","1979"
"4","The Cardigans","1992"
```

Query

```
LOAD CSV FROM 'http://neo4j.com/docs/2.2.1/csv/artists.csv' AS line
CREATE (:Artist { name: line[1], year: toInt(line[2])})
```

A new node with the `Artist` label is created for each row in the CSV file. In addition, two columns from the CSV file are set as properties on the nodes.

Result

```
(empty result)
```

Nodes created: 4
Properties set: 8
Labels added: 4

Import data from a CSV file containing headers

When your CSV file has headers, you can view each row in the file as a map instead of as an array of strings.

artists-with-headers.csv

```
"Id","Name","Year"
"1","ABBA","1992"
"2","Roxette","1986"
"3","Europe","1979"
```

```
"4","The Cardigans","1992"
```

Query

```
LOAD CSV WITH HEADERS FROM 'http://neo4j.com/docs/2.2.1/csv/artists-with-headers.csv' AS line
CREATE (:Artist { name: line.Name, year: toInt(line.Year)})
```

This time, the file starts with a single row containing column names. Indicate this using `WITH HEADERS` and you can access specific fields by their corresponding column name.

Result

(empty result)

Nodes created: 4
Properties set: 8
Labels added: 4

Import data from a CSV file with a custom field delimiter

Sometimes, your CSV file has other field delimiters than commas. You can specify which delimiter your file uses using `FIELDTERMINATOR`.

artists-fieldterminator.csv

```
"1";"ABBA";"1992"
"2";"Roxette";"1986"
"3";"Europe";"1979"
"4";"The Cardigans";"1992"
```

Query

```
LOAD CSV FROM 'http://neo4j.com/docs/2.2.1/csv/artists-fieldterminator.csv' AS line FIELDTERMINATOR
  ';'
CREATE (:Artist { name: line[1], year: toInt(line[2])})
```

As values in this file are separated by a semicolon, a custom `FIELDTERMINATOR` is specified in the `LOAD CSV` clause.

Result

(empty result)

Nodes created: 4
Properties set: 8
Labels added: 4

Importing large amounts of data

If the CSV file contains a significant number of rows (approaching hundreds of thousands or millions), `USING PERIODIC COMMIT` can be used to instruct Neo4j to perform a commit after a number of rows. This reduces the memory overhead of the transaction state. By default, the commit will happen every 1000 rows. For more information, see Section 12.9, “Using Periodic Commit” [205].

Query

```
USING PERIODIC COMMIT
LOAD CSV FROM 'http://neo4j.com/docs/2.2.1/csv/artists.csv' AS line
CREATE (:Artist { name: line[1], year: toInt(line[2])})
```

Result

(empty result)

Nodes created: 4
Properties set: 8
Labels added: 4

Setting the rate of periodic commits

You can set the number of rows as in the example, where it is set to 500 rows.

Query

```
USING PERIODIC COMMIT 500
LOAD CSV FROM 'http://neo4j.com/docs/2.2.1/csv/artists.csv' AS line
CREATE (:Artist { name: line[1], year: toInt(line[2])})
```

Result

(empty result)

Nodes created: 4
Properties set: 8
Labels added: 4

Import data containing escaped characters

In this example, we both have additional quotes around the values, as well as escaped quotes inside one value.

artists-with-escaped-char.csv

```
"1","The ""Symbol""","1992"
```

Query

```
LOAD CSV FROM 'http://neo4j.com/docs/2.2.1/csv/artists-with-escaped-char.csv' AS line
CREATE (a:Artist { name: line[1], year: toInt(line[2])})
RETURN a.name AS name, a.year AS year, length(a.name) AS length
```

Note that strings are wrapped in quotes in the output here. You can see that when comparing to the length of the string in this case!

Result

name	year	length
"The "Symbol""	1992	12

1 row
Nodes created: 1
Properties set: 2
Labels added: 1

Chapter 12. Writing Clauses

Write data to the database.

12.1. Create

The `CREATE` clause is used to create graph elements — nodes and relationships.

Tip
In the `CREATE` clause, patterns are used a lot. Read Section 9.6, "Patterns" [125] for an introduction.

Create nodes

Create single node

Creating a single node is done by issuing the following query.

Query

```
CREATE (n)
```

Nothing is returned from this query, except the count of affected nodes.

Result

```
(empty result)
```

Nodes created: 1

Create a node with a label

To add a label when creating a node, use the syntax below.

Query

```
CREATE (n:Person)
```

Nothing is returned from this query.

Result

```
(empty result)
```

Nodes created: 1
Labels added: 1

Create a node with multiple labels

To add labels when creating a node, use the syntax below. In this case, we add two labels.

Query

```
CREATE (n:Person:Swedish)
```

Nothing is returned from this query.

Result

```
(empty result)
```

Nodes created: 1
Labels added: 2

Create node and add labels and properties

When creating a new node with labels, you can add properties at the same time.

Query

```
CREATE (n:Person { name : 'Andres', title : 'Developer' })
```

Nothing is returned from this query.

Result

(empty result)

Nodes created: 1
Properties set: 2
Labels added: 1

Return created node

Creating a single node is done by issuing the following query.

Query

```
CREATE (a { name : 'Andres' })
RETURN a
```

The newly created node is returned.

Result

a
Node[0]{name:"Andres"}

1 row
Nodes created: 1
Properties set: 1

Create relationships

Create a relationship between two nodes

To create a relationship between two nodes, we first get the two nodes. Once the nodes are loaded, we simply create a relationship between them.

Query

```
MATCH (a:Person),(b:Person)
WHERE a.name = 'Node A' AND b.name = 'Node B'
CREATE (a)-[r:RELTYPE]->(b)
RETURN r
```

The created relationship is returned by the query.

Result

r
:RELTYPE[0]{}

1 row
Relationships created: 1

Create a relationship and set properties

Setting properties on relationships is done in a similar manner to how it's done when creating nodes. Note that the values can be any expression.

Query

```
MATCH (a:Person),(b:Person)
WHERE a.name = 'Node A' AND b.name = 'Node B'
CREATE (a)-[r:RELTYPE { name : a.name + '<->' + b.name }]->(b)
RETURN r
```

The newly created relationship is returned by the example query.

Result

r
:RELTYPE[0]{name:"Node A<->Node B"}

1 row
Relationships created: 1
Properties set: 1

Create a full path

When you use `CREATE` and a pattern, all parts of the pattern that are not already in scope at this time will be created.

Query

```
CREATE p =(andres { name:'Andres' })-[:WORKS_AT]->(neo)<-[:WORKS_AT]-(michael { name:'Michael' })
RETURN p
```

This query creates three nodes and two relationships in one go, assigns it to a path identifier, and returns it.

Result

p
[Node[0]{name:"Andres"},:WORKS_AT[0]{},Node[1]{},:WORKS_AT[1]{},Node[2]{name:"Michael"}]

1 row
Nodes created: 3
Relationships created: 2
Properties set: 2

Use parameters with CREATE

Create node with a parameter for the properties

You can also create a graph entity from a map. All the key/value pairs in the map will be set as properties on the created relationship or node. In this case we add a `Person` label to the node as well.

Parameters

```
{
  "props" : {
    "name" : "Andres",
    "position" : "Developer"
  }
}
```

Query

```
CREATE (n:Person { props })
RETURN n
```

Result

n
Node[0]{name:"Andres",position:"Developer"}

1 row
Nodes created: 1
Properties set: 2
Labels added: 1

Create multiple nodes with a parameter for their properties

By providing Cypher an array of maps, it will create a node for each map.

Note
When you do this, you can't create anything else in the same `CREATE` clause.

Parameters

```
{
  "props" : [ {
    "name" : "Andres",
    "position" : "Developer"
  }, {
    "name" : "Michael",
    "position" : "Developer"
  } ]
}
```

Query

```
CREATE (n { props })
RETURN n
```

Result

n
`Node[0]{name:"Andres",position:"Developer"}`
`Node[1]{name:"Michael",position:"Developer"}`

2 rows
Nodes created: 2
Properties set: 4

12.2. Merge

The `MERGE` clause ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.

Introduction

`MERGE` either matches existing nodes and binds them, or it creates new data and binds that. It’s like a combination of `MATCH` and `CREATE` that additionally allows you to specify what happens if the data was matched or created.

For example, you can specify that the graph must contain a node for a user with a certain name. If there isn’t a node with the correct name, a new node will be created and its name property set.

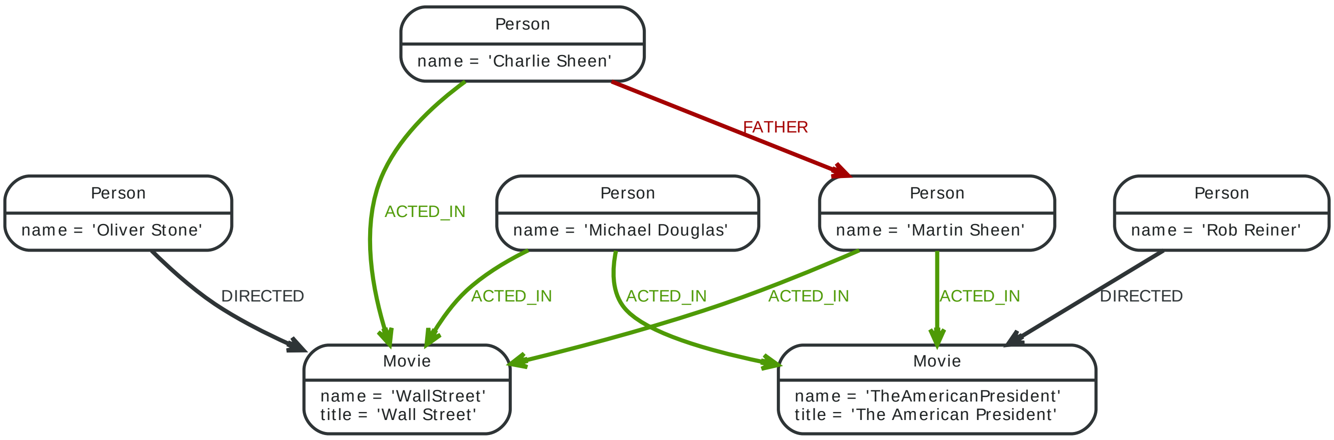
When using `MERGE` on full patterns, the behavior is that either the whole pattern matches, or the whole pattern is created. `MERGE` will not partially use existing patterns — it’s all or nothing. If partial matches are needed, this can be accomplished by splitting a pattern up into multiple `MERGE` clauses.

As with `MATCH`, `MERGE` can match multiple occurrences of a pattern. If there are multiple matches, they will all be passed on to later stages of the query.

The last part of `MERGE` is the `ON CREATE` and `ON MATCH`. These allow a query to express additional changes to the properties of a node or relationship, depending on if the element was `MATCH`ed in the database or if it was `CREATE`d.

The following graph is used for the examples below:

Figure 12.1. Graph



Merge nodes

Merge single node with a label

Merging a single node with a given label.

Query

```
MERGE (robert:Critic)
RETURN robert, labels(robert)
```

Because there are no nodes labeled `Critic` in the database, a new node is created.

Result

robert	labels(robert)
Node[7]{}	["Critic"]

1 row
Nodes created: 1
Labels added: 1

Merge single node with properties

Merging a single node with properties where not all properties match any existing node.

Query

```
MERGE (charlie { name:'Charlie Sheen', age:10 })
RETURN charlie
```

A new node with the name Charlie Sheen will be created since not all properties matched the existing Charlie Sheen node.

Result

charlie
Node[7]{name:"Charlie Sheen",age:10}

1 row
Nodes created: 1
Properties set: 2

Merge single node specifying both label and property

Merging a single node with both label and property matching an existing node.

Query

```
MERGE (michael:Person { name:'Michael Douglas' })
RETURN michael
```

Michael Douglas will be matched and returned.

Result

michael
Node[6]{name:"Michael Douglas"}

1 row

Use ON CREATE and ON MATCH

Merge with ON CREATE

Merge a node and set properties if the node needs to be created.

Query

```
MERGE (keanu:Person { name:'Keanu Reeves' })
ON CREATE SET keanu.created = timestamp()
RETURN keanu
```

Creates the Keanu node, and sets a timestamp on creation time.

Result

keanu
Node[7]{name:"Keanu Reeves",created:1428953961639}

1 row
Nodes created: 1
Properties set: 2
Labels added: 1

Merge with ON MATCH

Merging nodes and setting properties on found nodes.

Query

```
MERGE (person:Person)
ON MATCH SET person.found = TRUE RETURN person
```

Finds all the `Person` nodes, sets a property on them, and returns them.

Result

person
Node[0]{name:"Oliver Stone",found:true}
Node[1]{name:"Charlie Sheen",found:true}
Node[2]{name:"Martin Sheen",found:true}
Node[5]{name:"Rob Reiner",found:true}
Node[6]{name:"Michael Douglas",found:true}

5 rows
Properties set: 5

Merge with ON CREATE and ON MATCH

Merge a node and set properties if the node needs to be created.

Query

```
MERGE (keanu:Person { name:'Keanu Reeves' })
ON CREATE SET keanu.created = timestamp()
ON MATCH SET keanu.lastSeen = timestamp()
RETURN keanu
```

The query creates the Keanu node, and sets a timestamp on creation time. If Keanu already existed, a different property would have been set.

Result

keanu
Node[7]{name:"Keanu Reeves",created:1428953962716}

1 row
Nodes created: 1
Properties set: 2
Labels added: 1

Merge with ON MATCH setting multiple properties

If multiple properties should be set, simply separate them with commas.

Query

```
MERGE (person:Person)
ON MATCH SET person.found = TRUE , person.lastAccessed = timestamp()
RETURN person
```

Result

person
Node[0]{name:"Oliver Stone",found:true,lastAccessed:1428953962383}
Node[1]{name:"Charlie Sheen",found:true,lastAccessed:1428953962383}
Node[2]{name:"Martin Sheen",found:true,lastAccessed:1428953962383}
Node[5]{name:"Rob Reiner",found:true,lastAccessed:1428953962383}
Node[6]{name:"Michael Douglas",found:true,lastAccessed:1428953962383}

5 rows
Properties set: 10

Merge relationships

Merge on a relationship

`MERGE` can be used to match or create a relationship.

Query

```
MATCH (charlie:Person { name:'Charlie Sheen' }),(wallStreet:Movie { title:'Wall Street' })
MERGE (charlie)-[r:ACTED_IN]->(wallStreet)
RETURN r
```

Charlie Sheen had already been marked as acting on Wall Street, so the existing relationship is found and returned. Note that in order to match or create a relationship when using `MERGE`, at least one bound node must be specified, which is done via the `MATCH` clause in the above example.

Result

r
:ACTED_IN[0]{}

1 row

Merge on multiple relationships

When `MERGE` is used on a whole pattern, either everything matches, or everything is created.

Query

```
MATCH (oliver:Person { name:'Oliver Stone' }),(reiner:Person { name:'Rob Reiner' })
MERGE (oliver)-[:DIRECTED]->(movie:Movie)<-[:ACTED_IN]-(reiner)
RETURN movie
```

In our example graph, Oliver Stone and Rob Reiner have never worked together. When we try to `MERGE` a movie between them, Cypher will not use any of the existing movies already connected to either person. Instead, a new movie node is created.

Result

movie
Node[7]{}

1 row
Nodes created: 1
Relationships created: 2
Labels added: 1

Merge on an undirected relationship

`MERGE` can also be used with an undirected relationship. When it needs to create a new one, it will pick a direction.

Query

```
MATCH (charlie:Person { name:'Charlie Sheen' }),(oliver:Person { name:'Oliver Stone' })
MERGE (charlie)-[r:KNOWS]-(oliver)
RETURN r
```

Assume that Charlie Sheen and Oliver Stone do not know each other then this `MERGE` query will create a `:KNOWS` relationship between them. The direction of the created relationship is arbitrary.

Result

r
:KNOWS[8]{}

1 row
Relationships created: 1

Using unique constraints with MERGE

Cypher prevents getting conflicting results from `MERGE` when using patterns that involve uniqueness constrains. In this case, there must be at most one node that matches that pattern.

For example, given two uniqueness constraints on `:Person(id)` and `:Person(ssn)`: then a query such as `MERGE (n:Person {id: 12, ssn: 437})` will fail, if there are two different nodes (one with `id` 12 and one with `ssn` 437) or if there is only one node with only one of the properties. In other words, there must be exactly one node that matches the pattern, or no matching nodes.

Note that the following examples assume the existence of uniqueness constraints that have been created using:

```
CREATE CONSTRAINT ON (n:Person) ASSERT n.name IS UNIQUE;
CREATE CONSTRAINT ON (n:Person) ASSERT n.role IS UNIQUE;
```

Merge using unique constraints creates a new node if no node is found

Merge using unique constraints creates a new node if no node is found.

Query

```
MERGE (laurence:Person { name: 'Laurence Fishburne' })
RETURN laurence
```

The query creates the laurence node. If laurence already existed, merge would just return the existing node.

Result

laurence
Node[7]{name:"Laurence Fishburne"}

1 row
Nodes created: 1
Properties set: 1
Labels added: 1

Merge using unique constraints matches an existing node

Merge using unique constraints matches an existing node.

Query

```
MERGE (oliver:Person { name:'Oliver Stone' })
RETURN oliver
```

The oliver node already exists, so merge just returns it.

Result

oliver
Node[0]{name:"Oliver Stone"}

1 row

Merge with unique constraints and partial matches

Merge using unique constraints fails when finding partial matches.

Query

```
MERGE (michael:Person { name:'Michael Douglas', role:'Gordon Gekko' })
RETURN michael
```

While there is a matching unique michael node with the name *Michael Douglas*, there is no unique node with the role of *Gordon Gekko* and merge fails to match.

Error message

```
Merge did not find a matching node and can not create a new node due to conflicts
with both existing and missing unique nodes. The conflicting constraints are on:
:Person.name and :Person.role
```

Merge with unique constraints and conflicting matches

Merge using unique constraints fails when finding conflicting matches.

Query

```
MERGE (oliver:Person { name:'Oliver Stone', role:'Gordon Gekko' })
RETURN oliver
```

While there is a matching unique oliver node with the name *Oliver Stone*, there is also another unique node with the role of *Gordon Gekko* and merge fails to match.

Error message

```
Merge did not find a matching node and can not create a new node due to conflicts
with both existing and missing unique nodes. The conflicting constraints are on:
:Person.name and :Person.role
```

Using map parameters with MERGE

`MERGE` does not support map parameters like for example `CREATE` does. To use map parameters with `MERGE`, it is necessary to explicitly use the expected properties, like in the following example. For more information on parameters, see Section 8.5, "Parameters" [112].

Parameters

```
{
  "param" : {
    "name" : "Keanu Reeves",
    "role" : "Neo"
  }
}
```

Query

```
MERGE (oliver:Person { name: { param }.name, role: { param }.role })
RETURN oliver
```

Result

oliver
Node[7]{name:"Keanu Reeves",role:"Neo"}

1 row
Nodes created: 1
Properties set: 2
Labels added: 1

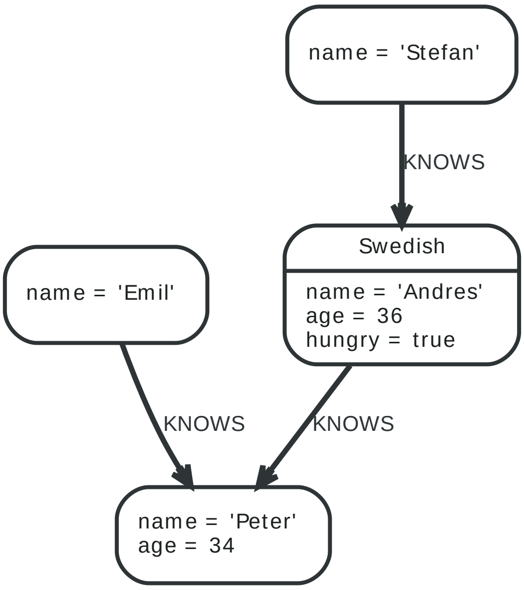
12.3. Set

The `SET` clause is used to update labels on nodes and properties on nodes and relationships.

`SET` can also be used with maps from parameters to set properties.

Note
Setting labels on a node is an idempotent operations — if you try to set a label on a node that already has that label on it, nothing happens. The query statistics will tell you if something needed to be done or not.

The examples use this graph as a starting point:



Set a property

To set a property on a node or relationship, use `SET`.

Query

```
MATCH (n { name: 'Andres' })
SET n.surname = 'Taylor'
RETURN n
```

The newly changed node is returned by the query.

Result

n
Node[3]{name:"Andres",age:36,hungry:true,surname:"Taylor"}

1 row
Properties set: 1

Remove a property

Normally you remove a property by using `REMOVE`, but it's sometimes handy to do it using the `SET` command. One example is if the property comes from a parameter.

Query

```
MATCH (n { name: 'Andres' })
```

```
SET n.name = NULL RETURN n
```

The node is returned by the query, and the name property is now missing.

Result

n
Node[3]{age:36,hungry:true}

1 row
Properties set: 1

Copying properties between nodes and relationships

You can also use `SET` to copy all properties from one graph element to another. Remember that doing this will remove all other properties on the receiving graph element.

Query

```
MATCH (at { name: 'Andres' }),(pn { name: 'Peter' })
SET at = pn
RETURN at, pn
```

The Andres node has had all it's properties replaced by the properties in the Peter node.

Result

at	pn
Node[3]{name:"Peter",age:34}	Node[2]{name:"Peter",age:34}

1 row
Properties set: 3

Adding properties from maps

When setting properties from a map (literal, paremeter, or graph element), you can use the `+=` form of `SET` to only add properties, and not remove any of the existing properties on the graph element.

Query

```
MATCH (peter { name: 'Peter' })
SET peter += { hungry: TRUE , position: 'Entrepreneur' }
```

Result

(empty result)

Properties set: 2

Set a property using a parameter

Use a parameter to give the value of a property.

Parameters

```
{
  "surname" : "Taylor"
}
```

Query

```
MATCH (n { name: 'Andres' })
SET n.surname = { surname }
RETURN n
```

The Andres node has got an surname added.

Result

n
Node[3]{name:"Andres",age:36,hungry:true,surname:"Taylor"}

1 row
Properties set: 1

Set all properties using a parameter

This will replace all existing properties on the node with the new set provided by the parameter.

Parameters

```
{
  "props" : {
    "name" : "Andres",
    "position" : "Developer"
  }
}
```

Query

```
MATCH (n { name: 'Andres' })
SET n = { props }
RETURN n
```

The Andres node has had all it's properties replaced by the properties in the `props` parameter.

Result

n
Node[3]{name:"Andres",position:"Developer"}

1 row
Properties set: 4

Set multiple properties using one SET clause

If you want to set multiple properties in one go, simply separate them with a comma.

Query

```
MATCH (n { name: 'Andres' })
SET n.position = 'Developer', n.surname = 'Taylor'
```

Result

(empty result)

Properties set: 2

Set a label on a node

To set a label on a node, use `SET`.

Query

```
MATCH (n { name: 'Stefan' })
SET n :German
RETURN n
```

The newly labeled node is returned by the query.

Result

n
Node[1]{name:"Stefan"}

1 row
Labels added: 1

Set multiple labels on a node

To set multiple labels on a node, use `SET` and separate the different labels using `:`.

Query

```
MATCH (n { name: 'Emil' })
SET n :Swedish:Bossman
RETURN n
```

The newly labeled node is returned by the query.

Result

n
Node[0]{name:"Emil"}

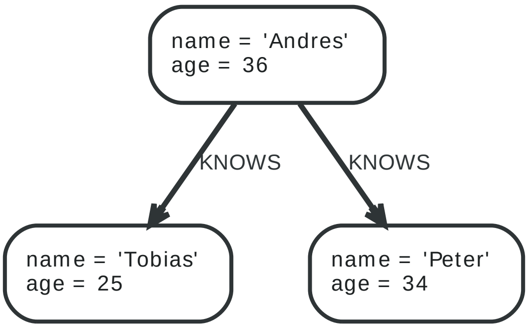
1 row
Labels added: 2

12.4. Delete

The `DELETE` clause is used to delete graph elements — nodes and relationships.

For removing properties and labels, see Section 12.5, "Remove" [197].

The examples start out with the following database:



Delete single node

To delete a node, use the `DELETE` clause.

Query

```
MATCH (n:Useless)
DELETE n
```

Nothing is returned from this query, except the count of affected nodes.

Result

```
(empty result)
```

Nodes deleted: 1

Delete a node and connected relationships

If you are trying to delete a node with relationships on it, you have to delete these as well.

Query

```
MATCH (n { name: 'Andres' })-[r]-()
DELETE n, r
```

Nothing is returned from this query, except the count of affected nodes.

Result

```
(empty result)
```

Nodes deleted: 1
Relationships deleted: 2

Delete all nodes and relationships

This query isn't for deleting large amounts of data, but is nice when playing around with small example data sets.

Query

```
MATCH (n)
OPTIONAL MATCH (n)-[r]-()
DELETE n,r
```

Nothing is returned from this query, except the count of affected nodes.

Result

```
(empty result)
```

Nodes deleted: 3
Relationships deleted: 2

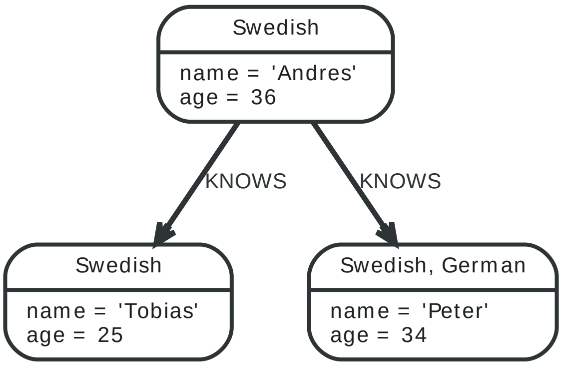
12.5. Remove

The `REMOVE` clause is used to remove properties and labels from graph elements.

For deleting nodes and relationships, see Section 12.4, “Delete” [195].

Note
Removing labels from a node is an idempotent operation: If you try to remove a label from a node that does not have that label on it, nothing happens. The query statistics will tell you if something needed to be done or not.

The examples start out with the following database:



Remove a property

Neo4j doesn’t allow storing `null` in properties. Instead, if no value exists, the property is just not there. So, to remove a property value on a node or a relationship, is also done with `REMOVE`.

Query

```
MATCH (andres { name: 'Andres' })
REMOVE andres.age
RETURN andres
```

The node is returned, and no property `age` exists on it.

Result

andres
Node[2]{name:"Andres"}

1 row
Properties set: 1

Remove a label from a node

To remove labels, you use `REMOVE`.

Query

```
MATCH (n { name: 'Peter' })
REMOVE n:German
RETURN n
```

Result

n
Node[1]{name:"Peter",age:34}

1 row
Labels removed: 1

Removing multiple labels

To remove multiple labels, you use `REMOVE`.

Query

```
MATCH (n { name: 'Peter' })
REMOVE n:German:Swedish
RETURN n
```

Result

n
Node[1]{name:"Peter",age:34}

1 row
Labels removed: 2

12.6. Foreach

The `FOREACH` clause is used to update data within a collection, whether components of a path, or result of aggregation.

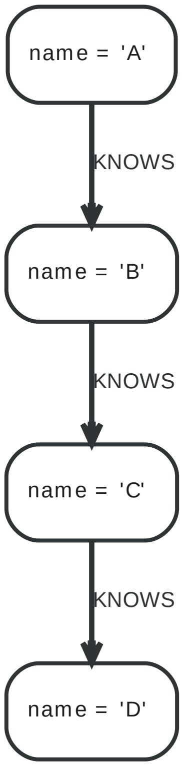
Collections and paths are key concepts in Cypher. To use them for updating data, you can use the `FOREACH` construct. It allows you to do updating commands on elements in a collection — a path, or a collection created by aggregation.

The identifier context inside of the foreach parenthesis is separate from the one outside it. This means that if you `CREATE` a node identifier inside of a `FOREACH`, you will *not* be able to use it outside of the foreach statement, unless you match to find it.

Inside of the `FOREACH` parentheses, you can do any of the updating commands — `CREATE`, `CREATE UNIQUE`, `MERGE`, `DELETE`, and `FOREACH`.

If you want to execute an additional `MATCH` for each element in a collection then `UNWIND` (see Section 10.6, “Unwind” [145]) would be a more appropriate command.

Figure 12.2. Data for the examples



Mark all nodes along a path

This query will set the property `marked` to true on all nodes along a path.

Query

```
MATCH p =(begin)-[*]->(END )
WHERE begin.name='A' AND END .name='D'
FOREACH (n IN nodes(p)| SET n.marked = TRUE )
```

Nothing is returned from this query, but four properties are set.

Result

```
(empty result)
```

Properties set: 4

12.7. Create Unique

The `CREATE UNIQUE` clause is a mix of `MATCH` and `CREATE` — it will match what it can, and create what is missing.

Introduction

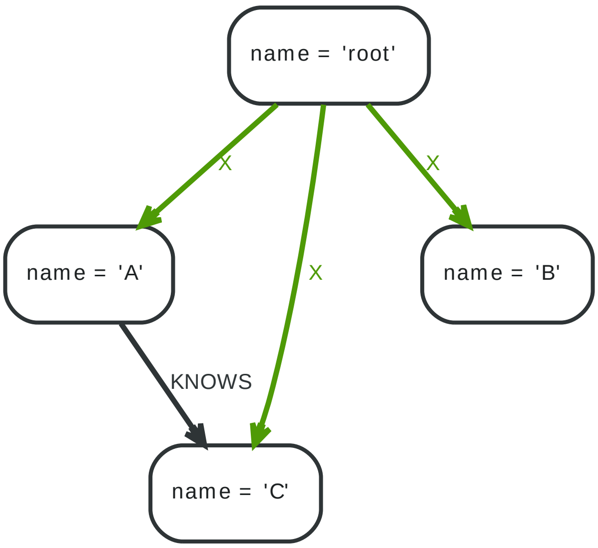
Tip
`MERGE` might be what you want to use instead of `CREATE UNIQUE`. Note however, that `MERGE` doesn't give as strong guarantees for relationships being unique.

`CREATE UNIQUE` is in the middle of `MATCH` and `CREATE` — it will match what it can, and create what is missing. `CREATE UNIQUE` will always make the least change possible to the graph — if it can use parts of the existing graph, it will.

Another difference to `MATCH` is that `CREATE UNIQUE` assumes the pattern to be unique. If multiple matching subgraphs are found an error will be generated.

Tip
In the `CREATE UNIQUE` clause, patterns are used a lot. Read Section 9.6, "Patterns" [125] for an introduction.

The examples start out with the following data set:



Create unique nodes

Create node if missing

If the pattern described needs a node, and it can't be matched, a new node will be created.

Query

```
MATCH (root { name: 'root' })
CREATE UNIQUE (root)-[:LOVES]-(someone)
RETURN someone
```

The root node doesn't have any `LOVES` relationships, and so a node is created, and also a relationship to that node.

Result

someone
Node[4]{}

1 row
Nodes created: 1
Relationships created: 1

Create nodes with values

The pattern described can also contain values on the node. These are given using the following syntax: `prop : <expression>`.

Query

```
MATCH (root { name: 'root' })
CREATE UNIQUE (root)-[:X]-(leaf { name:'D' })
RETURN leaf
```

No node connected with the root node has the name `D`, and so a new node is created to match the pattern.

Result

leaf
Node[4]{name:"D"}

1 row
Nodes created: 1
Relationships created: 1
Properties set: 1

Create labeled node if missing

If the pattern described needs a labeled node and there is none with the given labels, Cypher will create a new one.

Query

```
MATCH (a { name: 'A' })
CREATE UNIQUE (a)-[:KNOWS]-(c:blue)
RETURN c
```

The A node is connected in a `KNOWS` relationship to the c node, but since C doesn't have the `:blue` label, a new node labeled as `:blue` is created along with a `KNOWS` relationship from A to it.

Result

c
Node[4]{}

1 row
Nodes created: 1
Relationships created: 1
Labels added: 1

Create unique relationships

Create relationship if it is missing

`CREATE UNIQUE` is used to describe the pattern that should be found or created.

Query

```
MATCH (lft { name: 'A' }),(rgt)
WHERE rgt.name IN ['B', 'C']
CREATE UNIQUE (lft)-[r:KNOWS]->(rgt)
RETURN r
```

The left node is matched agains the two right nodes. One relationship already exists and can be matched, and the other relationship is created before it is returned.

Result

r
:KNOWS[4]{}
:KNOWS[3]{}

2 rows
Relationships created: 1

Create relationship with values

Relationships to be created can also be matched on values.

Query

```
MATCH (root { name: 'root' })
CREATE UNIQUE (root)-[r:X { since:'forever' }]-()
RETURN r
```

In this example, we want the relationship to have a value, and since no such relationship can be found, a new node and relationship are created. Note that since we are not interested in the created node, we don't name it.

Result

r
:X[4]{since:"forever"}

1 row
Nodes created: 1
Relationships created: 1
Properties set: 1

Describe complex pattern

The pattern described by `CREATE UNIQUE` can be separated by commas, just like in `MATCH` and `CREATE`.

Query

```
MATCH (root { name: 'root' })
CREATE UNIQUE (root)-[:FOO]->(x),(root)-[:BAR]->(x)
RETURN x
```

This example pattern uses two paths, separated by a comma.

Result

x
Node[4]{}

1 row
Nodes created: 1
Relationships created: 2

12.8. Importing CSV files with Cypher

This tutorial will show you how to import data from CSV files using `LOAD CSV`.

In this example, we're given three CSV files: a list of persons, a list of movies, and a list of which role was played by some of these persons in each movie.

CSV files can be stored on the database server and are then accessible using a `file://` URL. Alternatively, `LOAD CSV` also supports accessing CSV files via `HTTPS`, `HTTP`, and `FTP`.

Using the following Cypher queries, we'll create a node for each person, a node for each movie and a relationship between the two with a property denoting the role. We're also keeping track of the country in which each movie was made.

Let's start with importing the persons:

```
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.2.1/csv/import/persons.csv" AS csvLine
CREATE (p:Person { id: toInt(csvLine.id), name: csvLine.name })
```

The CSV file we're using looks like this:

persons.csv

```
id,name
1,Charlie Sheen
2,Oliver Stone
3,Michael Douglas
4,Martin Sheen
5,Morgan Freeman
```

Now, let's import the movies. This time, we're also creating a relationship to the country in which the movie was made. If you are storing your data in a SQL database, this is the one-to-many relationship type.

We're using `MERGE` to create nodes that represent countries. Using `MERGE` avoids creating duplicate country nodes in the case where multiple movies have been made in the same country.

Important
When using `MERGE` or `MATCH` with `LOAD CSV` we need to make sure we have an index (see Section 14.1, "Indexes" [236]) or a unique constraint (see Section 14.2, "Constraints" [238]) on the property we're merging. This will ensure the query executes in a performant way.

Before running our query to connect movies and countries we'll create an index for the name property on the Country label to ensure the query runs as fast as it can:

```
CREATE INDEX ON :Country(name)
```

```
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.2.1/csv/import/movies.csv" AS csvLine
MERGE (country:Country { name: csvLine.country })
CREATE (movie:Movie { id: toInt(csvLine.id), title: csvLine.title, year:toInt(csvLine.year)})
CREATE (movie)-[:MADE_IN]->(country)
```

movies.csv

```
id,title,country,year
1,Wall Street,USA,1987
2,The American President,USA,1995
3,The Shawshank Redemption,USA,1994
```

Lastly, we create the relationships between the persons and the movies. Since the relationship is a many to many relationship, one actor can participate in many movies, and one movie has many actors in it. We have this data in a separate file.

We'll index the id property on Person and Movie nodes. The id property is a temporary property used to look up the appropriate nodes for a relationship when importing the third file. By indexing the id property, node lookup (e.g. by `MATCH`) will be much faster. Since we expect the ids to be unique in each set, we'll create a unique constraint. This protects us from invalid data since constraint creation will fail if there are multiple nodes with the same id property. Creating a unique constraint also creates a unique index (which is faster than a regular index).

```
CREATE CONSTRAINT ON (person:Person) ASSERT person.id IS UNIQUE
```

```
CREATE CONSTRAINT ON (movie:Movie) ASSERT movie.id IS UNIQUE
```

Now importing the relationships is a matter of finding the nodes and then creating relationships between them.

For this query we'll use `USING PERIODIC COMMIT` (see Section 12.9, "Using Periodic Commit" [205]) which is helpful for queries that operate on large CSV files. This hint tells Neo4j that the query might build up inordinate amounts of transaction state, and so needs to be periodically committed. In this case we also set the limit to `500` rows per commit.

```
USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.2.1/csv/import/roles.csv" AS csvLine
MATCH (person:Person { id: toInt(csvLine.personId)}),(movie:Movie { id: toInt(csvLine.movieId)})
CREATE (person)-[:PLAYED { role: csvLine.role }]->(movie)
```

roles.csv

```
personId,movieId,role
1,1,Bud Fox
4,1,Carl Fox
3,1,Gordon Gekko
4,2,A.J. MacInerney
3,2,President Andrew Shepherd
5,3,Ellis Boyd 'Red' Redding
```

Finally, as the `id` property was only necessary to import the relationships, we can drop the constraints and the id property from all movie and person nodes.

```
DROP CONSTRAINT ON (person:Person) ASSERT person.id IS UNIQUE
```

```
DROP CONSTRAINT ON (movie:Movie) ASSERT movie.id IS UNIQUE
```

```
MATCH (n)
WHERE n:Person OR n:Movie
REMOVE n.id
```

12.9. Using Periodic Commit

Note
See Section 12.8, "Importing CSV files with Cypher" [203] on how to import data from CSV files.

Updating very large amounts of data (for example when importing) with a single Cypher query may fail due to memory constraints. This will manifest itself as an `OutOfMemoryError`.

For this situation *only,* Cypher provides the global `USING PERIODIC COMMIT` query hint for updating queries. You can optionally set the limit for the number of rows per commit like so: `USING PERIODIC COMMIT 500`.

Periodic Commit will process the rows until the number of rows reaches a limit. Then the current transaction will be committed and replaced with a newly opened transaction. If no limit is set, a default value will be used.

See the section called "Importing large amounts of data" [178] in Section 11.6, "Load CSV" [177] for examples of `USING PERIODIC COMMIT` with and without setting the number of rows per commit.

Important
Using periodic commit will prevent running out of memory when updating large amounts of data. However it will also break transactional isolation thus it should only be used where needed.

Chapter 13. Functions

This chapter contains information on all functions in Cypher. Note that related information exists in Section 9.4, “Operators” [122].

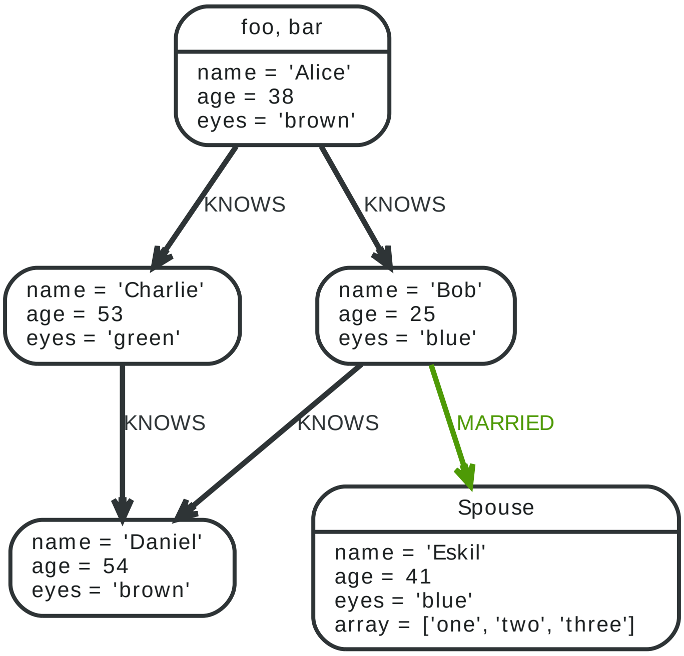
Note
Most functions in Cypher will return `NULL` if an input parameter is `NULL`.

13.1. 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.

See also the section called “Comparison operators” [122].

Figure 13.1. Graph



ALL

Tests whether a predicate holds for all element of this collection collection.

Syntax: `ALL(identifier in collection WHERE predicate)`

Arguments:

- *collection:* An expression that returns a collection
- *identifier:* This is the identifier that can be used from the predicate.
- *predicate:* A predicate that is tested against all items in the collection.

Query

```
MATCH p=(a)-[*1..3]->(b)
WHERE a.name='Alice' AND b.name='Daniel' AND ALL (x IN nodes(p) WHERE x.age > 30)
RETURN p
```

All nodes in the returned paths will have an `age` property of at least 30.

Result

p
[Node[2]{name:"Alice",age:38,eyes:"brown"},:KNOWS[1]{},Node[4]{name:"Charlie",age:53,eyes:"green"},:KNOWS[3]{},Node[0]{name:"Daniel",age:54,eyes:"brown"}]
1 row

ANY

Tests whether a predicate holds for at least one element in the collection.

Syntax: `ANY(identifier in collection WHERE predicate)`

Arguments:

- *collection:* An expression that returns a collection
- *identifier:* This is the identifier that can be used from the predicate.
- *predicate:* A predicate that is tested against all items in the collection.

Query

```
MATCH (a)
WHERE a.name='Eskil' AND ANY (x IN a.array WHERE x = "one")
RETURN a
```

All nodes in the returned paths has at least one `one` value set in the array property named `array`.

Result

a
Node[1]{name:"Eskil",age:41,eyes:"blue",array:["one","two","three"]}

1 row

NONE

Returns true if the predicate holds for no element in the collection.

Syntax: `NONE(identifier in collection WHERE predicate)`

Arguments:

- *collection:* An expression that returns a collection
- *identifier:* This is the identifier that can be used from the predicate.
- *predicate:* A predicate that is tested against all items in the collection.

Query

```
MATCH p=(n)-[*1..3]->(b)
WHERE n.name='Alice' AND NONE (x IN nodes(p) WHERE x.age = 25)
RETURN p
```

No nodes in the returned paths has a `age` property set to `25`.

Result

p
[Node[2]{name:"Alice",age:38,eyes:"brown"},:KNOWS[1]{},Node[4]{name:"Charlie",age:53,eyes:"green"}]
[Node[2]{name:"Alice",age:38,eyes:"brown"},:KNOWS[1]{},Node[4]{name:"Charlie",age:53, eyes:"green"},:KNOWS[3]{},Node[0]{name:"Daniel",age:54,eyes:"brown"}]

2 rows

SINGLE

Returns true if the predicate holds for exactly one of the elements in the collection.

Syntax: `SINGLE(identifier in collection WHERE predicate)`

Arguments:

- *collection:* An expression that returns a collection
- *identifier:* This is the identifier that can be used from the predicate.
- *predicate:* A predicate that is tested against all items in the collection.

Query

```
MATCH p=(n)-->(b)
WHERE n.name='Alice' AND SINGLE (var IN nodes(p) WHERE var.eyes = "blue")
RETURN p
```

Exactly one node in every returned path will have the `eyes` property set to `"blue"`.

Result

p
[Node[2]{name:"Alice",age:38,eyes:"brown"},:KNOWS[0]{},Node[3]{name:"Bob",age:25,eyes:"blue"}]

1 row

EXISTS

Returns true if a match for the pattern exists in the graph, or the property exists in the node, relationship or map.

Syntax: `EXISTS( pattern-or-property )`

Arguments:

- *pattern-or-property:* A pattern or a property (in the form *identifier.prop*).

Query

```
MATCH (n)
WHERE EXISTS(n.name)
RETURN n.name AS name, EXISTS((n)-[:MARRIED]->()) AS is_married
```

This query returns all the nodes with a name property along with a boolean true/false indicating if they are married.

Result

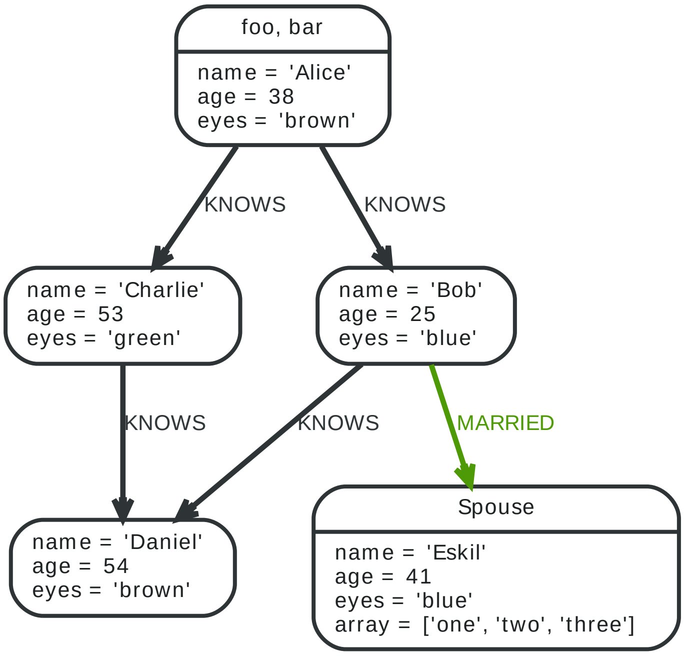
name	is_married
"Daniel"	false
"Eskil"	false
"Alice"	false
"Bob"	true
"Charlie"	false

5 rows

13.2. Scalar functions

Scalar functions return a single value.

Figure 13.2. Graph



LENGTH

To return or filter on the length of a collection, use the `LENGTH()` function.

Syntax: `LENGTH( collection )`

Arguments:

- *collection:* An expression that returns a collection

Query

```
MATCH p=(a)-->(b)-->(c)
WHERE a.name='Alice'
RETURN length(p)
```

The length of the path `p` is returned by the query.

Result

length(p)
2
2
2
3 rows

LENGTH of pattern expression

This is the same `LENGTH()` method described before, but instead of passing in a collection directly, you provide a pattern expression that can be used in a match query to provide a new set of results. The length of the results is calculated, not the length of the expression itself.

Syntax: `LENGTH( pattern expression )`

Arguments:

- *pattern expression:* A pattern expression that returns a collection

Query

```
MATCH (a)
WHERE a.name='Alice'
RETURN length((a)-->()-->()) AS fof
```

The number of sub-graphs matching the pattern expression is returned by the query.

Result

fof
3

1 row

TYPE

Returns a string representation of the relationship type.

Syntax: `TYPE( relationship )`

Arguments:

- *relationship:* A relationship.

Query

```
MATCH (n)-[r]->()
WHERE n.name='Alice'
RETURN type(r)
```

The relationship type of `r` is returned by the query.

Result

type(r)
"KNOWS"
"KNOWS"

2 rows

ID

Returns the id of the relationship or node.

Syntax: `ID( property-container )`

Arguments:

- *property-container:* A node or a relationship.

Query

```
MATCH (a)
RETURN id(a)
```

This returns the node id for three nodes.

Result

id(a)
0

5 rows

id(a)
1
2
3
4

5 rows

COALESCE

Returns the first non-`NULL` value in the list of expressions passed to it. In case all arguments are `NULL`, `NULL` will be returned.

Syntax: `COALESCE( expression [, expression]* )`

Arguments:

- *expression:* The expression that might return NULL.

Query

```
MATCH (a)
WHERE a.name='Alice'
RETURN coalesce(a.hairColor, a.eyes)
```

Result

coalesce(a.hairColor, a.eyes)
"brown"

1 row

HEAD

`HEAD` returns the first element in a collection.

Syntax: `HEAD( expression )`

Arguments:

- *expression:* This expression should return a collection of some kind.

Query

```
MATCH (a)
WHERE a.name='Eskil'
RETURN a.array, head(a.array)
```

The first node in the path is returned.

Result

a.array	head(a.array)
["one","two","three"]	"one"

1 row

LAST

`LAST` returns the last element in a collection.

Syntax: `LAST( expression )`

Arguments:

- *expression:* This expression should return a collection of some kind.

Query

```
MATCH (a)
WHERE a.name='Eskil'
RETURN a.array, last(a.array)
```

The last node in the path is returned.

Result

a.array	last(a.array)
["one","two","three"]	"three"

1 row

TIMESTAMP

`TIMESTAMP` returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC. It will return the same value during the whole one query, even if the query is a long running one.

Syntax: `TIMESTAMP()`

Arguments:

Query

```
RETURN timestamp()
```

The time in milliseconds is returned.

Result

timestamp()
1428953941657

1 row

STARTNODE

`STARTNODE` returns the starting node of a relationship

Syntax: `STARTNODE( relationship )`

Arguments:

- *relationship:* An expression that returns a relationship

Query

```
MATCH (x:foo)-[r]-()
RETURN startNode(r)
```

Result

startNode(r)
Node[2]{name:"Alice",age:38,eyes:"brown"}
Node[2]{name:"Alice",age:38,eyes:"brown"}

2 rows

ENDNODE

`ENDNODE` returns the end node of a relationship

Syntax: `ENDNODE( relationship )`

Arguments:

- *relationship:* An expression that returns a relationship

Query

```
MATCH (x:foo)-[r]-()
RETURN endNode(r)
```

Result

endNode(r)
Node[4]{name:"Charlie",age:53,eyes:"green"}
Node[3]{name:"Bob",age:25,eyes:"blue"}

2 rows

TOINT

`TOINT` converts the argument to an integer. A string is parsed as if it was an integer number. If the parsing fails, `NULL` will be returned. A floating point number will be cast into an integer.

Syntax: `TOINT( expression )`

Arguments:

- *expression:* An expression that returns anything

Query

```
RETURN toInt("42"), toInt("not a number")
```

Result

toInt("42")	toInt("not a number")
42	<null>

1 row

TOFLOAT

`TOFLOAT` converts the argument to a float. A string is parsed as if it was an floating point number. If the parsing fails, `NULL` will be returned. An integer will be cast to a floating point number.

Syntax: `TOFLOAT( expression )`

Arguments:

- *expression:* An expression that returns anything

Query

```
RETURN toFloat("11.5"), toFloat("not a number")
```

Result

toFloat("11.5")	toFloat("not a number")
11.5	<null>

1 row

TOSTRING

`TOSTRING` converts the argument to a string. It converts integers and floating point numbers to strings, and if called with a string will leave it unchanged.

Syntax: `TOSTRING( expression )`

Arguments:

- *expression:* An expression that returns anything

Query

```
RETURN toString(11.5), toString("already a string")
```

Result

toString(11.5)	toString("already a string")
"11.5"	"already a string"

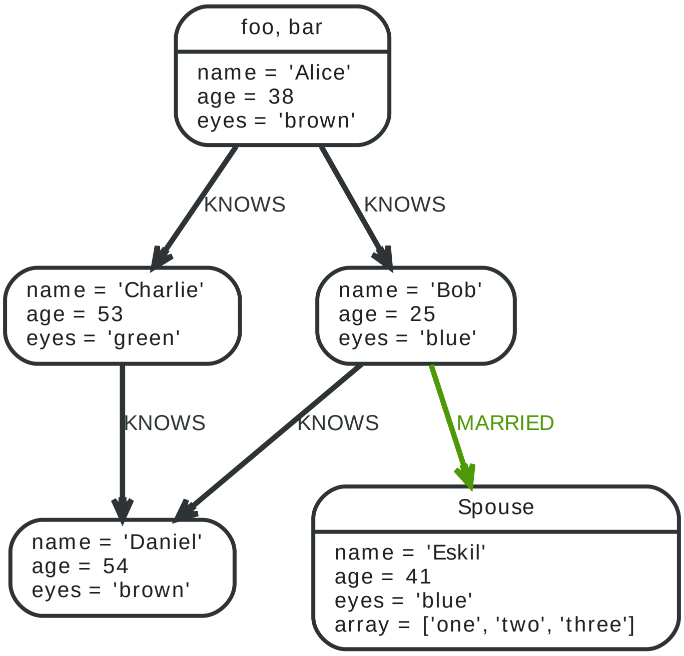
1 row

13.3. Collection functions

Collection functions return collections of things — nodes in a path, and so on.

See also the section called “Collection operators” [122].

Figure 13.3. Graph



NODES

Returns all nodes in a path.

Syntax: `NODES( path )`

Arguments:

- *path:* A path.

Query

```
MATCH p=(a)-->(b)-->(c)
WHERE a.name='Alice' AND c.name='Eskil'
RETURN nodes(p)
```

All the nodes in the path `p` are returned by the example query.

Result

nodes(p)
[Node[2]{name:"Alice",age:38,eyes:"brown"},Node[3]{name:"Bob",age:25,eyes:"blue"},Node[1]{name:"Eskil",age:41,eyes:"blue",array:["one","two","three"]}]
1 row

RELATIONSHIPS

Returns all relationships in a path.

Syntax: `RELATIONSHIPS( path )`

Arguments:

- *path:* A path.

Query

```
MATCH p=(a)-->(b)-->(c)
WHERE a.name='Alice' AND c.name='Eskil'
RETURN relationships(p)
```

All the relationships in the path `p` are returned.

Result

relationships(p)
[:KNOWS[0]{},:MARRIED[4]{}]

1 row

LABELS

Returns a collection of string representations for the labels attached to a node.

Syntax: `LABELS( node )`

Arguments:

- *node:* Any expression that returns a single node

Query

```
MATCH (a)
WHERE a.name='Alice'
RETURN labels(a)
```

The labels of `n` is returned by the query.

Result

labels(a)
["foo","bar"]

1 row

KEYS

Returns a collection of string representations for the property names of a node, relationship, or map.

Syntax: `KEYS( property-container )`

Arguments:

- *property-container:* A node, a relationship, or a literal map.

Query

```
MATCH (a)
WHERE a.name='Alice'
RETURN keys(a)
```

The name of the properties of `n` is returned by the query.

Result

keys(a)
["name","age","eyes"]

1 row

EXTRACT

To return a single property, or the value of a function from a collection of nodes or relationships, you can use `EXTRACT`. It will go through a collection, run an expression on every element, and return the results in an collection with these values. It works like the `map` method in functional languages such as Lisp and Scala.

Syntax: `EXTRACT( identifier in collection | expression )`

Arguments:

- *collection:* An expression that returns a collection
- *identifier:* The closure will have an identifier introduced in it's context. Here you decide which identifier to use.
- *expression:* This expression will run once per value in the collection, and produces the result collection.

Query

```
MATCH p=(a)-->(b)-->(c)
WHERE a.name='Alice' AND b.name='Bob' AND c.name='Daniel'
RETURN extract(n IN nodes(p)| n.age) AS extracted
```

The age property of all nodes in the path are returned.

Result

extracted
[38,25,54]

1 row

FILTER

`FILTER` returns all the elements in a collection that comply to a predicate.

Syntax: `FILTER(identifier in collection WHERE predicate)`

Arguments:

- *collection:* An expression that returns a collection
- *identifier:* This is the identifier that can be used from the predicate.
- *predicate:* A predicate that is tested against all items in the collection.

Query

```
MATCH (a)
WHERE a.name='Eskil'
RETURN a.array, filter(x IN a.array WHERE length(x)= 3)
```

This returns the property named `array` and a list of values in it, which have the length `3`.

Result

a.array	filter(x in a.array WHERE length(x) = 3)
["one","two","three"]	["one","two"]

1 row

TAIL

`TAIL` returns all but the first element in a collection.

Syntax: `TAIL( expression )`

Arguments:

- *expression:* This expression should return a collection of some kind.

Query

```
MATCH (a)
WHERE a.name='Eskil'
RETURN a.array, tail(a.array)
```

This returns the property named `array` and all elements of that property except the first one.

Result

a.array	tail(a.array)
["one","two","three"]	["two","three"]

1 row

RANGE

Returns numerical values in a range with a non-zero step value step. Range is inclusive in both ends.

Syntax: `RANGE( start, end [, step] )`

Arguments:

- *start:* A numerical expression.
- *end:* A numerical expression.
- *step:* A numerical expression.

Query

```
RETURN range(0,10), range(2,18,3)
```

Two lists of numbers are returned.

Result

range(0,10)	range(2,18,3)
[0,1,2,3,4,5,6,7,8,9,10]	[2,5,8,11,14,17]

1 row

REDUCE

To run an expression against individual elements of a collection, and store the result of the expression in an accumulator, you can use `REDUCE`. It will go through a collection, run an expression on every element, storing the partial result in the accumulator. It works like the `fold` or `reduce` method in functional languages such as Lisp and Scala.

Syntax: `REDUCE( accumulator = initial, identifier in collection | expression )`

Arguments:

- *accumulator:* An identifier that will hold the result and the partial results as the collection is iterated
- *initial:* An expression that runs once to give a starting value to the accumulator
- *collection:* An expression that returns a collection
- *identifier:* The closure will have an identifier introduced in it's context. Here you decide which identifier to use.
- *expression:* This expression will run once per value in the collection, and produces the result value.

Query

```
MATCH p=(a)-->(b)-->(c)
WHERE a.name='Alice' AND b.name='Bob' AND c.name='Daniel'
RETURN reduce(totalAge = 0, n IN nodes(p)| totalAge + n.age) AS reduction
```

The age property of all nodes in the path are summed and returned as a single value.

Result

reduction
117

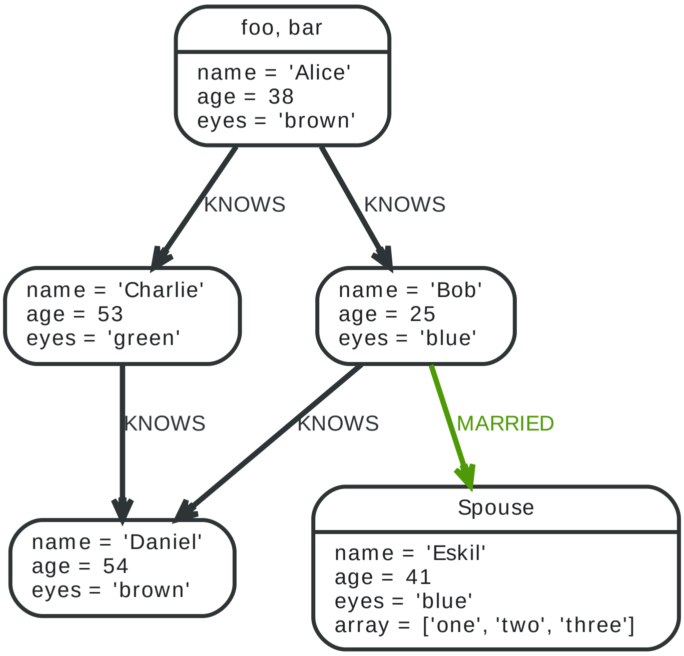
1 row

13.4. Mathematical functions

These functions all operate on numerical expressions only, and will return an error if used on any other values.

See also the section called “Mathematical operators” [122].

Figure 13.4. Graph



ABS

`ABS` returns the absolute value of a number.

Syntax: `ABS( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
MATCH (a),(e)
WHERE a.name = 'Alice' AND e.name = 'Eskil'
RETURN a.age, e.age, abs(a.age - e.age)
```

The absolute value of the age difference is returned.

Result

a.age	e.age	abs(a.age - e.age)
38	41	3.0

1 row

ACOS

`ACOS` returns the arccosine of the expression, in radians.

Syntax: `ACOS( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN acos(0.5)
```

The arccosine of 0.5.

Result

acos(0.5)
1.0471975511965979

1 row

ASIN

`ASIN` returns the arcsine of the expression, in radians.

Syntax: `ASIN( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN asin(0.5)
```

The arcsine of 0.5.

Result

asin(0.5)
0.5235987755982989

1 row

ATAN

`ATAN` returns the arctangent of the expression, in radians.

Syntax: `ATAN( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN atan(0.5)
```

The arctangent of 0.5.

Result

atan(0.5)
0.4636476090008061

1 row

ATAN2

`ATAN2` returns the arctangent2 of a set of coordinates, in radians.

Syntax: `ATAN2( expression , expression)`

Arguments:

- *expression:* A numeric expression for y.

- *expression:* A numeric expression for x.

Query

```
RETURN atan2(0.5, 0.6)
```

The arctangent2 of 0.5, 0.6.

Result

atan2(0.5, 0.6)
0.6947382761967033
1 row

COS

`COS` returns the cosine of the expression.

Syntax: `COS( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN cos(0.5)
```

The cosine of 0.5 is returned.

Result

cos(0.5)
0.8775825618903728
1 row

COT

`COT` returns the cotangent of the expression.

Syntax: `COT( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN cot(0.5)
```

The cotangent of 0.5 is returned.

Result

cot(0.5)
1.830487721712452
1 row

DEGREES

`DEGREES` converts radians to degrees.

Syntax: `DEGREES( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN degrees(3.14159)
```

The number of degrees in something close to pi.

Result

degrees(3.14159)
179.99984796050427

1 row

E

`E` returns the constant, e.

Syntax: `E()`

Arguments:

Query

```
RETURN e()
```

The constant e is returned (the base of natural log).

Result

e()
2.718281828459045

1 row

EXP

`EXP` returns the value e raised to the power of the expression.

Syntax: `EXP( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN exp(2)
```

The exp of 2 is returned: e^2.

Result

exp(2)
7.38905609893065

1 row

FLOOR

`FLOOR` returns the greatest integer less than or equal to the expression.

Syntax: `FLOOR( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN floor(0.9)
```

The floor of 0.9 is returned.

Result

floor(0.9)
0.0

1 row

HAVERSIN

`HAVERSIN` returns half the versine of the expression.

Syntax: `HAVERSIN( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN haversin(0.5)
```

The haversine of 0.5 is returned.

Result

haversin(0.5)
0.06120871905481362

1 row

Spherical distance using the haversin function

The `haversin` function may be used to compute the distance on the surface of a sphere between two points (each given by their latitude and longitude). In this example the spherical distance (in km) between Berlin in Germany (at lat 52.5, lon 13.4) and San Mateo in California (at lat 37.5, lon -122.3) is calculated using an average earth radius of 6371 km.

Query

```
CREATE (ber:City { lat: 52.5, lon: 13.4 }),(sm:City { lat: 37.5, lon: -122.3 })
RETURN 2 * 6371 * asin(sqrt(haversin(radians(sm.lat - ber.lat))+ cos(radians(sm.lat))*
  cos(radians(ber.lat))* haversin(radians(sm.lon - ber.lon)))) AS dist
```

The distance between Berlin and San Mateo is returned (about 9129 km).

Result

dist
9129.969740051658

1 row
Nodes created: 2
Properties set: 4
Labels added: 2

LOG

`LOG` returns the natural logarithm of the expression.

Syntax: `LOG( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN log(27)
```

The log of 27 is returned.

Result

log(27)
3.295836866004329

1 row

LOG10

`LOG10` returns the base 10 logarithm of the expression.

Syntax: `LOG10( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN log10(27)
```

The log10 of 27 is returned.

Result

log10(27)
1.4313637641589874

1 row

PI

`PI` returns the mathematical constant pi.

Syntax: `PI()`

Arguments:

Query

```
RETURN pi()
```

The constant pi is returned.

Result

pi()
3.141592653589793

1 row

RADIANS

`RADIANS` converts degrees to radians.

Syntax: `RADIANS( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN radians(180)
```

The number of radians in 180 is returned (pi).

Result

radians(180)
3.141592653589793
1 row

RAND

`RAND` returns a random double between 0 and 1.0.

Syntax: `RAND( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN rand() AS x1
```

A random number is returned.

Result

x1
0.19400145612766406
1 row

ROUND

`ROUND` returns the numerical expression, rounded to the nearest integer.

Syntax: `ROUND( expression )`

Arguments:

- *expression:* A numerical expression.

Query

```
RETURN round(3.141592)
```

Result

round(3.141592)
3.0
1 row

SIGN

`SIGN` returns the signum of a number — zero if the expression is zero, `-1` for any negative number, and `1` for any positive number.

Syntax: `SIGN( expression )`

Arguments:

- *expression:* A numerical expression

Query

```
RETURN sign(-17), sign(0.1)
```

Result

sign(-17)	sign(0.1)
-1.0	1.0

1 row

SIN

`SIN` returns the sine of the expression.

Syntax: `SIN( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN sin(0.5)
```

The sine of 0.5 is returned.

Result

sin(0.5)
0.479425538604203

1 row

SQRT

`SQRT` returns the square root of a number.

Syntax: `SQRT( expression )`

Arguments:

- *expression:* A numerical expression

Query

```
RETURN sqrt(256)
```

Result

sqrt(256)
16.0

1 row

TAN

`TAN` returns the tangent of the expression.

Syntax: `TAN( expression )`

Arguments:

- *expression:* A numeric expression.

Query

```
RETURN tan(0.5)
```

The tangent of 0.5 is returned.

Result

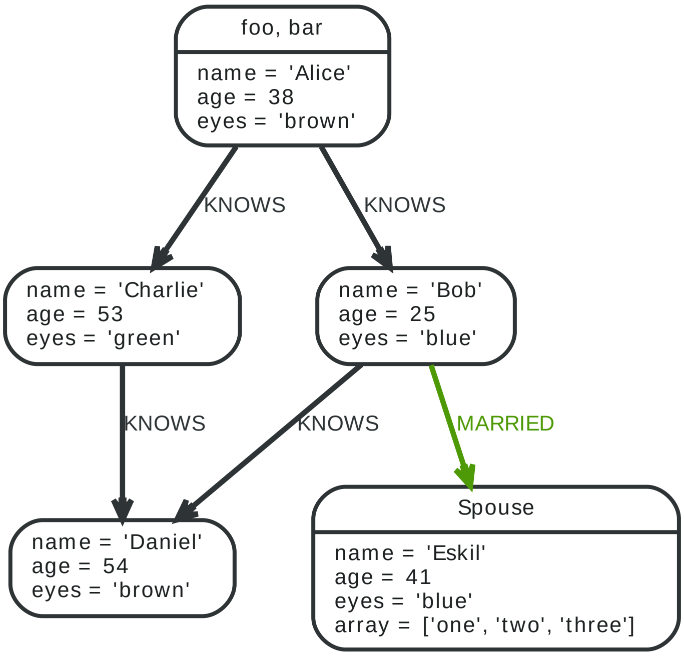
tan(0.5)
0.5463024898437905
1 row

13.5. String functions

These functions all operate on string expressions only, and will return an error if used on any other values. Except `STR()`, which converts to strings.

See also the section called "String operators" [122].

Figure 13.5. Graph



STR

`STR` returns a string representation of the expression.

Syntax: `STR( expression )`

Arguments:

- *expression:* An expression that returns anything

Query

```
RETURN str(1)
```

Result

str(1)
"1"
1 row

REPLACE

`REPLACE` returns a string with the search string replaced by the replace string. It replaces all occurrences.

Syntax: `REPLACE( original, search, replace )`

Arguments:

- *original:* An expression that returns a string
- *search:* An expression that returns a string to search for
- *replace:* An expression that returns the string to replace the search string with

Query

```
RETURN replace("hello", "l", "w")
```

Result

replace("hello", "l", "w")
"hewwo"

1 row

SUBSTRING

`SUBSTRING` returns a substring of the original, with a 0-based index start and length. If length is omitted, it returns a substring from start until the end of the string.

Syntax: `SUBSTRING( original, start [, length] )`

Arguments:

- *original:* An expression that returns a string
- *start:* An expression that returns a positive number
- *length:* An expression that returns a positive number

Query

```
RETURN substring("hello", 1, 3), substring("hello", 2)
```

Result

substring("hello", 1, 3)	substring("hello", 2)
"ell"	"llo"

1 row

LEFT

`LEFT` returns a string containing the left n characters of the original string.

Syntax: `LEFT( original, length )`

Arguments:

- *original:* An expression that returns a string
- *n:* An expression that returns a positive number

Query

```
RETURN left("hello", 3)
```

Result

left("hello", 3)
"hel"

1 row

RIGHT

`RIGHT` returns a string containing the right n characters of the original string.

Syntax: `RIGHT( original, length )`

Arguments:

- *original:* An expression that returns a string

- *n:* An expression that returns a positive number

Query

```
RETURN right("hello", 3)
```

Result

right("hello", 3)
"llo"

1 row

LTRIM

`LTRIM` returns the original string with whitespace removed from the left side.

Syntax: `LTRIM( original )`

Arguments:

- *original:* An expression that returns a string

Query

```
RETURN ltrim("   hello")
```

Result

ltrim(" hello")
"hello"

1 row

RTRIM

`RTRIM` returns the original string with whitespace removed from the right side.

Syntax: `RTRIM( original )`

Arguments:

- *original:* An expression that returns a string

Query

```
RETURN rtrim("hello   ")
```

Result

rtrim("hello ")
"hello"

1 row

TRIM

`TRIM` returns the original string with whitespace removed from both sides.

Syntax: `TRIM( original )`

Arguments:

- *original:* An expression that returns a string

Query

```
RETURN trim("   hello   ")
```

Result

trim(" hello ")
"hello"

1 row

LOWER

`LOWER` returns the original string in lowercase.

Syntax: `LOWER( original )`

Arguments:

- *original:* An expression that returns a string

Query

```
RETURN lower("HELLO")
```

Result

lower("HELLO")
"hello"

1 row

UPPER

`UPPER` returns the original string in uppercase.

Syntax: `UPPER( original )`

Arguments:

- *original:* An expression that returns a string

Query

```
RETURN upper("hello")
```

Result

upper("hello")
"HELLO"

1 row

SPLIT

`SPLIT` returns the sequence of strings witch are delimited by split patterns.

Syntax: `SPLIT( original, splitPattern )`

Arguments:

- *original:* An expression that returns a string
- *splitPattern:* The string to split to original string with

Query

```
RETURN split("one,two", ",")
```

Result

split("one,two", ",")
["one","two"]

1 row

Chapter 14. Schema

Neo4j 2.0 introduced an optional schema for the graph, based around the concept of labels. Labels are used in the specification of indexes, and for defining constraints on the graph. Together, indexes and constraints are the schema of the graph. Cypher includes data definition language (DDL) statements for manipulating the schema.

14.1. Indexes

A database index is a redundant copy of information in the database for the purpose of making retrieving said data more efficient. This comes at the cost of additional storage space and slower writes.

Cypher allows the creation of indexes over a property for all nodes that have a given label. These indexes are automatically managed and kept up to date by the database whenever the graph is changed.

Create index on a label

To create an index on a property for all nodes that have a label, use `CREATE INDEX ON`. Note that the index is not immediately available, but will be created in the background. See the section called “Indexes” [10] for details.

Query

```
CREATE INDEX ON :Person(name)
```

Result

(empty result)

Drop index on a label

To drop an index on all nodes that have a label, use the `DROP INDEX` clause.

Query

```
DROP INDEX ON :Person(name)
```

Result

(empty result)

Indexes removed: 1

Use index

There is usually no need to specify which indexes to use in a query, Cypher will figure that out by itself. For example the query below will use the `Person(name)` index, if it exists. If you for some reason want to hint to specific indexes, see Section 10.8, “Using” [149].

Query

```
MATCH (person:Person { name: 'Andres' })
RETURN person
```

Result

person
Node[20]{name:"Andres"}

1 row

Use index with WHERE

Indexes are also automatically used for equality comparisons of a indexed property in the WHERE clause.If you for some reason want to hint to specific indexes, see Section 10.8, “Using” [149].

Query

```
MATCH (person:Person)
WHERE person.name = 'Andres'
RETURN person
```

Result

person
Node[20]{name:"Andres"}

1 row

Use index with IN

The IN predicate on `person.name` in the following query will use the `Person(name)` index, if it exists. If you for some reason want Cypher to use specific indexes, you can enforce it using hints. See Section 10.8, “Using” [149].

Query

```
MATCH (person:Person)
WHERE person.name IN ['Andres', 'Mark']
RETURN person
```

Result

person
Node[20]{name:"Andres"}
Node[21]{name:"Mark"}

2 rows

14.2. Constraints

Neo4j helps enforce data integrity with the use of constraints.

You can use unique constraints to ensure that property values are unique for all nodes with a specific label. Unique constraints do not mean that all nodes have to have a unique value for the properties — nodes without the property are not subject to this rule.

Remember that adding constraints is an atomic operation that can take a while — all existing data has to be scanned before Neo4j can turn the constraint “on”.

You can have multiple unique constraints for a given label.

Note that adding a uniqueness constraint on a property will also add an index on that property, so you cannot add such an index separately. Cypher will use that index for lookups just like other indexes. If you drop a constraint and still want an index on the property, you will have to create the index.

Create uniqueness constraint

To create a constraint that makes sure that your database will never contain more than one node with a specific label and one property value, use the `IS UNIQUE` syntax.

Query

```
CREATE CONSTRAINT ON (book:Book) ASSERT book.isbn IS UNIQUE
```

Result

```
(empty result)
```

Constraints added: 1

Drop uniqueness constraint

By using `DROP CONSTRAINT`, you remove a constraint from the database.

Query

```
DROP CONSTRAINT ON (book:Book) ASSERT book.isbn IS UNIQUE
```

Result

```
(empty result)
```

Constraints removed: 1

Create a node that complies with constraints

Create a `Book` node with an `isbn` that isn’t already in the database.

Query

```
CREATE (book:Book { isbn: '1449356265', title: 'Graph Databases' })
```

Result

```
(empty result)
```

Nodes created: 1
Properties set: 2
Labels added: 1

Create a node that breaks a constraint

Create a `Book` node with an `isbn` that is already used in the database.

Query

```
CREATE (book:Book { isbn: '1449356265', title: 'Graph Databases' })
```

In this case the node isn't created in the graph.

Error message

```
Node 0 already exists with label Book and property "isbn"=[1449356265]
```

Failure to create a constraint due to conflicting nodes

Create a constraint on the property `isbn` on nodes with the `Book` label when there are two nodes with the same `isbn`.

Query

```
CREATE CONSTRAINT ON (book:Book) ASSERT book.isbn IS UNIQUE
```

In this case the constraint can't be created because it is violated by existing data. We may choose to use Section 14.1, "Indexes" [236] instead or remove the offending nodes and then re-apply the constraint.

Error message

```
Unable to create CONSTRAINT ON ( book:Book ) ASSERT book.isbn IS UNIQUE:
Multiple nodes with label `Book` have property `isbn` = '1449356265':
  node(0)
  node(1)
```

14.3. Statistics

When you issue a Cypher query, it gets compiled to an execution plan (see Chapter 16, *Execution Plans* [247]) that can run and answer your question. To produce an efficient plan for your query, Neo4j needs information about your database, such as the schema — what indexes and constraints do exist? Neo4j will also use statistical information it keeps about your database to optimize the execution plan. With this information, Neo4j can decide which access pattern leads to the best performing plans.

The statistical information that Neo4j keeps is:

1. The number of nodes with a certain label.
2. Selectivity per index.
3. The number of relationships by type.
4. The number of relationships by type, ending or starting from a node with a specific label.

Neo4j keeps the statistics up to date in two different ways. For label counts for example, the number is updated whenever you set or remove a label from a node. For indexes, Neo4j needs to scan the full index to produce the selectivity number. Since this is potentially a very time-consuming operation, these numbers are collected in the background when enough data on the index has been changed.

Configuration options

`index_background_sampling_enabled`
: Controls whether indexes will automatically be re-sampled when they have been updated enough. The Cypher query planner depends on accurate statistics to create efficient plans, so it is important it is kept up to date as the database evolves.

> **Tip**
> If background sampling is turned off, make sure to trigger manual sampling when data has been updated.

`index_sampling_update_percentage`
: Controls how large portion of the index has to have been updated before a new sampling run is triggered.

Managing statistics from the shell

Usage:

`schema sample -a`	will sample all indexes.
`schema sample -l Person -p name`	will sample the index for label `Person` on property `name` (if existing).
`schema sample -a -f`	will force a sample of all indexes.
`schema sample -f -l :Person -p name`	will force sampling of a specific index.

Chapter 15. Query Tuning

Neo4j works very hard to execute queries as fast as possible.

However, when optimizing for maximum query execution performance, it may be helpful to rephrase queries using knowledge about the domain and the application.

The overall goal of manual query performance optimization is to ensure that only necessary data is retrieved from the graph. At least data should get filtered out as early as possible in order to reduce the amount of work that has to be done at later stages of query execution. This also goes for what gets returned: avoid returning whole nodes and relationships — instead, pick the data you need and return only that. You should also make sure to set an upper limit on variable length patterns, so they don't cover larger portions of the dataset than needed.

Each Cypher query gets optimized and transformed into an execution plan by the Cypher execution engine. To minimize the resources used for this, make sure to use parameters instead of literals when possible. This allows Cypher to re-use your queries instead of having to parse and build new execution plans.

To read more about the execution plan operators mentioned in this chapter, see Chapter 16, *Execution Plans* [247].

15.1. How are queries executed?

Each query is turned into an execution plan by something called the *execution planner*. The execution plan tells Neo4j which operations to perform when executing the query. Two different execution planning strategies are included in Neo4j:

Rule
: This planner has rules that are used to produce execution plans. The planner considers available indexes, but does not use statistical information to guide the query compilation.

Cost
: This planner uses the statistics service in Neo4j to assign cost to alternative plans and picks the cheapest one. While this should lead to superior execution plans in most cases, it is still under development.

By default, Neo4j 2.2 will use the cost planner for some queries, but not all. You can force it to use a specific planner by using the `query.planner.version` configuration setting (see dbms.cypher.planner [447]), or by prepending your query with `CYPHER planner=cost` or `CYPHER planner=rule`. Neo4j might still not use the planner you selected — not all queries are solvable by the cost planner at this point. Note that using `PLANNER COST` or `PLANNER RULE` in order to switch between planners has been deprecated and will stop working in future versions.

You can see which planner was used by looking at the execution plan.

Note
When Cypher is building execution plans, it looks at the schema to see if it can find indexes it can use. These index decisions are only valid until the schema changes, so adding or removing indexes leads to the execution plan cache being flushed.

15.2. How do I profile a query?

There are two options to choose from when you want to analyze a query by looking at its execution plan:

`EXPLAIN`
: If you want to see the execution plan but not run the statement, prepend your Cypher statement with `EXPLAIN`. The statement will always return an empty result and make no changes to the database.

`PROFILE`
: If you want to run the statement and see which operators are doing most of the work, use `PROFILE`. This will run your statement and keep track of how many rows pass through each operator, and how much each operator needs to interact with the storage layer to retrieve the necessary data. Please note that *profiling your query uses more resources,* so you should not profile unless you are actively working on a query.

See Chapter 16, *Execution Plans* [247] for a detailed explanation of each of the operators contained in an execution plan.

Tip

Being explicit about what types and labels you expect relationships and nodes to have in your query helps Neo4j use the best possible statistical information, which leads to better execution plans. This means that when you know that a relationship can only be of a certain type, you should add that to the query. The same goes for labels, where declaring labels on both the start and end nodes of a relationship helps Neo4j find the best way to execute the statement.

15.3. Basic query tuning example

We'll start with a basic example to help you get the hang of profiling queries. The following examples will use a movies data set.

Let's start by importing the data:

```
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.2.1/csv/query-tuning/movies.csv" AS line
MERGE (m:Movie { title:line.title })
ON CREATE SET m.released = toInt(line.released), m.tagline = line.tagline
```

```
LOAD CSV WITH HEADERS FROM 'http://neo4j.com/docs/2.2.1/csv/query-tuning/actors.csv' AS line
MATCH (m:Movie { title:line.title })
MERGE (p:Person { name:line.name })
ON CREATE SET p.born = toInt(line.born)
MERGE (p)-[:ACTED_IN { roles:split(line.roles,";")}]->(m)
```

```
LOAD CSV WITH HEADERS FROM 'http://neo4j.com/docs/2.2.1/csv/query-tuning/directors.csv' AS line
MATCH (m:Movie { title:line.title })
MERGE (p:Person { name:line.name })
ON CREATE SET p.born = toInt(line.born)
MERGE (p)-[:DIRECTED]->(m)
```

Let's say we want to write a query to find Tom Hanks. The naive way of doing this would be to write the following:

```
MATCH (p { name:"Tom Hanks" })
RETURN p
```

This query will find the Tom Hanks node but as the number of nodes in the database increase it will become slower and slower. We can profile the query to find out why that is.

You can learn more about the options for profiling queries in Section 15.2, "How do I profile a query?" [243] but in this case we're going to prefix our query with `PROFILE`:

```
PROFILE
MATCH (p { name:"Tom Hanks" })
RETURN p
```

```
Filter
  |
  +AllNodesScan

+--------------+---------------+------+--------+-------------+--------------------------+
|     Operator | EstimatedRows | Rows | DbHits | Identifiers |                    Other |
+--------------+---------------+------+--------+-------------+--------------------------+
|       Filter |            16 |    1 |    326 |           p | p.name == {  AUTOSTRING0} |
| AllNodesScan |           163 |  163 |    164 |           p |                          |
+--------------+---------------+------+--------+-------------+--------------------------+

Total database accesses: 490
```

The first thing to keep in mind when reading execution plans is that you need to read from the bottom up.

In that vein, starting from the last row, the first thing we notice is that the value in the `Rows` column seems high given there is only one node with the name property `Tom Hanks` in the database. If we look across to the `Operator` column we'll see that AllNodesScan has been used which means that the query planner scanned through all the nodes in the database.

Moving up to the previous row we see the Filter operator which will check the `name` property on each of the nodes passed through by `AllNodesScan`.

This seems like an inefficient way of finding `Tom Hanks` given that we are looking at many nodes that aren't even people and therefore aren't what we're looking for.

The solution to this problem is that whenever we're looking for a node we should specify a label to help the query planner narrow down the search space. For this query we'd need to add a `Person` label.

```
MATCH (p:Person { name:"Tom Hanks" })
RETURN p
```

This query will be faster than the first one but as the number of people in our database increase we again notice that the query slows down.

Again we can profile the query to work out why:

```
PROFILE
MATCH (p:Person { name:"Tom Hanks" })
RETURN p
```

```
Filter
  |
  +NodeByLabelScan

+-----------------+---------------+------+--------+-------------+--------------------------+
|        Operator | EstimatedRows | Rows | DbHits | Identifiers |                    Other |
+-----------------+---------------+------+--------+-------------+--------------------------+
|          Filter |            13 |    1 |    250 |           p | p.name == {  AUTOSTRING0} |
| NodeByLabelScan |           125 |  125 |    126 |           p |                  :Person |
+-----------------+---------------+------+--------+-------------+--------------------------+

Total database accesses: 376
```

This time the `Rows` value on the last row has reduced so we're not scanning some nodes that we were before which is a good start. The NodeByLabelScan operator indicates that we achieved this by first doing a linear scan of all the `Person` nodes in the database.

Once we've done that we again scan through all those nodes using the `Filter` operator, comparing the name property of each one.

This might be acceptable in some cases but if we're going to be looking up people by name frequently then we'll see better performance if we create an index on the `name` property for the `Person` label:

```
CREATE INDEX ON :Person(name)
```

Now if we run the query again it will run more quickly:

```
MATCH (p:Person { name:"Tom Hanks" })
RETURN p
```

Let's profile the query to see why that is:

```
PROFILE
MATCH (p:Person { name:"Tom Hanks" })
RETURN p
```

```
NodeIndexSeek

+---------------+---------------+------+--------+-------------+---------------+
|      Operator | EstimatedRows | Rows | DbHits | Identifiers |         Other |
+---------------+---------------+------+--------+-------------+---------------+
| NodeIndexSeek |             1 |    1 |      2 |           p | :Person(name) |
+---------------+---------------+------+--------+-------------+---------------+

Total database accesses: 2
```

Our execution plan is down to a single row and uses the Node Index Seek operator which does a schema index lookup (see Section 14.1, “Indexes” [236]) to find the appropriate node.

Chapter 16. Execution Plans

Cypher breaks down the work of executing a query into small pieces called operators. Each operator is responsible for a small part of the overall query. The operators are connected together in a pattern called a execution plan.

Each operator is annotated with statistics.

`Rows`	The number of rows that the operator produced. Only available if the query was profiled.
`EstimatedRows`	If Neo4j used the cost-based compiler you will see the estimated number of rows that will be produced by the operator. The compiler uses this estimate to choose a suitable execution plan.
`DbHits`	Each operator will ask the Neo4j storage engine to do work such as retrieving or updating data. A *database hit* is an abstract unit of this storage engine work.

See Section 15.2, “How do I profile a query?” [243] for how to view the execution plan for your query.

For a deeper understanding of how each operator works, see the relavent section. Operators are grouped into high-level categories.

16.1. Starting a query operators

These operators find parts of the graph from which to start a query.

All Nodes Scan

Reads all nodes from the node store. The identifier that will contain the nodes is seen in the arguments. The following query will return all nodes. It's not a good idea to run a query like this on a production database.

Query

```
MATCH (n)
RETURN n
```

Query Plan

```
AllNodesScan

+--------------+---------------+------+--------+-------------+-------+
|     Operator | EstimatedRows | Rows | DbHits | Identifiers | Other |
+--------------+---------------+------+--------+-------------+-------+
| AllNodesScan |            27 |   27 |     28 |           n |       |
+--------------+---------------+------+--------+-------------+-------+

Total database accesses: 28
```

Directed Relationship By Id Seek

Reads one or more relationships by id from the relationship store. Produces both the relationship and the nodes on either side.

The following query will find the relationship with id `0` and will return a row for the source node of that relationship.

Query

```
MATCH (n1)-[r]->()
WHERE id(r)= 0
RETURN r, n1
```

Query Plan

```
Projection
  |
  +DirectedRelationshipByIdSeekPipe

+-----------------------------------+---------------+------+--------+-----------------
+--------------------------------------------------+
|                          Operator | EstimatedRows | Rows | DbHits |     Identifiers |
       Other |
+-----------------------------------+---------------+------+--------+-----------------
+--------------------------------------------------+
|                        Projection |             0 |    1 |      0 | anon[17], n1, r |
       r; n1 |
| DirectedRelationshipByIdSeekPipe |             0 |    1 |      1 | anon[17], n1, r | EntityByIdRhs(EntityByIdExprs(List({
  AUTOINT0}))) |
+-----------------------------------+---------------+------+--------+-----------------
+--------------------------------------------------+

Total database accesses: 1
```

Node by Id seek

Reads one or more nodes by id from the node store. The following query will return the node which has nodeId `0`.

Query

```
MATCH n
WHERE id(n)= 0
RETURN n
```

Query Plan

```
NodeByIdSeek

+--------------+---------------+------+--------+-------------+-------+
|     Operator | EstimatedRows | Rows | DbHits | Identifiers | Other |
+--------------+---------------+------+--------+-------------+-------+
| NodeByIdSeek |             1 |    1 |      1 |           n |       |
+--------------+---------------+------+--------+-------------+-------+

Total database accesses: 1
```

Node by label scan

Using the label index, fetches all nodes with a specific label on them. The following query will return all nodes which have label `Person` where the property `name` has the value "me" via a scan of the `Person` label index.

Query

```
MATCH (person:Person { name: "me" })
RETURN person
```

Query Plan

```
Filter
  |
  +NodeByLabelScan

+-----------------+---------------+------+--------+-------------+--------------------------------+
|        Operator | EstimatedRows | Rows | DbHits | Identifiers |                          Other |
+-----------------+---------------+------+--------+-------------+--------------------------------+
|          Filter |             1 |    1 |     12 |      person | person.name == {  AUTOSTRING0} |
| NodeByLabelScan |             6 |    6 |      7 |      person |                        :Person |
+-----------------+---------------+------+--------+-------------+--------------------------------+

Total database accesses: 19
```

Node index seek

Finds nodes using an index seek. The node identifier and the index used is shown in the arguments of the operator. The following query will return all nodes which have label `Location` where the property `name` has the value "Malmo" using the `Location` index.

Query

```
MATCH (location:Location { name: "Malmo" })
RETURN location
```

Query Plan

```
NodeIndexSeek
```

```
+---------------+---------------+------+--------+-------------+-----------------+
|      Operator | EstimatedRows | Rows | DbHits | Identifiers |           Other |
+---------------+---------------+------+--------+-------------+-----------------+
| NodeIndexSeek |             1 |    1 |      2 |    location | :Location(name) |
+---------------+---------------+------+--------+-------------+-----------------+

Total database accesses: 2
```

Node unique index seek

Finds nodes using an index seek on a unique index. The node identifier and the index used is shown in the arguments of the operator. The following query will return all nodes which have the label `Team` where the property `name` has the value "Field" using the `Team` unique index.

Query

```
MATCH (team:Team { name: "Field" })
RETURN team
```

Query Plan

```
NodeUniqueIndexSeek

+---------------------+---------------+------+--------+-------------+-------------+
|            Operator | EstimatedRows | Rows | DbHits | Identifiers |       Other |
+---------------------+---------------+------+--------+-------------+-------------+
| NodeUniqueIndexSeek |             1 |    1 |      1 |        team | :Team(name) |
+---------------------+---------------+------+--------+-------------+-------------+

Total database accesses: 1
```

Undirected Relationship By Id Seek

Reads one or more relationships by id from the relationship store. For each relationship, two rows are produced with start and end nodes arranged differently.

The following query will find the relationship with id `1` and will return a row for both the source and destination nodes of that relationship.

Query

```
MATCH (n1)-[r]-()
WHERE id(r)= 1
RETURN r, n1
```

Query Plan

```
Projection
  |
  +UndirectedRelationshipByIdSeek

+--------------------------------+---------------+------+--------+----------------+-------+
|                       Operator | EstimatedRows | Rows | DbHits |    Identifiers | Other |
+--------------------------------+---------------+------+--------+----------------+-------+
|                     Projection |             1 |    2 |      0 | anon[16], n1, r | r; n1 |
| UndirectedRelationshipByIdSeek |             1 |    2 |      1 | anon[16], n1, r |       |
+--------------------------------+---------------+------+--------+----------------+-------+

Total database accesses: 1
```

16.2. Expand operators

Thes operators explore the graph by expanding graph patterns.

Expand All

Given a start node, expand will follow relationships coming in or out, depending on the pattern relationship. Can also handle variable length pattern relationships.

The following query will return my friends of friends.

Query

```
MATCH (p:Person { name: "me" })-[:FRIENDS_WITH]->(fof)
RETURN fof
```

Query Plan

```
Projection
  |
  +Expand(All)
    |
    +Filter
      |
      +NodeByLabelScan

+-----------------+---------------+------+--------+------------------+----------------------------+
|        Operator | EstimatedRows | Rows | DbHits |      Identifiers |                      Other |
+-----------------+---------------+------+--------+------------------+----------------------------+
|      Projection |             0 |    1 |      0 | anon[30], fof, p |                        fof |
|     Expand(All) |             0 |    1 |      2 | anon[30], fof, p | (p)-[:FRIENDS_WITH]->(fof) |
|          Filter |             1 |    1 |     12 |                p |  p.name == {  AUTOSTRING0} |
| NodeByLabelScan |             6 |    6 |      7 |                p |                    :Person |
+-----------------+---------------+------+--------+------------------+----------------------------+

Total database accesses: 21
```

Optional Expand All

Optional expand traverses relationships from a given node, and makes sure that predicates are evaluated before producing rows.

If no matching relationships are found, a single row with `NULL` for the relationship and end node identifier is produced.

The following query will find all the people and the location they work in as long as they've worked there for more than 180 days.

Query

```
MATCH (p:Person)
OPTIONAL MATCH (p)-[works_in:WORKS_IN]->(l)
WHERE works_in.duration > 180
RETURN p, l
```

Query Plan

```
Projection
  |
  +OptionalExpand(All)
    |
    +NodeByLabelScan

+---------------------+---------------+------+--------+----------------+------------------------------+
|            Operator | EstimatedRows | Rows | DbHits |    Identifiers |                        Other |
```

```
+---------------------+---------------+------+--------+----------------+------------------------------+
|          Projection |             6 |    6 |      0 | l, p, works_in |                         p; l |
| OptionalExpand(All) |             6 |    6 |     18 | l, p, works_in | (p)-[works_in:WORKS_IN]->(l) |
|     NodeByLabelScan |             6 |    6 |      7 |              p |                      :Person |
+---------------------+---------------+------+--------+----------------+------------------------------+

Total database accesses: 25
```

16.3. Combining operators

Anti Semi Apply

Tests for the absence of a pattern predicate. A pattern predicate that is prepended by `NOT` is solved with `AntiSemiApply`.

The following query will find all the people who aren't my friend.

Query

```
MATCH (me:Person { name: "me" }),(other:Person)
WHERE NOT (me)-[:FRIENDS_WITH]->(other)
RETURN other
```

Query Plan

```
Projection
  |
  +AntiSemiApply
    |
    +CartesianProduct
    |  |
    |  +Filter
    |  |  |
    |  |  +NodeByLabelScan(0)
    |  |
    |  +NodeByLabelScan(1)
    |
    +Expand(Into)
       |
       +Argument

+--------------------+---------------+------+--------+---------------------+-------------------------------+
|           Operator | EstimatedRows | Rows | DbHits |         Identifiers |                         Other |
+--------------------+---------------+------+--------+---------------------+-------------------------------+
|         Projection |             1 |    5 |      0 |           me, other |                         other |
|      AntiSemiApply |             1 |    5 |      0 |           me, other |                               |
|   CartesianProduct |             4 |    6 |      0 |           me, other |                               |
|             Filter |             1 |    1 |     12 |                  me |    me.name == {  AUTOSTRING0} |
| NodeByLabelScan(0) |             6 |    6 |      7 |                  me |                       :Person |
| NodeByLabelScan(1) |             6 |    6 |      7 |               other |                       :Person |
|       Expand(Into) |             0 |    0 |     20 | anon[73], me, other | (me)-[:FRIENDS_WITH]->(other) |
|           Argument |             4 |    6 |      0 |           me, other |                               |
+--------------------+---------------+------+--------+---------------------+-------------------------------+

Total database accesses: 46
```

Let Anti Semi Apply

Tests for the absence of a pattern predicate. When a query contains multiple pattern predicates `LetSemiApply` will be used to evaluate the first of these. It will record the result of evaluating the predicate but will leave any filtering to another operator. The following query will find all the people who don't have anyfriend or who work somewhere. The `LetSemiApply` operator will be used to check for the absence of the `FRIENDS_WITH` relationship from each person.

Query

```
MATCH (other:Person)
WHERE NOT ((other)-[:FRIENDS_WITH]->()) OR (other)-[:WORKS_IN]->()
RETURN other
```

Query Plan

```
Projection
  |
  +SelectOrSemiApply
    |
    +LetAntiSemiApply
    |  |
    |  +NodeByLabelScan
    |  |
    |  +Expand(All)(0)
    |     |
    |     +Argument(1)
    |
    +Expand(All)(1)
       |
       +Argument(1)

+--------------------+---------------+------+--------+----------------------------+-----------------------------+
|           Operator | EstimatedRows | Rows | DbHits |                Identifiers |                       Other |
+--------------------+---------------+------+--------+----------------------------+-----------------------------+
|         Projection |             5 |    6 |      0 |           anon[42], other  |                       other |
| SelectOrSemiApply  |             5 |    6 |      0 |           anon[42], other  |                    anon[42] |
|  LetAntiSemiApply  |             6 |    6 |      0 |           anon[42], other  |                             |
|   NodeByLabelScan  |             6 |    6 |      7 |                      other |                     :Person |
|    Expand(All)(0)  |             2 |    0 |      6 | anon[50], anon[68], other  | (other)-[:FRIENDS_WITH]->() |
|       Argument(1)  |             6 |    8 |      0 |                      other |                             |
|    Expand(All)(1)  |             4 |    0 |      2 | anon[82], anon[96], other  |     (other)-[:WORKS_IN]->() |
|       Argument(1)  |             6 |    8 |      0 |                      other |                             |
+--------------------+---------------+------+--------+----------------------------+-----------------------------+

Total database accesses: 15
```

Let Semi Apply

Tests for the existence of a pattern predicate. When a query contains multiple pattern predicates `LetSemiApply` will be used to evaluate the first of these. It will record the result of evaluating the predicate but will leave any filtering to a another operator. The following query will find all the people who have a friend or who work somewhere. The `LetSemiApply` operator will be used to check for the existence of the `FRIENDS_WITH` relationship from each person.

Query

```
MATCH (other:Person)
WHERE (other)-[:FRIENDS_WITH]->() OR (other)-[:WORKS_IN]->()
RETURN other
```

Query Plan

```
Projection
  |
  +SelectOrSemiApply
    |
    +LetSemiApply
    |  |
    |  +NodeByLabelScan
    |  |
    |  +Expand(All)(0)
    |     |
    |     +Argument(1)
    |
    +Expand(All)(1)
       |
       +Argument(1)

+--------------------+---------------+------+--------+----------------------------+-----------------------------+
```

```
|          Operator | EstimatedRows | Rows | DbHits |              Identifiers |                      Other |
+-------------------+---------------+------+--------+--------------------------+----------------------------+
|        Projection |             6 |    4 |      0 |           anon[38], other |                      other |
| SelectOrSemiApply |             6 |    4 |      0 |           anon[38], other |                   anon[38] |
|      LetSemiApply |             6 |    6 |      0 |           anon[38], other |                            |
|   NodeByLabelScan |             6 |    6 |      7 |                     other |                    :Person |
|    Expand(All)(0) |             2 |    0 |      6 | anon[46], anon[64], other | (other)-[:FRIENDS_WITH]->() |
|       Argument(1) |             6 |   10 |      0 |                     other |                            |
|    Expand(All)(1) |             4 |    0 |      4 | anon[77], anon[91], other |     (other)-[:WORKS_IN]->() |
|       Argument(1) |             6 |   10 |      0 |                     other |                            |
+-------------------+---------------+------+--------+--------------------------+----------------------------+

Total database accesses: 17
```

Select Or Anti Semi Apply

Tests for the absence of a pattern predicate and evaluates a predicate.

The following query will find all the people who don't have a friend or are older than 25.

Query

```
MATCH (other:Person)
WHERE other.age > 25 OR NOT (other)-[:FRIENDS_WITH]->()
RETURN other
```

Query Plan

```
SelectOrAntiSemiApply
  |
  +NodeByLabelScan
  |
  +Expand(All)
     |
     +Argument

+----------------------+---------------+------+--------+--------------------------+----------------------------+
|             Operator | EstimatedRows | Rows | DbHits |              Identifiers |                      Other |
+----------------------+---------------+------+--------+--------------------------+----------------------------+
| SelectOrAntiSemiApply |             3 |    4 |      6 |                     other |    other.age > {  AUTOINT0} |
|      NodeByLabelScan |             6 |    6 |      7 |                     other |                    :Person |
|          Expand(All) |             2 |    0 |      6 | anon[68], anon[86], other | (other)-[:FRIENDS_WITH]->() |
|             Argument |             6 |    6 |      0 |                     other |                            |
+----------------------+---------------+------+--------+--------------------------+----------------------------+

Total database accesses: 19
```

Select Or Semi Apply

Tests for the existence of a pattern predicate and evaluates a predicate. This operator allows for the mixing of normal predicates and pattern predicates that check for the existing of a pattern. First the normal expression predicate is evaluated, and only if it returns `FALSE` the costly pattern predicate evaluation is performed.

The following query will find all the people who have a friend or are older than 25.

Query

```
MATCH (other:Person)
WHERE other.age > 25 OR (other)-[:FRIENDS_WITH]->()
RETURN other
```

Query Plan

```
SelectOrSemiApply
```

```
  |
  +NodeByLabelScan
  |
  +Expand(All)
     |
     +Argument

+-------------------+---------------+------+--------+---------------------------+-----------------------------+
|          Operator | EstimatedRows | Rows | DbHits |               Identifiers |                       Other |
+-------------------+---------------+------+--------+---------------------------+-----------------------------+
| SelectOrSemiApply |             5 |    2 |      6 |                     other |    other.age > {  AUTOINT0} |
|   NodeByLabelScan |             6 |    6 |      7 |                     other |                     :Person |
|       Expand(All) |             2 |    0 |      6 | anon[64], anon[82], other | (other)-[:FRIENDS_WITH]->() |
|          Argument |             6 |    6 |      0 |                     other |                             |
+-------------------+---------------+------+--------+---------------------------+-----------------------------+

Total database accesses: 19
```

Semi Apply

Tests for the existence of a pattern predicate. `SemiApply` takes a row from it's child operator and feeds it to the `Argument` operator on the right hand side of `SemiApply`. If the right hand side operator tree yields at least one row, the row from the left hand side is yielded by the `SemiApply` operator. This makes `SemiApply` a filtering operator, used mostly for pattern predicates in queries.

The following query will find all the people who have a friend.

Query

```
MATCH (other:Person)
WHERE (other)-[:FRIENDS_WITH]->()
RETURN other
```

Query Plan

```
SemiApply
  |
  +NodeByLabelScan
  |
  +Expand(All)
     |
     +Argument

+-----------------+---------------+------+--------+---------------------------+-----------------------------+
|        Operator | EstimatedRows | Rows | DbHits |               Identifiers |                       Other |
+-----------------+---------------+------+--------+---------------------------+-----------------------------+
|       SemiApply |             5 |    2 |      0 |                     other |                             |
| NodeByLabelScan |             6 |    6 |      7 |                     other |                     :Person |
|     Expand(All) |             2 |    0 |      6 | anon[46], anon[64], other | (other)-[:FRIENDS_WITH]->() |
|        Argument |             6 |    6 |      0 |                     other |                             |
+-----------------+---------------+------+--------+---------------------------+-----------------------------+

Total database accesses: 13
```

16.4. Row operators

These operators take rows produced by another operator and transform them to a different set of rows

Distinct

Removes duplicate rows. The following query will return unique locations that have people working in them

Query

```
MATCH (l:Location)<-[:WORKS_IN]-(p:Person)
RETURN DISTINCT l
```

Query Plan

```
Distinct
  |
  +Filter
    |
    +Expand(All)
      |
      +NodeByLabelScan

+-----------------+---------------+------+--------+----------------+----------------------+
|        Operator | EstimatedRows | Rows | DbHits |    Identifiers |                Other |
+-----------------+---------------+------+--------+----------------+----------------------+
|        Distinct |             4 |    1 |      0 |              l |                    l |
|          Filter |             4 |    4 |      4 | anon[19], l, p |           l:Location |
|     Expand(All) |             4 |    4 |     10 | anon[19], l, p | (p)-[:WORKS_IN]->(l) |
| NodeByLabelScan |             6 |    6 |      7 |              p |              :Person |
+-----------------+---------------+------+--------+----------------+----------------------+

Total database accesses: 21
```

Eager Aggregation

Eagerly loads resulting sub graphs before starting to emit the aggregated results. The following query will collect the people who work in every location before returning any rows.

Query

```
MATCH (l:Location)<-[:WORKS_IN]-(p:Person)
RETURN l.name AS location, COLLECT(p.name) AS people
```

Query Plan

```
EagerAggregation
  |
  +Projection
    |
    +Filter
      |
      +Expand(All)
        |
        +NodeByLabelScan

+------------------+---------------+------+--------+--------------------------+----------------------+
|         Operator | EstimatedRows | Rows | DbHits |              Identifiers |                Other |
+------------------+---------------+------+--------+--------------------------+----------------------+
| EagerAggregation |             2 |    1 |      8 |         location, people |             location |
|       Projection |             4 |    4 |      8 | anon[19], l, location, p |            l.name; p |
|           Filter |             4 |    4 |      4 |           anon[19], l, p |           l:Location |
|      Expand(All) |             4 |    4 |     10 |           anon[19], l, p | (p)-[:WORKS_IN]->(l) |
```

```
|  NodeByLabelScan |             6 |    6 |      7 |                        p |              :Person |
+------------------+---------------+------+--------+--------------------------+----------------------+

Total database accesses: 37
```

Filter

Filters each row coming from the child operator, only passing through rows that evaluate the predicates to `TRUE`. The following query will look for nodes with the label `Person` and filter those whose name begins with the letter `a`.

Query

```
MATCH (p:Person)
WHERE p.name =~ "^a.*"
RETURN p
```

Query Plan

```
Filter
  |
  +NodeByLabelScan

+-----------------+---------------+------+--------+-------------+----------------------------+
|        Operator | EstimatedRows | Rows | DbHits | Identifiers |                      Other |
+-----------------+---------------+------+--------+-------------+----------------------------+
|          Filter |             5 |    0 |     12 |           p | p.name ~= /{  AUTOSTRING0}/ |
| NodeByLabelScan |             6 |    6 |      7 |           p |                    :Person |
+-----------------+---------------+------+--------+-------------+----------------------------+

Total database accesses: 19
```

Limit

Returns the first *n* rows.

The following query will return the first 3 people in an arbitrary order.

Query

```
MATCH (p:Person)
RETURN p
LIMIT 3
```

Query Plan

```
Limit
  |
  +NodeByLabelScan

+-----------------+---------------+------+--------+-------------+--------------+
|        Operator | EstimatedRows | Rows | DbHits | Identifiers |        Other |
+-----------------+---------------+------+--------+-------------+--------------+
|           Limit |             6 |    3 |      0 |           p | {  AUTOINT0} |
| NodeByLabelScan |             6 |    3 |      4 |           p |      :Person |
+-----------------+---------------+------+--------+-------------+--------------+

Total database accesses: 4
```

Projection

For each row from its input, projection executes a set of expressions and produces a row with the results of the expressions. The following query will produce one row with the value "hello".

Query

```
RETURN "hello" AS greeting
```

Query Plan

```
Projection

+------------+---------------+------+--------+-------------+-----------------+
|   Operator | EstimatedRows | Rows | DbHits | Identifiers |           Other |
+------------+---------------+------+--------+-------------+-----------------+
| Projection |             1 |    1 |      0 |    greeting | {  AUTOSTRING0} |
+------------+---------------+------+--------+-------------+-----------------+

Total database accesses: 0
```

Skip

Skips *n* rows

The following query will skip the person with the lowest `id` property and return the rest.

Query

```
MATCH (p:Person)
RETURN p
ORDER BY p.id
SKIP 1
```

Query Plan

```
Projection(0)
  |
  +Skip
    |
    +Sort
      |
      +Projection(1)
        |
        +Projection(2)
          |
          +NodeByLabelScan

+-----------------+---------------+------+--------+-------------------------+-----------------------+
|        Operator | EstimatedRows | Rows | DbHits |             Identifiers |                 Other |
+-----------------+---------------+------+--------+-------------------------+-----------------------+
|   Projection(0) |             6 |    5 |      0 | anon[35], anon[59], p, p |             anon[35] |
|            Skip |             6 |    5 |      0 |    anon[35], anon[59], p |          {  AUTOINT0} |
|            Sort |             6 |    6 |      0 |    anon[35], anon[59], p |             anon[59] |
|   Projection(1) |             6 |    6 |      6 |    anon[35], anon[59], p | anon[35]; anon[35].id |
|   Projection(2) |             6 |    6 |      0 |              anon[35], p |                     p |
| NodeByLabelScan |             6 |    6 |      7 |                       p |               :Person |
+-----------------+---------------+------+--------+-------------------------+-----------------------+

Total database accesses: 13
```

Sort

Sorts rows by a provided key.

The following query will find all the people and return them sorted alphabetically by name.

Query

```
MATCH (p:Person)
RETURN p
ORDER BY p.name
```

Query Plan

```
Projection(0)
  |
  +Sort
    |
    +Projection(1)
      |
      +Projection(2)
        |
        +NodeByLabelScan

+-----------------+---------------+------+--------+--------------------------+-------------------------+
|        Operator | EstimatedRows | Rows | DbHits |              Identifiers |                   Other |
+-----------------+---------------+------+--------+--------------------------+-------------------------+
|   Projection(0) |             6 |    6 |      0 | anon[24], anon[37], p, p |                anon[24] |
|            Sort |             6 |    6 |      0 |    anon[24], anon[37], p |                anon[37] |
|   Projection(1) |             6 |    6 |     12 |    anon[24], anon[37], p | anon[24]; anon[24].name |
|   Projection(2) |             6 |    6 |      0 |              anon[24], p |                       p |
| NodeByLabelScan |             6 |    6 |      7 |                        p |                 :Person |
+-----------------+---------------+------+--------+--------------------------+-------------------------+

Total database accesses: 19
```

Top

Returns the first *n* rows sorted by a provided key. The physical operator is called `Top`. Instead of sorting the whole input, only the top X rows are kept.

The following query will find the first 2 people sorted alphabetically by name.

Query

```
MATCH (p:Person)
RETURN p
ORDER BY p.name
LIMIT 2
```

Query Plan

```
Projection(0)
  |
  +Top
    |
    +Projection(1)
      |
      +Projection(2)
        |
        +NodeByLabelScan

+-----------------+---------------+------+--------+--------------------------+-------------------------+
|        Operator | EstimatedRows | Rows | DbHits |              Identifiers |                   Other |
+-----------------+---------------+------+--------+--------------------------+-------------------------+
|   Projection(0) |             6 |    2 |      0 | anon[24], anon[37], p, p |                anon[24] |
|             Top |             6 |    2 |      0 |    anon[24], anon[37], p |          {  AUTOINT0};  |
|   Projection(1) |             6 |    6 |     12 |    anon[24], anon[37], p | anon[24]; anon[24].name |
|   Projection(2) |             6 |    6 |      0 |              anon[24], p |                       p |
| NodeByLabelScan |             6 |    6 |      7 |                        p |                 :Person |
+-----------------+---------------+------+--------+--------------------------+-------------------------+

Total database accesses: 19
```

Union

Union concatenates the results from the right plan after the results of the left plan.

Query

```
MATCH (p:Location)
RETURN p.name
UNION ALL MATCH (p:Country)
RETURN p.name
```

Query Plan

```
Union
  |
  +Projection(0)
  |  |
  |  +NodeByLabelScan(0)
  |
  +Projection(1)
     |
     +NodeByLabelScan(1)

+--------------------+---------------+------+--------+-------------+-----------+
|           Operator | EstimatedRows | Rows | DbHits | Identifiers |     Other |
+--------------------+---------------+------+--------+-------------+-----------+
|              Union |            10 |   11 |      0 |   p, p.name |           |
|      Projection(0) |            10 |   10 |     20 |   p, p.name |    p.name |
| NodeByLabelScan(0) |            10 |   10 |     11 |           p | :Location |
|      Projection(1) |             1 |    1 |      2 |   p, p.name |    p.name |
| NodeByLabelScan(1) |             1 |    1 |      2 |           p |  :Country |
+--------------------+---------------+------+--------+-------------+-----------+

Total database accesses: 35
```

Unwind

Takes a collection of values and returns one row per item in the collection. The following query will return one row for each of the numbers 1 to 5.

Query

```
UNWIND range(1,5) AS value
RETURN value;
```

Query Plan

```
UNWIND
  |
  +Argument

+----------+---------------+------+--------+-------------+-------+
| Operator | EstimatedRows | Rows | DbHits | Identifiers | Other |
+----------+---------------+------+--------+-------------+-------+
|   UNWIND |            10 |    5 |      0 |       value |       |
| Argument |             - |    1 |      0 |             |       |
+----------+---------------+------+--------+-------------+-------+

Total database accesses: 0
```

16.5. Update Operators

These operators are used in queries that update the graph.

Constraint Operation

Creates a constraint on a (label,property) pair. The following query will create a unique constraint on the `name` property of nodes with the `Country` label. This will ensure that we won't end up with duplicate `Country` nodes in our database.

Query

```
CREATE CONSTRAINT ON (c:Country) ASSERT c.name IS UNIQUE
```

Query Plan

```
ConstraintOperation

+---------------------+------+--------+-------------+-------+
|            Operator | Rows | DbHits | Identifiers | Other |
+---------------------+------+--------+-------------+-------+
| ConstraintOperation |    0 |      3 |             |       |
+---------------------+------+--------+-------------+-------+

Total database accesses: 3
```

Empty Result

Represents the fact that a query doesn't return any results. The following query will create a node but won't return anything.

Query

```
CREATE (:Person)
```

Query Plan

```
EmptyResult
  |
  +UpdateGraph

+-------------+------+--------+-------------+------------+
|    Operator | Rows | DbHits | Identifiers |      Other |
+-------------+------+--------+-------------+------------+
| EmptyResult |    0 |      0 |             |            |
| UpdateGraph |    1 |      2 |     anon[7] | CreateNode |
+-------------+------+--------+-------------+------------+

Total database accesses: 2
```

Update Graph

Applies updates to the graph. The following query will create a `Person` node with the name property set to `Alistair`

Query

```
CREATE (:Person { name: "Alistair" })
```

Query Plan

```
EmptyResult
  |
  +UpdateGraph
```

```
+-------------+------+--------+-------------+------------+
|    Operator | Rows | DbHits | Identifiers |      Other |
+-------------+------+--------+-------------+------------+
| EmptyResult |    0 |      0 |             |            |
| UpdateGraph |    1 |      4 |     anon[7] | CreateNode |
+-------------+------+--------+-------------+------------+

Total database accesses: 4
```

Part IV. Reference

The reference part is the authoritative source for details on Neo4j usage. It covers details on capabilities, transactions, indexing and queries among other topics.

Chapter 17. Capabilities

17.1. 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.

17.2. Data Integrity

In order to keep data consistent, a good database needs mechanisms and structures that guarantee the integrity of all stored data. In Neo4j, data integrity is guaranteed both for graph elements (Nodes, Relationships and Properties) and for non-graph data, such as the indexes. Neo4j's transactional architecture ensures that data is protected and provides for fast recovery from an unexpected failure, without the need to rebuild internal indexes or other costly operations.

17.3. 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.

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.

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.

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.

17.4. 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.

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.

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.

17.5. Capacity

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.

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.

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.

Data size

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

nodes	2^{35} (~ 34 billion)
relationships	2^{35} (~ 34 billion)
properties	2^{36} to 2^{38} depending on property types (maximum ~ 274 billion, always at least ~ 68 billion)
relationship types	2^{15} (~ 32 000)

Chapter 18. 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 database operations that access the graph, indexes, or the schema must be performed in a transaction.
- 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.

18.1. Interaction cycle

All database operations that access the graph, indexes, or the schema 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.

The interaction cycle of working with transactions looks like this:

1. Begin a transaction.
2. Perform database 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.

18.2. Isolation levels

Transactions in Neo4j use a read-committed isolation level, which means they will see data as soon as it has been committed and will not see data in other transactions that have not yet been committed. This type of isolation is weaker than serialization but offers significant performance advantages whilst being sufficient for the overwhelming majority of cases.

In addition, the Neo4j Java API (see Part VIII, “Advanced Usage” [574]) enables explicit locking of nodes and relationships. Using locks gives the opportunity to simulate the effects of higher levels of isolation by obtaining and releasing locks explicitly. For example, if a write lock is taken on a common node or relationship, then all transactions will serialize on that lock — giving the effect of a serialization isolation level.

18.3. 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.

18.4. Deadlocks

Understanding 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.

Deadlock handling example code

Below you'll find examples of how deadlocks can be handled in server extensions/plugins or when using Neo4j embedded.

Tip
The full source code used for the code snippets can be found at DeadlockDocTest.java[1].

When dealing with deadlocks in code, there are several issues you may want to address:

- Only do a limited amount of retries, and fail if a threshold is reached.
- Pause between each attempt to allow the other transaction to finish before trying again.
- A retry-loop can be useful not only for deadlocks, but for other types of transient errors as well.

In the following sections you'll find example code in Java which shows how this can be implemented.

Handling deadlocks using TransactionTemplate

If you don't want to write all the code yourself, there is a class called `TransactionTemplate`[2] that will help you achieve what's needed. Below is an example of how to create, customize, and use this template for retries in transactions.

First, define the base template:

[1] https://github.com/neo4j/neo4j/blob/2.2.1/community/kernel/src/test/java/examples/DeadlockDocTest.java
[2] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/helpers/TransactionTemplate.html

```
TransactionTemplate template = new TransactionTemplate(  ).retries( 5 ).backoff( 3, TimeUnit.SECONDS );
```

Next, specify the database to use and a function to execute:

```
Object result = template.with(graphDatabaseService).execute( new Function<Transaction, Object>()
{
    @Override
    public Object apply( Transaction transaction ) throws RuntimeException
    {
        Object result = null;
        return result;
    }
} );
```

The operations that could lead to a deadlock should go into the `apply` method.

The `TransactionTemplate` uses a fluent API for configuration, and you can choose whether to set everything at once, or (as in the example) provide some details just before using it. The template allows setting a predicate for what exceptions to retry on, and also allows for easy monitoring of events that take place.

Handling deadlocks using a retry loop

If you want to roll your own retry-loop code, see below for inspiration. Here's an example of what a retry block might look like:

```
Throwable txEx = null;
int RETRIES = 5;
int BACKOFF = 3000;
for ( int i = 0; i < RETRIES; i++ )
{
    try ( Transaction tx = graphDatabaseService.beginTx() )
    {
        Object result = doStuff(tx);
        tx.success();
        return result;
    }
    catch ( Throwable ex )
    {
        txEx = ex;

        // Add whatever exceptions to retry on here
        if ( !(ex instanceof DeadlockDetectedException) )
        {
            break;
        }
    }

    // Wait so that we don't immediately get into the same deadlock
    if ( i < RETRIES - 1 )
    {
        try
        {
            Thread.sleep( BACKOFF );
        }
        catch ( InterruptedException e )
        {
            throw new TransactionFailureException( "Interrupted", e );
        }
    }
}

if ( txEx instanceof TransactionFailureException )
{
```

```
    throw ((TransactionFailureException) txEx);
}
else if ( txEx instanceof Error )
{
    throw ((Error) txEx);
}
else if ( txEx instanceof RuntimeException )
{
    throw ((RuntimeException) txEx);
}
else
{
    throw new TransactionFailureException( "Failed", txEx );
}
```

The above is the gist of what such a retry block would look like, and which you can customize to fit your needs.

18.5. 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.

18.6. Creating unique nodes

In many use cases, a certain level of uniqueness is desired among entities. You could for instance imagine that only one user with a certain e-mail address may exist in a system. If multiple concurrent threads naively try to create the user, duplicates will be created. There are three main strategies for ensuring uniqueness, and they all work across High Availability and single-instance deployments.

Single thread

By using a single thread, no two threads will even try to create a particular entity simultaneously. On High Availability, an external single-threaded client can perform the operations on the cluster.

Get or create

The preferred way to get or create a unique node is to use unique constraints and Cypher. See the section called "Get or create unique node using Cypher and unique constraints" [616] for more information.

By using `put-if-absent`[3] functionality, entity uniqueness can be guaranteed using a legacy index. Here the legacy index acts as the lock and will only lock the smallest part needed to guaranteed uniqueness across threads and transactions.

See the section called "Get or create unique node using a legacy index" [616] for how to do this using the core Java API. When using the REST API, see Section 21.21, "Unique Indexing" [403].

Pessimistic locking

Important
While this is a working solution, please consider using the preferred the section called "Get or create" [280] instead.

By using explicit, pessimistic locking, unique creation of entities can be achieved in a multi-threaded environment. It is most commonly done by locking on a single or a set of common nodes.

See the section called "Pessimistic locking for node creation" [617] for how to do this using the core Java API.

[3] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/Index.html#putIfAbsent%28T,%20java.lang.String,%20java.lang.Object%29

18.7. Transaction events

Transaction event handlers can be registered to receive Neo4j Transaction events. Once it has been registered at a `GraphDatabaseService` instance it will receive events about what has happened in each transaction which is about to be committed. Handlers won't get notified about transactions which haven't performed any write operation or won't be committed (either if `Transaction#success()` hasn't been called or the transaction has been marked as failed `Transaction#failure()`. Right before a transaction is about to be committed the `beforeCommit` method is called with the entire diff of modifications made in the transaction. At this point the transaction is still running so changes can still be made. However there's no guarantee that other handlers will see such changes since the order in which handlers are executed is undefined. This method can also throw an exception and will, in such a case, prevent the transaction from being committed (where a call to `afterRollback` will follow). If `beforeCommit` is successfully executed the transaction will be committed and the `afterCommit` method will be called with the same transaction data as well as the object returned from `beforeCommit`. This assumes that all other handlers (if more were registered) also executed `beforeCommit` successfully.

Chapter 19. Data Import

For importing data using Cypher and CSV, see Section 12.8, “Importing CSV files with Cypher” [203].

For high-performance data import, see Chapter 29, *Import Tool* [515].

Chapter 20. 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*.

20.1. Introduction

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

- Javadocs[1]
- Download[2]
- Source code[3]

For examples, see Section 21.18, "Graph Algorithms" [383] (REST API) and Section 35.10, "Graph Algorithm examples" [613] (embedded database).

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`[4] Note that it should be used with the same version of `org.neo4j:neo4j-kernel`[5]. 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`[6] artifact which makes it simple to keep the versions in sync.

The starting point to find and use graph algorithms is `GraphAlgoFactory`[7] when using the Java Core API.

[1] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphalgo/package-summary.html
[2] http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-graph-algo%22
[3] https://github.com/neo4j/neo4j/tree/2.2.1/community/graph-algo
[4] http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-graph-algo%22
[5] http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-kernel%22
[6] http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j%22%20AND%20a%3A%22neo4j%22
[7] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphalgo/GraphAlgoFactory.html

Chapter 21. REST API

The Neo4j REST API is designed with discoverability in mind, so that you can start with a `GET` on the Section 21.4, “Service root” [301] 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[1], both for responses and for data sent with `POST`/`PUT` requests.

Below follows a listing of ways to interact with the REST API. For language bindings to the REST API, see Chapter 7, *Languages* [95].

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.

The server supports streaming results, with better performance and lower memory overhead. See Section 21.5, “Streaming” [302] for more information.

[1] http://www.json.org/

21.1. Transactional Cypher HTTP endpoint

The default way to interact with Neo4j is by using this endpoint.

The Neo4j transactional HTTP endpoint allows you to execute a series of Cypher statements within the scope of a transaction. The transaction may be kept open across multiple HTTP requests, until the client chooses to commit or roll back. Each HTTP request can include a list of statements, and for convenience you can include statements along with a request to begin or commit a transaction.

The server guards against orphaned transactions by using a timeout. If there are no requests for a given transaction within the timeout period, the server will roll it back. You can configure the timeout in the server configuration, by setting *org.neo4j.server.transaction.timeout* to the number of seconds before timeout. The default timeout is 60 seconds.

The key difference between the transactional HTTP endpoint for Cypher and the Cypher endpoint (see Section 21.6, "Legacy Cypher HTTP endpoint" [303]) is the ability to use the same transaction across multiple HTTP requests. The Cypher endpoint always attempts to commit a transaction at the end of each HTTP request. There has also been improvements to the serialization format.

Note

- Literal line breaks are not allowed inside Cypher statements.
- Open transactions are not shared among members of an HA cluster. Therefore, if you use this endpoint in an HA cluster, you must ensure that all requests for a given transaction are sent to the same Neo4j instance.
- Cypher queries with `USING PERIODIC COMMIT` (see Section 12.9, "Using Periodic Commit" [205]) may only be executed when creating a new transaction and immediately committing it with a single HTTP request (see the section called "Begin and commit a transaction in one request" [286] for how to do that).
- The serialization format for Cypher results is mostly the same as the Cypher endpoint. However, the format for raw entities is slightly less verbose and does not include hypermedia links.

Tip
In order to speed up queries in repeated scenarios, try not to use literals but replace them with parameters wherever possible. This will let the server cache query plans. See Section 8.5, "Parameters" [112] for more information.

Begin and commit a transaction in one request

If there is no need to keep a transaction open across multiple HTTP requests, you can begin a transaction, execute statements, and commit with just a single HTTP request.

Example request

- `POST http://localhost:7474/db/data/transaction/commit`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
{
  "statements" : [ {
    "statement" : "CREATE (n) RETURN id(n)"
  } ]
}
```

Example response

- `200: OK`

- Content-Type: application/json


```
{
  "results" : [ {
    "columns" : [ "id(n)" ],
    "data" : [ {
      "row" : [ 15 ]
    } ]
  } ],
  "errors" : [ ]
}
```


Begin a transaction

You begin a new transaction by posting zero or more Cypher statements to the transaction endpoint. The server will respond with the result of your statements, as well as the location of your open transaction.

Example request

- POST http://localhost:7474/db/data/transaction
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "statements" : [ {
    "statement" : "CREATE (n {props}) RETURN n",
    "parameters" : {
      "props" : {
        "name" : "My Node"
      }
    }
  } ]
}
```

Example response

- 201: Created
- Content-Type: application/json
- Location: http://localhost:7474/db/data/transaction/7


```
{
  "commit" : "http://localhost:7474/db/data/transaction/7/commit",
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ {
        "name" : "My Node"
      } ]
    } ]
  } ],
  "transaction" : {
    "expires" : "Mon, 13 Apr 2015 19:32:51 +0000"
  },
  "errors" : [ ]
}
```


Execute statements in an open transaction

Given that you have an open transaction, you can make a number of requests, each of which executes additional statements, and keeps the transaction open by resetting the transaction timeout.

Example request

- POST `http://localhost:7474/db/data/transaction/9`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "statements" : [ {
    "statement" : "CREATE (n) RETURN n"
  } ]
}
```

Example response

- 200: OK
- Content-Type: `application/json`

```
{
  "commit" : "http://localhost:7474/db/data/transaction/9/commit",
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ { } ]
    } ]
  } ],
  "transaction" : {
    "expires" : "Mon, 13 Apr 2015 19:32:51 +0000"
  },
  "errors" : [ ]
}
```

Execute statements in an open transaction in REST format for the return

Given that you have an open transaction, you can make a number of requests, each of which executes additional statements, and keeps the transaction open by resetting the transaction timeout. Specifying the `REST` format will give back full Neo4j Rest API representations of the Neo4j Nodes, Relationships and Paths, if returned.

Example request

- POST `http://localhost:7474/db/data/transaction/1`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "statements" : [ {
    "statement" : "CREATE (n) RETURN n",
    "resultDataContents" : [ "REST" ]
  } ]
}
```

Example response

- 200: OK
- Content-Type: `application/json`

```
{
  "commit" : "http://localhost:7474/db/data/transaction/1/commit",
  "results" : [ {
    "columns" : [ "n" ],
```

```
    "data" : [ {
      "rest" : [ {
        "labels" : "http://localhost:7474/db/data/node/12/labels",
        "outgoing_relationships" : "http://localhost:7474/db/data/node/12/relationships/out",
        "traverse" : "http://localhost:7474/db/data/node/12/traverse/{returnType}",
        "all_typed_relationships" : "http://localhost:7474/db/data/node/12/relationships/all/{-list|&|types}",
        "self" : "http://localhost:7474/db/data/node/12",
        "property" : "http://localhost:7474/db/data/node/12/properties/{key}",
        "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/12/relationships/out/{-list|&|types}",
        "properties" : "http://localhost:7474/db/data/node/12/properties",
        "incoming_relationships" : "http://localhost:7474/db/data/node/12/relationships/in",
        "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}",
        "metadata" : {
          "id" : 12,
          "labels" : [ ]
        },
        "data" : { }
      } ]
    } ]
  } ],
  "transaction" : {
    "expires" : "Mon, 13 Apr 2015 19:32:47 +0000"
  },
  "errors" : [ ]
}
```


Reset transaction timeout of an open transaction

Every orphaned transaction is automatically expired after a period of inactivity. This may be prevented by resetting the transaction timeout.

The timeout may be reset by sending a keep-alive request to the server that executes an empty list of statements. This request will reset the transaction timeout and return the new time at which the transaction will expire as an RFC1123 formatted timestamp value in the "transaction" section of the response.

Example request

- POST `http://localhost:7474/db/data/transaction/2`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`


```
{
  "statements" : [ ]
}
```


Example response

- 200: OK
- Content-Type: `application/json`


```
{
  "commit" : "http://localhost:7474/db/data/transaction/2/commit",
  "results" : [ ],
  "transaction" : {
    "expires" : "Mon, 13 Apr 2015 19:32:50 +0000"
  },
  "errors" : [ ]
}
```

Commit an open transaction

Given you have an open transaction, you can send a commit request. Optionally, you submit additional statements along with the request that will be executed before committing the transaction.

Example request

- POST `http://localhost:7474/db/data/transaction/4/commit`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "statements" : [ {
    "statement" : "CREATE (n) RETURN id(n)"
  } ]
}
```

Example response

- 200: OK
- Content-Type: `application/json`

```
{
  "results" : [ {
    "columns" : [ "id(n)" ],
    "data" : [ {
      "row" : [ 14 ]
    } ]
  } ],
  "errors" : [ ]
}
```

Rollback an open transaction

Given that you have an open transaction, you can send a rollback request. The server will rollback the transaction. Any further statements trying to run in this transaction will fail immediately.

Example request

- DELETE `http://localhost:7474/db/data/transaction/3`
- Accept: `application/json; charset=UTF-8`

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
{
  "results" : [ ],
  "errors" : [ ]
}
```

Return results in graph format

If you want to understand the graph structure of nodes and relationships returned by your query, you can specify the "graph" results data format. For example, this is useful when you want to visualise the graph structure. The format collates all the nodes and relationships from all columns of the result, and also flattens collections of nodes and relationships, including paths.

Example request

- POST `http://localhost:7474/db/data/transaction/commit`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "statements" : [ {
    "statement" : "CREATE ( bike:Bike { weight: 10 } ) CREATE ( frontWheel:Wheel { spokes: 3 } ) CREATE ( backWheel:Wheel
 { spokes: 32 } ) CREATE p1 = (bike)-[:HAS { position: 1 } ]->(frontWheel) CREATE p2 = (bike)-[:HAS { position: 2 } ]-
>(backWheel) RETURN bike, p1, p2",
    "resultDataContents" : [ "row", "graph" ]
  } ]
}
```

Example response

- 200: OK
- Content-Type: `application/json`

```
{
  "results" : [ {
    "columns" : [ "bike", "p1", "p2" ],
    "data" : [ {
      "row" : [ {
        "weight" : 10
      }, [ {
        "weight" : 10
      }, {
        "position" : 1
      }, {
        "spokes" : 3
      } ], [ {
        "weight" : 10
      }, {
        "position" : 2
      }, {
        "spokes" : 32
      } ] ],
      "graph" : {
        "nodes" : [ {
          "id" : "17",
          "labels" : [ "Wheel" ],
          "properties" : {
            "spokes" : 3
          }
        }, {
          "id" : "16",
          "labels" : [ "Bike" ],
          "properties" : {
            "weight" : 10
          }
        }, {
          "id" : "18",
          "labels" : [ "Wheel" ],
          "properties" : {
            "spokes" : 32
          }
        } ],
        "relationships" : [ {
          "id" : "9",
          "type" : "HAS",
          "startNode" : "16",
          "endNode" : "17",
          "properties" : {
```

```
            "position" : 1
          }
        }, {
          "id" : "10",
          "type" : "HAS",
          "startNode" : "16",
          "endNode" : "18",
          "properties" : {
            "position" : 2
          }
        } ]
      }
    } ]
  } ],
  "errors" : [ ]
}
```

Handling errors

The result of any request against the transaction endpoint is streamed back to the client. Therefore the server does not know whether the request will be successful or not when it sends the HTTP status code.

Because of this, all requests against the transactional endpoint will return 200 or 201 status code, regardless of whether statements were successfully executed. At the end of the response payload, the server includes a list of errors that occurred while executing statements. If this list is empty, the request completed successfully.

If any errors occur while executing statements, the server will roll back the transaction.

In this example, we send the server an invalid statement to demonstrate error handling.

For more information on the status codes, see Section 21.2, "Neo4j Status Codes" [293].

Example request

- POST `http://localhost:7474/db/data/transaction/8/commit`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
{
  "statements" : [ {
    "statement" : "This is not a valid Cypher Statement."
  } ]
}
```

Example response

- `200: OK`
- `Content-Type: application/json`

```
{
  "results" : [ ],
  "errors" : [ {
    "code" : "Neo.ClientError.Statement.InvalidSyntax",
    "message" : "Invalid input 'T': expected <init> (line 1, column 1 (offset: 0))\n\"This is not a valid Cypher Statement.
\"\n ^"
  } ]
}
```

21.2. Neo4j Status Codes

The transactional endpoint may in any response include zero or more status codes, indicating issues or information for the client. Each status code follows the same format: "Neo.[Classification].[Category].[Title]". The fact that a status code is returned by the server does always mean there is a fatal error. Status codes can also indicate transient problems that may go away if you retry the request.

What the effect of the status code is can be determined by its classification.

Note
This is not the same thing as HTTP status codes. Neo4j Status Codes are returned in the response body, at the very end of the response.

Classifications

Classification	Description	Effect on transaction
`ClientError`	The Client sent a bad request - changing the request might yield a successful outcome.	None
`DatabaseError`	The database failed to service the request.	Rollback
`TransientError`	The database cannot service the request right now, retrying later might yield a successful outcome.	None

Status codes

This is a complete list of all status codes Neo4j may return, and what they mean.

Status Code	Description
`Neo.ClientError.General.ReadOnly`	This is a read only database, writing or modifying the database is not allowed.
`Neo.ClientError.LegacyIndex.NoSuchIndex`	The request (directly or indirectly) referred to a index that does not exist.
`Neo.ClientError.Request.Invalid`	The client provided an invalid request.
`Neo.ClientError.Request.InvalidFormat`	The client provided a request that was missing required fields, or had values that are not allowed.
`Neo.ClientError.Schema.ConstraintAlreadyExists`	Unable to perform operation because it would clash with a pre-existing constraint.
`Neo.ClientError.Schema.ConstraintVerificationFailure`	Unable to create constraint because data that exists in the database violates it.
`Neo.ClientError.Schema.ConstraintViolation`	A constraint imposed by the database was violated.
`Neo.ClientError.Schema.IllegalTokenName`	A token name, such as a label, relationship type or property key, used is not valid. Tokens cannot be empty strings and cannot be null.
`Neo.ClientError.Schema.IndexAlreadyExists`	Unable to perform operation because it would clash with a pre-existing index.
`Neo.ClientError.Schema.IndexBelongsToConstraint`	A requested operation can not be performed on the specified index because the index is part of a constraint. If you want to drop the index, for instance, you must drop the constraint.

Status Code	Description
Neo.ClientError.Schema.IndexLimitReached	The maximum number of index entries supported has been reached, no more entities can be indexed.
Neo.ClientError.Schema.LabelLimitReached	The maximum number of labels supported has been reached, no more labels can be created.
Neo.ClientError.Schema.NoSuchConstraint	The request (directly or indirectly) referred to a constraint that does not exist.
Neo.ClientError.Schema.NoSuchIndex	The request (directly or indirectly) referred to an index that does not exist.
Neo.ClientError.Security.AuthenticationFailed	The client provided an incorrect username and/or password.
Neo.ClientError.Security.AuthenticationRateLimit	The client has provided incorrect authentication details too many times in a row.
Neo.ClientError.Security.AuthorizationFailed	The client does not have privileges to perform the operation requested.
Neo.ClientError.Statement.ArithmeticError	Invalid use of arithmetic, such as dividing by zero.
Neo.ClientError.Statement.ConstraintViolation	A constraint imposed by the statement is violated by the data in the database.
Neo.ClientError.Statement.EntityNotFound	The statement is directly referring to an entity that does not exist.
Neo.ClientError.Statement.InvalidArguments	The statement is attempting to perform operations using invalid arguments
Neo.ClientError.Statement.InvalidSemantics	The statement is syntactically valid, but expresses something that the database cannot do.
Neo.ClientError.Statement.InvalidSyntax	The statement contains invalid or unsupported syntax.
Neo.ClientError.Statement.InvalidType	The statement is attempting to perform operations on values with types that are not supported by the operation.
Neo.ClientError.Statement.NoSuchLabel	The statement is referring to a label that does not exist.
Neo.ClientError.Statement.NoSuchProperty	The statement is referring to a property that does not exist.
Neo.ClientError.Statement.ParameterMissing	The statement is referring to a parameter that was not provided in the request.
Neo.ClientError.Transaction.ConcurrentRequest	There were concurrent requests accessing the same transaction, which is not allowed.
Neo.ClientError.Transaction.EventHandlerThrewException	A transaction event handler threw an exception. The transaction will be rolled back.
Neo.ClientError.Transaction.HookFailed	Transaction hook failure.
Neo.ClientError.Transaction.InvalidType	The transaction is of the wrong type to service the request. For instance, a transaction that has had schema modifications performed in it cannot be used to subsequently perform data operations, and vice versa.

Status Code	Description
`Neo.ClientError.Transaction.MarkedAsFailed`	Transaction was marked as both successful and failed. Failure takes precedence and so this transaction was rolled back although it may have looked like it was going to be committed
`Neo.ClientError.Transaction.UnknownId`	The request referred to a transaction that does not exist.
`Neo.ClientError.Transaction.ValidationFailed`	Transaction changes did not pass validation checks
`Neo.DatabaseError.General.CorruptSchemaRule`	A malformed schema rule was encountered. Please contact your support representative.
`Neo.DatabaseError.General.FailedIndex`	The request (directly or indirectly) referred to an index that is in a failed state. The index needs to be dropped and recreated manually.
`Neo.DatabaseError.General.UnknownFailure`	An unknown failure occurred.
`Neo.DatabaseError.Schema.ConstraintCreationFailure`	Creating a requested constraint failed.
`Neo.DatabaseError.Schema.ConstraintDropFailure`	The database failed to drop a requested constraint.
`Neo.DatabaseError.Schema.IndexCreationFailure`	Failed to create an index.
`Neo.DatabaseError.Schema.IndexDropFailure`	The database failed to drop a requested index.
`Neo.DatabaseError.Schema.NoSuchLabel`	The request accessed a label that did not exist.
`Neo.DatabaseError.Schema.NoSuchPropertyKey`	The request accessed a property that does not exist.
`Neo.DatabaseError.Schema.NoSuchRelationshipType`	The request accessed a relationship type that does not exist.
`Neo.DatabaseError.Schema.NoSuchSchemaRule`	The request referred to a schema rule that does not exist.
`Neo.DatabaseError.Statement.ExecutionFailure`	The database was unable to execute the statement.
`Neo.DatabaseError.Transaction.CouldNotBegin`	The database was unable to start the transaction.
`Neo.DatabaseError.Transaction.CouldNotCommit`	The database was unable to commit the transaction.
`Neo.DatabaseError.Transaction.CouldNotRollback`	The database was unable to roll back the transaction.
`Neo.DatabaseError.Transaction.CouldNotWriteToLog`	The database was unable to write transaction to log.
`Neo.DatabaseError.Transaction.ReleaseLocksFailed`	The transaction was unable to release one or more of its locks.
`Neo.TransientError.General.DatabaseUnavailable`	The database is not currently available to serve your request, refer to the database logs for more details. Retrying your request at a later time may succeed.
`Neo.TransientError.Network.UnknownFailure`	An unknown network failure occurred, a retry may resolve the issue.

Status Code	Description
`Neo.TransientError.Schema.ModifiedConcurrently`	The database schema was modified while this transaction was running, the transaction should be retried.
`Neo.TransientError.Security.ModifiedConcurrently`	The user was modified concurrently to this request.
`Neo.TransientError.Statement.ExternalResourceFailure`	The external resource is not available
`Neo.TransientError.Transaction.AcquireLockTimeout`	The transaction was unable to acquire a lock, for instance due to a timeout or the transaction thread being interrupted.
`Neo.TransientError.Transaction.DeadlockDetected`	This transaction, and at least one more transaction, has acquired locks in a way that it will wait indefinitely, and the database has aborted it. Retrying this transaction will most likely be successful.

21.3. REST API Authentication and Authorization

In order to prevent unauthorized access to Neo4j, the REST API supports authorization and authentication. When enabled, requests to the REST API must be authorized using the username and password of a valid user. Authorization is enabled by default, see the section called "Server authentication and authorization" [497] for how to disable it.

When Neo4j is first installed you can authenticate with the default user `neo4j` and the default password `neo4j`. However, the default password must be changed (see the section called "User status and password changing" [298]) before access to resources will be permitted. This can easily be done via the Neo4j Browser, or via direct HTTP calls.

The username and password combination is local to each Neo4j instance. If you wish to have multiple instances in a cluster, you should ensure that all instances share the same credential. For automated deployments, you may also copy security configuration from another Neo4j instance (see the section called "Copying security configuration from one instance to another" [300]).

Authenticating

Missing authorization

If an `Authorization` header is not supplied, the server will reply with an error.

Example request

- `GET http://localhost:7474/db/data/`
- `Accept: application/json; charset=UTF-8`

Example response

- `401: Unauthorized`
- `Content-Type: application/json; charset=UTF-8`
- `WWW-Authenticate: None`

```
{
  "errors" : [ {
    "message" : "No authorization header supplied.",
    "code" : "Neo.ClientError.Security.AuthorizationFailed"
  } ]
}
```

Authenticate to access the server

Authenticate by sending a username and a password to Neo4j using HTTP Basic Auth. Requests should include an `Authorization` header, with a value of `Basic <payload>`, where "payload" is a base64 encoded string of "username:password".

Example request

- `GET http://localhost:7474/user/neo4j`
- `Accept: application/json; charset=UTF-8`
- `Authorization: Basic bmVvNGo6c2VjcmV0`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

```
{
```

```
  "username" : "neo4j",
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "password_change_required" : false
}
```

Incorrect authentication

If an incorrect username or password is provided, the server replies with an error.

Example request

- `POST http://localhost:7474/db/data/`
- `Accept: application/json; charset=UTF-8`
- `Authorization: Basic bmVvNGo6aW5jb3JyZWN0`

Example response

- `401: Unauthorized`
- `Content-Type: application/json; charset=UTF-8`
- `WWW-Authenticate: None`

```
{
  "errors" : [ {
    "message" : "Invalid username or password.",
    "code" : "Neo.ClientError.Security.AuthorizationFailed"
  } ]
}
```

Required password changes

In some cases, like the very first time Neo4j is accessed, the user will be required to choose a new password. The database will signal that a new password is required and deny access.

See the section called "User status and password changing" [298] for how to set a new password.

Example request

- `GET http://localhost:7474/db/data/`
- `Accept: application/json; charset=UTF-8`
- `Authorization: Basic bmVvNGo6bmVvNGo=`

Example response

- `403: Forbidden`
- `Content-Type: application/json; charset=UTF-8`

```
{
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "errors" : [ {
    "message" : "User is required to change their password.",
    "code" : "Neo.ClientError.Security.AuthorizationFailed"
  } ]
}
```

User status and password changing

User status

Given that you know the current password, you can ask the server for the user status.

Example request

- GET `http://localhost:7474/user/neo4j`
- Accept: `application/json; charset=UTF-8`
- Authorization: `Basic bmVvNGo6c2VjcmV0`

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
{
  "username" : "neo4j",
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "password_change_required" : false
}
```

User status on first access

On first access, and using the default password, the user status will indicate that the users password requires changing.

Example request

- GET `http://localhost:7474/user/neo4j`
- Accept: `application/json; charset=UTF-8`
- Authorization: `Basic bmVvNGo6bmVvNGo=`

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
{
  "username" : "neo4j",
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "password_change_required" : true
}
```

Changing the user password

Given that you know the current password, you can ask the server to change a users password. You can choose any password you like, as long as it is different from the current password.

Example request

- POST `http://localhost:7474/user/neo4j/password`
- Accept: `application/json; charset=UTF-8`
- Authorization: `Basic bmVvNGo6bmVvNGo=`
- Content-Type: `application/json`

```
{
  "password" : "secret"
}
```

Example response

- 200: OK

```
```

Access when auth is disabled

When auth is disabled

When auth has been disabled in the configuration, requests can be sent without an `Authorization` header.

Example request

- `GET http://localhost:7474/db/data/`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`


```
{
  "extensions" : { },
  "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",
  "extensions_info" : "http://localhost:7474/db/data/ext",
  "relationship_types" : "http://localhost:7474/db/data/relationship/types",
  "batch" : "http://localhost:7474/db/data/batch",
  "cypher" : "http://localhost:7474/db/data/cypher",
  "indexes" : "http://localhost:7474/db/data/schema/index",
  "constraints" : "http://localhost:7474/db/data/schema/constraint",
  "transaction" : "http://localhost:7474/db/data/transaction",
  "node_labels" : "http://localhost:7474/db/data/labels",
  "neo4j_version" : "2.2.1"
}
```


Copying security configuration from one instance to another

In many cases, such as automated deployments, you may want to start a Neo4j instance with pre-configured authentication and authorization. This is possible by copying the auth database file from a pre-existing Neo4j instance to your new instance.

This file is located at *data/dbms/auth*, and simply copying that file into a new Neo4j instance will transfer your password and authorization token.

21.4. Service root

Get service root

The service root is your starting point to discover the REST API. It contains the basic starting points for the database, and some version and extension information.

Figure 21.1. Final Graph

Example request

- GET http://localhost:7474/db/data/
- Accept: application/json; charset=UTF-8

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8


```
{
  "extensions" : { },
  "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",
  "extensions_info" : "http://localhost:7474/db/data/ext",
  "relationship_types" : "http://localhost:7474/db/data/relationship/types",
  "batch" : "http://localhost:7474/db/data/batch",
  "cypher" : "http://localhost:7474/db/data/cypher",
  "indexes" : "http://localhost:7474/db/data/schema/index",
  "constraints" : "http://localhost:7474/db/data/schema/constraint",
  "transaction" : "http://localhost:7474/db/data/transaction",
  "node_labels" : "http://localhost:7474/db/data/labels",
  "neo4j_version" : "2.2.1"
}
```

21.5. Streaming

All responses from the REST API can be transmitted as JSON streams, resulting in better performance and lower memory overhead on the server side. To use streaming, supply the header `X-Stream: true` with each request.

Figure 21.2. Final Graph

Example request

- GET `http://localhost:7474/db/data/`
- `Accept: application/json`
- `X-Stream: true`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8; stream=true`


```
{
  "extensions" : { },
  "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",
  "extensions_info" : "http://localhost:7474/db/data/ext",
  "relationship_types" : "http://localhost:7474/db/data/relationship/types",
  "batch" : "http://localhost:7474/db/data/batch",
  "cypher" : "http://localhost:7474/db/data/cypher",
  "indexes" : "http://localhost:7474/db/data/schema/index",
  "constraints" : "http://localhost:7474/db/data/schema/constraint",
  "transaction" : "http://localhost:7474/db/data/transaction",
  "node_labels" : "http://localhost:7474/db/data/labels",
  "neo4j_version" : "2.2.1"
}
```

21.6. Legacy Cypher HTTP endpoint

Note
This endpoint is deprecated. Please transition to using the new transactional endpoint (see Section 21.1, "Transactional Cypher HTTP endpoint" [286]). Among other things it allows you to run multiple Cypher statements in the same transaction.

The Neo4j REST API allows querying with Cypher, see Part III, "Cypher Query Language" [101]. 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`, `Path` or any simple value like `String`.

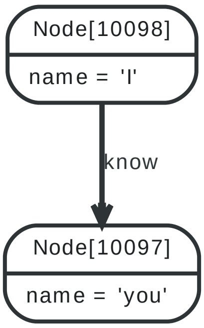
Tip
In order to speed up queries in repeated scenarios, try not to use literals but replace them with parameters wherever possible in order to let the server cache query plans, see the section called "Use parameters" [303] for details. Also see Section 8.5, "Parameters" [112] for where parameters can be used.

Use parameters

Cypher supports queries with parameters which are submitted as JSON.

```
MATCH (x { name: { startName }})-[r]-(friend)
WHERE friend.name = { name }
RETURN TYPE(r)
```

Figure 21.3. Final Graph



Example request

- `POST http://localhost:7474/db/data/cypher`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
{
  "query" : "MATCH (x {name: {startName}})-[r]-(friend) WHERE friend.name = {name} RETURN TYPE(r)",
  "params" : {
    "startName" : "I",
    "name" : "you"
  }
}
```

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

```
{
```

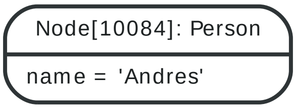
```
  "columns" : [ "TYPE(r)" ],
  "data" : [ [ "know" ] ]
}
```

Create a node

Create a node with a label and a property using Cypher. See the request for the parameter sent with the query.

```
CREATE (n:Person { name : { name }})
RETURN n
```

Figure 21.4. Final Graph



Example request

- POST `http://localhost:7474/db/data/cypher`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "query" : "CREATE (n:Person { name : {name} }) RETURN n",
  "params" : {
    "name" : "Andres"
  }
}
```

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

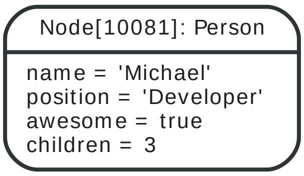
```
{
  "columns" : [ "n" ],
  "data" : [ [ {
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10084/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/10084/labels",
    "data" : {
      "name" : "Andres"
    },
    "traverse" : "http://localhost:7474/db/data/node/10084/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10084/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/10084",
    "property" : "http://localhost:7474/db/data/node/10084/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10084/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10084/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10084/relationships/in",
    "extensions" : { },
    "create_relationship" : "http://localhost:7474/db/data/node/10084/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/10084/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10084/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10084/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 10084,
      "labels" : [ "Person" ]
    }
  } ] ]
}
```

Create a node with multiple properties

Create a node with a label and multiple properties using Cypher. See the request for the parameter sent with the query.

```
CREATE (n:Person { props })
RETURN n
```

Figure 21.5. Final Graph



Example request

- POST `http://localhost:7474/db/data/cypher`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`


```
{
  "query" : "CREATE (n:Person { props } ) RETURN n",
  "params" : {
    "props" : {
      "position" : "Developer",
      "name" : "Michael",
      "awesome" : true,
      "children" : 3
    }
  }
}
```


Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`


```
{
  "columns" : [ "n" ],
  "data" : [ [ {
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10081/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/10081/labels",
    "data" : {
      "position" : "Developer",
      "awesome" : true,
      "name" : "Michael",
      "children" : 3
    },
    "traverse" : "http://localhost:7474/db/data/node/10081/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10081/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/10081",
    "property" : "http://localhost:7474/db/data/node/10081/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10081/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10081/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10081/relationships/in",
    "extensions" : { },
    "create_relationship" : "http://localhost:7474/db/data/node/10081/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/10081/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10081/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10081/relationships/in/{-list|&|types}",
    "metadata" : {
```

```
      "id" : 10081,
      "labels" : [ "Person" ]
    }
  } ] ]
}
```

Create multiple nodes with properties

Create multiple nodes with properties using Cypher. See the request for the parameter sent with the query.

```
CREATE (n:Person { props })
RETURN n
```

Figure 21.6. Final Graph

Example request

- POST http://localhost:7474/db/data/cypher
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "query" : "CREATE (n:Person { props } ) RETURN n",
  "params" : {
    "props" : [ {
      "name" : "Andres",
      "position" : "Developer"
    }, {
      "name" : "Michael",
      "position" : "Developer"
    } ]
  }
}
```

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

```
{
  "columns" : [ "n" ],
  "data" : [ [ {
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10085/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/10085/labels",
    "data" : {
      "position" : "Developer",
      "name" : "Andres"
    },
    "traverse" : "http://localhost:7474/db/data/node/10085/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10085/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/10085",
    "property" : "http://localhost:7474/db/data/node/10085/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10085/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10085/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10085/relationships/in",
    "extensions" : { },
    "create_relationship" : "http://localhost:7474/db/data/node/10085/relationships",
```

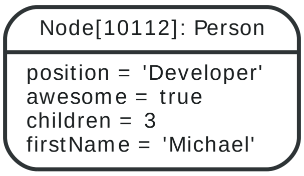
```
    "paged_traverse" : "http://localhost:7474/db/data/node/10085/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10085/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10085/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 10085,
      "labels" : [ "Person" ]
    }
  } ], [ {
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10086/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/10086/labels",
    "data" : {
      "position" : "Developer",
      "name" : "Michael"
    },
    "traverse" : "http://localhost:7474/db/data/node/10086/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10086/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/10086",
    "property" : "http://localhost:7474/db/data/node/10086/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10086/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10086/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10086/relationships/in",
    "extensions" : { },
    "create_relationship" : "http://localhost:7474/db/data/node/10086/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/10086/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10086/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10086/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 10086,
      "labels" : [ "Person" ]
    }
  } ] ]
}
```


Set all properties on a node using Cypher

Set all properties on a node.

```
CREATE (n:Person { name: 'this property is to be deleted' })
SET n = { props }
RETURN n
```

Figure 21.7. Final Graph



Example request

- POST `http://localhost:7474/db/data/cypher`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`


```
{
  "query" : "CREATE (n:Person { name: 'this property is to be deleted' } ) SET n = { props } RETURN n",
  "params" : {
    "props" : {
      "position" : "Developer",
      "firstName" : "Michael",
      "awesome" : true,
      "children" : 3
```

```
    }
  }
}
```

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

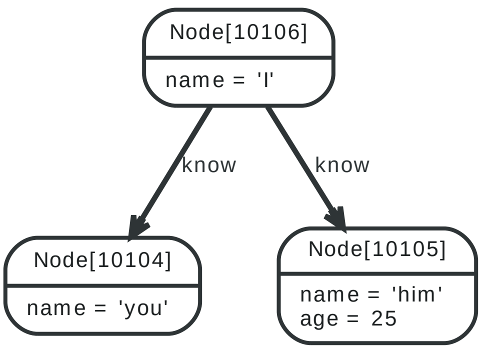

```
{
  "columns" : [ "n" ],
  "data" : [ [ {
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10112/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/10112/labels",
    "data" : {
      "position" : "Developer",
      "awesome" : true,
      "children" : 3,
      "firstName" : "Michael"
    },
    "traverse" : "http://localhost:7474/db/data/node/10112/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10112/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/10112",
    "property" : "http://localhost:7474/db/data/node/10112/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10112/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10112/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10112/relationships/in",
    "extensions" : { },
    "create_relationship" : "http://localhost:7474/db/data/node/10112/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/10112/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10112/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10112/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 10112,
      "labels" : [ "Person" ]
    }
  } ] ]
}
```


Send a query

A simple query returning all nodes connected to some node, returning the node and the name property, if it exists, otherwise `NULL`:

```
MATCH (x { name: 'I' })-[r]->(n)
RETURN type(r), n.name, n.age
```

Figure 21.8. Final Graph



Example request

- `POST http://localhost:7474/db/data/cypher`

- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "query" : "MATCH (x {name: 'I'})-[r]->(n) RETURN type(r), n.name, n.age",
  "params" : { }
}
```

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

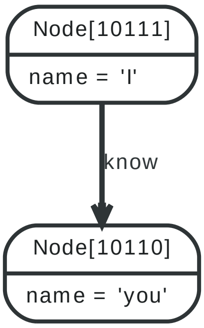
```
{
  "columns" : [ "type(r)", "n.name", "n.age" ],
  "data" : [ [ "know", "him", 25 ], [ "know", "you", null ] ]
}
```

Return paths

Paths can be returned just like other return types.

```
MATCH path =(x { name: 'I' })--(friend)
RETURN path, friend.name
```

Figure 21.9. Final Graph



Example request

- POST http://localhost:7474/db/data/cypher
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "query" : "MATCH path = (x {name: 'I'})--(friend) RETURN path, friend.name",
  "params" : { }
}
```

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

```
{
  "columns" : [ "path", "friend.name" ],
  "data" : [ [ {
    "directions" : [ "->" ],
    "start" : "http://localhost:7474/db/data/node/10111",
    "nodes" : [ "http://localhost:7474/db/data/node/10111", "http://localhost:7474/db/data/node/10110" ],
    "length" : 1,
    "relationships" : [ "http://localhost:7474/db/data/relationship/55" ],
```

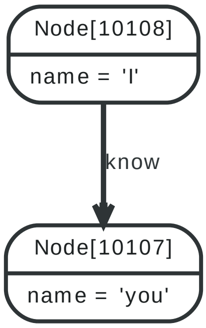
```
    "end" : "http://localhost:7474/db/data/node/10110"
  }, "you" ] ]
}
```

Nested results

When sending queries that return nested results like list and maps, these will get serialized into nested JSON representations according to their types.

```
MATCH (n)
WHERE n.name IN ['I', 'you']
RETURN collect(n.name)
```

Figure 21.10. Final Graph



Example request

- POST `http://localhost:7474/db/data/cypher`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "query" : "MATCH (n) WHERE n.name in ['I', 'you'] RETURN collect(n.name)",
  "params" : { }
}
```

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

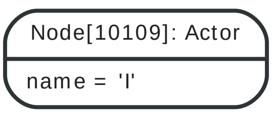
```
{
  "columns" : [ "collect(n.name)" ],
  "data" : [ [ [ "you", "I" ] ] ]
}
```

Retrieve query metadata

By passing in an additional GET parameter when you execute Cypher queries, metadata about the query will be returned, such as how many labels were added or removed by the query.

```
MATCH (n { name: 'I' })
SET n:Actor
REMOVE n:Director
RETURN labels(n)
```

Figure 21.11. Final Graph



Example request

- POST `http://localhost:7474/db/data/cypher?includeStats=true`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "query" : "MATCH (n {name: 'I'}) SET n:Actor REMOVE n:Director RETURN labels(n)",
  "params" : { }
}
```

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

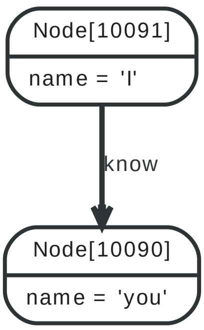
```
{
  "columns" : [ "labels(n)" ],
  "data" : [ [ [ "Actor" ] ] ],
  "stats" : {
    "relationships_created" : 0,
    "nodes_deleted" : 0,
    "relationship_deleted" : 0,
    "indexes_added" : 0,
    "properties_set" : 0,
    "constraints_removed" : 0,
    "indexes_removed" : 0,
    "labels_removed" : 1,
    "constraints_added" : 0,
    "labels_added" : 1,
    "nodes_created" : 0,
    "contains_updates" : true
  }
}
```

Errors

Errors on the server will be reported as a JSON-formatted message, exception name and stacktrace.

```
MATCH (x { name: 'I' })
RETURN x.dummy/0
```

Figure 21.12. Final Graph



Example request

- POST `http://localhost:7474/db/data/cypher`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
```

```
  "query" : "MATCH (x {name: 'I'}) RETURN x.dummy/0",
  "params" : { }
}
```

Example response

- 400: Bad Request
- Content-Type: application/json; charset=UTF-8


```
{
  "message": "/ by zero",
  "exception": "BadInputException",
  "fullname": "org.neo4j.server.rest.repr.BadInputException",
  "stackTrace": [

 "org.neo4j.server.rest.repr.RepresentationExceptionHandlingIterable.exceptionOnNext(RepresentationExceptionHandlingIterable.java:39)",
    "org.neo4j.helpers.collection.ExceptionHandlingIterable$1.next(ExceptionHandlingIterable.java:55)",
    "org.neo4j.helpers.collection.IteratorWrapper.next(IteratorWrapper.java:47)",
    "org.neo4j.server.rest.repr.ListRepresentation.serialize(ListRepresentation.java:64)",
    "org.neo4j.server.rest.repr.Serializer.serialize(Serializer.java:75)",
    "org.neo4j.server.rest.repr.MappingSerializer.putList(MappingSerializer.java:61)",
    "org.neo4j.server.rest.repr.CypherResultRepresentation.serialize(CypherResultRepresentation.java:58)",
    "org.neo4j.server.rest.repr.MappingRepresentation.serialize(MappingRepresentation.java:41)",
    "org.neo4j.server.rest.repr.OutputFormat.assemble(OutputFormat.java:242)",
    "org.neo4j.server.rest.repr.OutputFormat.formatRepresentation(OutputFormat.java:174)",
    "org.neo4j.server.rest.repr.OutputFormat.response(OutputFormat.java:157)",
    "org.neo4j.server.rest.repr.OutputFormat.ok(OutputFormat.java:73)",
    "org.neo4j.server.rest.web.CypherService.cypher(CypherService.java:127)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "java.lang.Thread.run(Thread.java:745)"
  ],
  "cause": {
    "message": "/ by zero",
    "errors": [
      {
        "message": "/ by zero",
        "code": "Neo.ClientError.Statement.ArithmeticError"
      }
    ],
    "cause": {
      "message": "/ by zero",
      "errors": [
        {
          "message": "/ by zero",
          "code": "Neo.ClientError.Statement.ArithmeticError"
        }
      ],
      "cause": {
        "message": "/ by zero",
        "errors": [
          {
            "message": "/ by zero",
            "code": "Neo.ClientError.Statement.ArithmeticError"
          }
        ],
        "exception": "ArithmeticException",
        "fullname": "org.neo4j.cypher.ArithmeticException",
        "stackTrace": [
          "org.neo4j.cypher.internal.compatibility.exceptionHandlerFor2_2$.arithmeticException(CompatibilityFor2_2.scala:56)",
          "org.neo4j.cypher.internal.compatibility.exceptionHandlerFor2_2$.arithmeticException(CompatibilityFor2_2.scala:53)",
          "org.neo4j.cypher.internal.compiler.v2_2.ArithmeticException.mapToPublic(CypherException.scala:104)",
          "org.neo4j.cypher.internal.compatibility.exceptionHandlerFor2_2$.runSafely(CompatibilityFor2_2.scala:110)",
          "org.neo4j.cypher.internal.compatibility.ExecutionResultWrapperFor2_2$$anon$1.next(CompatibilityFor2_2.scala:207)",
```

```
          "org.neo4j.cypher.internal.compatibility.ExecutionResultWrapperFor2_2$$anon$1.next(CompatibilityFor2_2.scala:202)",
          "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:214)",
          "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:50)",
          "org.neo4j.helpers.collection.ExceptionHandlingIterable$1.next(ExceptionHandlingIterable.java:53)",
          "org.neo4j.helpers.collection.IteratorWrapper.next(IteratorWrapper.java:47)",
          "org.neo4j.server.rest.repr.ListRepresentation.serialize(ListRepresentation.java:64)",
          "org.neo4j.server.rest.repr.Serializer.serialize(Serializer.java:75)",
          "org.neo4j.server.rest.repr.MappingSerializer.putList(MappingSerializer.java:61)",
          "org.neo4j.server.rest.repr.CypherResultRepresentation.serialize(CypherResultRepresentation.java:58)",
          "org.neo4j.server.rest.repr.MappingRepresentation.serialize(MappingRepresentation.java:41)",
          "org.neo4j.server.rest.repr.OutputFormat.assemble(OutputFormat.java:242)",
          "org.neo4j.server.rest.repr.OutputFormat.formatRepresentation(OutputFormat.java:174)",
          "org.neo4j.server.rest.repr.OutputFormat.response(OutputFormat.java:157)",
          "org.neo4j.server.rest.repr.OutputFormat.ok(OutputFormat.java:73)",
          "org.neo4j.server.rest.web.CypherService.cypher(CypherService.java:127)",
          "java.lang.reflect.Method.invoke(Method.java:606)",

"org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
          "java.lang.Thread.run(Thread.java:745)"
        ]
      },
      "exception": "QueryExecutionKernelException",
      "fullname": "org.neo4j.kernel.impl.query.QueryExecutionKernelException",
      "stackTrace": [
        "org.neo4j.cypher.javacompat.ExecutionResult.converted(ExecutionResult.java:344)",
        "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:218)",
        "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:50)",
        "org.neo4j.helpers.collection.ExceptionHandlingIterable$1.next(ExceptionHandlingIterable.java:53)",
        "org.neo4j.helpers.collection.IteratorWrapper.next(IteratorWrapper.java:47)",
        "org.neo4j.server.rest.repr.ListRepresentation.serialize(ListRepresentation.java:64)",
        "org.neo4j.server.rest.repr.Serializer.serialize(Serializer.java:75)",
        "org.neo4j.server.rest.repr.MappingSerializer.putList(MappingSerializer.java:61)",
        "org.neo4j.server.rest.repr.CypherResultRepresentation.serialize(CypherResultRepresentation.java:58)",
        "org.neo4j.server.rest.repr.MappingRepresentation.serialize(MappingRepresentation.java:41)",
        "org.neo4j.server.rest.repr.OutputFormat.assemble(OutputFormat.java:242)",
        "org.neo4j.server.rest.repr.OutputFormat.formatRepresentation(OutputFormat.java:174)",
        "org.neo4j.server.rest.repr.OutputFormat.response(OutputFormat.java:157)",
        "org.neo4j.server.rest.repr.OutputFormat.ok(OutputFormat.java:73)",
        "org.neo4j.server.rest.web.CypherService.cypher(CypherService.java:127)",
        "java.lang.reflect.Method.invoke(Method.java:606)",

"org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
        "java.lang.Thread.run(Thread.java:745)"
      ]
    },
    "exception": "QueryExecutionException",
    "fullname": "org.neo4j.graphdb.QueryExecutionException",
    "stackTrace": [
      "org.neo4j.kernel.impl.query.QueryExecutionKernelException.asUserException(QueryExecutionKernelException.java:35)",
      "org.neo4j.cypher.javacompat.ExecutionResult.converted(ExecutionResult.java:344)",
      "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:218)",
      "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:50)",
      "org.neo4j.helpers.collection.ExceptionHandlingIterable$1.next(ExceptionHandlingIterable.java:53)",
      "org.neo4j.helpers.collection.IteratorWrapper.next(IteratorWrapper.java:47)",
      "org.neo4j.server.rest.repr.ListRepresentation.serialize(ListRepresentation.java:64)",
      "org.neo4j.server.rest.repr.Serializer.serialize(Serializer.java:75)",
      "org.neo4j.server.rest.repr.MappingSerializer.putList(MappingSerializer.java:61)",
      "org.neo4j.server.rest.repr.CypherResultRepresentation.serialize(CypherResultRepresentation.java:58)",
      "org.neo4j.server.rest.repr.MappingRepresentation.serialize(MappingRepresentation.java:41)",
      "org.neo4j.server.rest.repr.OutputFormat.assemble(OutputFormat.java:242)",
      "org.neo4j.server.rest.repr.OutputFormat.formatRepresentation(OutputFormat.java:174)",
      "org.neo4j.server.rest.repr.OutputFormat.response(OutputFormat.java:157)",
      "org.neo4j.server.rest.repr.OutputFormat.ok(OutputFormat.java:73)",
      "org.neo4j.server.rest.web.CypherService.cypher(CypherService.java:127)",
      "java.lang.reflect.Method.invoke(Method.java:606)",
```

```
      "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
      "java.lang.Thread.run(Thread.java:745)"
    ]
  },
  "errors": [
    {
      "message": "/ by zero",
      "code": "Neo.ClientError.Request.InvalidFormat"
    }
  ]
}
```

21.7. Property values

The REST API allows setting properties on nodes and relationships through direct RESTful operations. However, there are restrictions as to what types of values can be used as property values. Allowed value types are as follows:

- Numbers: Both integer values, with capacity as Java's Long type, and floating points, with capacity as Java's Double.
- Booleans.
- Strings.
- Arrays of the basic types above.

Arrays

There are two important points to be made about array values. First, all values in the array must be of the same type. That means either all integers, all floats, all booleans or all strings. Mixing types is not currently supported.

Second, storing empty arrays is only possible given certain preconditions. Because the JSON transfer format does not contain type information for arrays, type is inferred from the values in the array. If the array is empty, the Neo4j Server cannot determine the type. In these cases, it will check if an array is already stored for the given property, and will use the stored array's type when storing the empty array. If no array exists already, the server will reject the request.

Property keys

You can list all property keys ever used in the database. This includes and property keys you have used, but deleted.

There is currently no way to tell which ones are in use and which ones are not, short of walking the entire set of properties in the database.

List all property keys

Example request

- `GET` `http://localhost:7474/db/data/propertykeys`
- `Accept:` `application/json; charset=UTF-8`

Example response

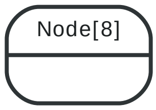
- `200:` `OK`
- `Content-Type:` `application/json; charset=UTF-8`

```
[ "name", "happy", "since", "non-existent", "cost", "name1", "name2" ]
```

21.8. Nodes

Create node

Figure 21.13. Final Graph



Example request

- POST `http://localhost:7474/db/data/node`
- Accept: `application/json; charset=UTF-8`

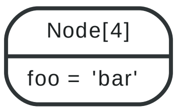
Example response

- 201: Created
- Content-Type: `application/json; charset=UTF-8`
- Location: `http://localhost:7474/db/data/node/8`

```
{
  "extensions" : { },
  "outgoing_relationships" : "http://localhost:7474/db/data/node/8/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/8/labels",
  "traverse" : "http://localhost:7474/db/data/node/8/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/8/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/8",
  "property" : "http://localhost:7474/db/data/node/8/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/8/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/8/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/8/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/8/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/8/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/8/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/8/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 8,
    "labels" : [ ]
  },
  "data" : { }
}
```

Create node with properties

Figure 21.14. Final Graph



Example request

- POST `http://localhost:7474/db/data/node`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "foo" : "bar"
```

```
}
```

Example response

- `201: Created`
- `Content-Length: 1209`
- `Content-Type: application/json; charset=UTF-8`
- `Location: http://localhost:7474/db/data/node/4`


```
{
  "extensions" : { },
  "outgoing_relationships" : "http://localhost:7474/db/data/node/4/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/4/labels",
  "traverse" : "http://localhost:7474/db/data/node/4/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/4/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/4",
  "property" : "http://localhost:7474/db/data/node/4/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/4/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/4/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/4/relationships/in",
  "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}",
  "metadata" : {
    "id" : 4,
    "labels" : [ ]
  },
  "data" : {
    "foo" : "bar"
  }
}
```


Get node

Note that the response contains URI/templates for the available operations for getting properties and relationships.

Figure 21.15. Final Graph

Example request

- GET `http://localhost:7474/db/data/node/10075`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`


```
{
  "extensions" : { },
  "outgoing_relationships" : "http://localhost:7474/db/data/node/10075/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/10075/labels",
  "traverse" : "http://localhost:7474/db/data/node/10075/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/10075/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/10075",
```

```
  "property" : "http://localhost:7474/db/data/node/10075/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/10075/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10075/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/10075/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/10075/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/10075/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/10075/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10075/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 10075,
    "labels" : [ ]
  },
  "data" : { }
}
```


Get non-existent node

Figure 21.16. Final Graph

Example request

- GET `http://localhost:7474/db/data/node/1007900000`
- `Accept: application/json; charset=UTF-8`

Example response

- `404: Not Found`
- `Content-Type: application/json; charset=UTF-8`


```
{
  "message": "Cannot find node with id [1007900000] in database.",
  "exception": "NodeNotFoundException",
  "fullname": "org.neo4j.server.rest.web.NodeNotFoundException",
  "stackTrace": [
    "org.neo4j.server.rest.web.DatabaseActions.node(DatabaseActions.java:176)",
    "org.neo4j.server.rest.web.DatabaseActions.getNode(DatabaseActions.java:221)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.getNode(RestfulGraphDatabase.java:266)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "java.lang.Thread.run(Thread.java:745)"
  ],
  "cause": {
    "message": "Node 1007900000 not found",
    "errors": [
      {
        "message": "Node 1007900000 not found",
        "code": "Neo.ClientError.Statement.EntityNotFound"
      }
    ],
    "cause": {
      "message": "Unable to load NODE with id 1007900000.",
      "errors": [
        {
          "message": "Unable to load NODE with id 1007900000.",
          "code": "Neo.ClientError.Statement.EntityNotFound"
        }
      ],
      "exception": "EntityNotFoundException",
      "fullname": "org.neo4j.kernel.api.exceptions.EntityNotFoundException",
```

```
      "stackTrace": [
        "org.neo4j.kernel.InternalAbstractGraphDatabase.getNodeById(InternalAbstractGraphDatabase.java:1066)",
        "org.neo4j.server.rest.web.DatabaseActions.node(DatabaseActions.java:172)",
        "org.neo4j.server.rest.web.DatabaseActions.getNode(DatabaseActions.java:221)",
        "org.neo4j.server.rest.web.RestfulGraphDatabase.getNode(RestfulGraphDatabase.java:266)",
        "java.lang.reflect.Method.invoke(Method.java:606)",

 "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
        "java.lang.Thread.run(Thread.java:745)"
      ]
    },
    "exception": "NotFoundException",
    "fullname": "org.neo4j.graphdb.NotFoundException",
    "stackTrace": [
      "org.neo4j.kernel.InternalAbstractGraphDatabase.getNodeById(InternalAbstractGraphDatabase.java:1066)",
      "org.neo4j.server.rest.web.DatabaseActions.node(DatabaseActions.java:172)",
      "org.neo4j.server.rest.web.DatabaseActions.getNode(DatabaseActions.java:221)",
      "org.neo4j.server.rest.web.RestfulGraphDatabase.getNode(RestfulGraphDatabase.java:266)",
      "java.lang.reflect.Method.invoke(Method.java:606)",
      "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
      "java.lang.Thread.run(Thread.java:745)"
    ]
  },
  "errors": [
    {
      "message": "Cannot find node with id [1007900000] in database.",
      "code": "Neo.ClientError.Statement.EntityNotFound"
    }
  ]
}
```

Delete node

Figure 21.17. Final Graph

Example request

- DELETE `http://localhost:7474/db/data/node/5`
- `Accept: application/json; charset=UTF-8`

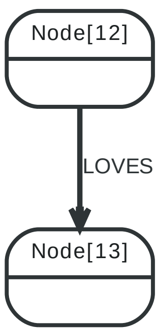
Example response

- `204: No Content`

Nodes with relationships cannot be deleted

The relationships on a node has to be deleted before the node can be deleted.

Figure 21.18. Final Graph



Example request

- DELETE `http://localhost:7474/db/data/node/12`
- Accept: `application/json; charset=UTF-8`

Example response

- 409: `Conflict`
- Content-Type: `application/json; charset=UTF-8`


```
{
  "message": "The node with id 12 cannot be deleted. Check that the node is orphaned before deletion.",
  "exception": "ConstraintViolationException",
  "fullname": "org.neo4j.graphdb.ConstraintViolationException",
  "stackTrace": [
    "org.neo4j.server.rest.web.DatabaseActions.deleteNode(DatabaseActions.java:230)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteNode(RestfulGraphDatabase.java:280)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "java.lang.Thread.run(Thread.java:745)"
  ],
  "errors": [
    {
      "message": "The node with id 12 cannot be deleted. Check that the node is orphaned before deletion.",
      "code": "Neo.ClientError.Schema.ConstraintViolation"
    }
  ]
}
```

21.9. 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.

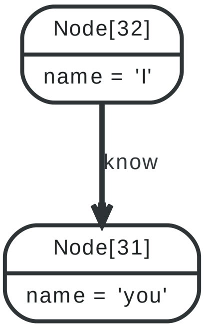
The general pattern to get relationships from a node is:

```
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.

Get Relationship by ID

Figure 21.19. Final Graph



Example request

- `GET http://localhost:7474/db/data/relationship/13`
- `Accept: application/json; charset=UTF-8`

Example response

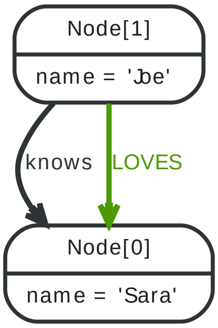
- `200: OK`
- `Content-Type: application/json; charset=UTF-8`


```
{
  "extensions" : { },
  "start" : "http://localhost:7474/db/data/node/32",
  "property" : "http://localhost:7474/db/data/relationship/13/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/13",
  "properties" : "http://localhost:7474/db/data/relationship/13/properties",
  "type" : "know",
  "end" : "http://localhost:7474/db/data/node/31",
  "metadata" : {
    "id" : 13,
    "type" : "know"
  },
  "data" : { }
}
```


Create relationship

Upon successful creation of a relationship, the new relationship is returned.

Figure 21.20. Final Graph



Example request

- `POST http://localhost:7474/db/data/node/1/relationships`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
{
  "to" : "http://localhost:7474/db/data/node/0",
  "type" : "LOVES"
}
```

Example response

- `201: Created`
- `Content-Type: application/json; charset=UTF-8`
- `Location: http://localhost:7474/db/data/relationship/1`

```
{
  "extensions" : { },
  "start" : "http://localhost:7474/db/data/node/1",
  "property" : "http://localhost:7474/db/data/relationship/1/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/1",
  "properties" : "http://localhost:7474/db/data/relationship/1/properties",
  "type" : "LOVES",
  "end" : "http://localhost:7474/db/data/node/0",
  "metadata" : {
    "id" : 1,
    "type" : "LOVES"
  },
  "data" : { }
}
```

Create a relationship with properties

Upon successful creation of a relationship, the new relationship is returned.

Figure 21.21. Starting Graph

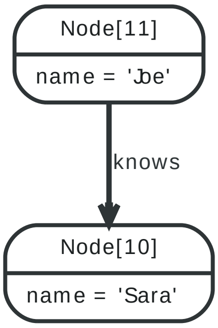
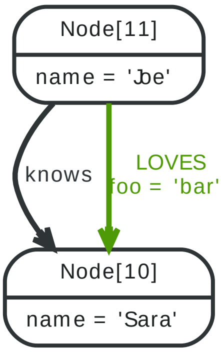


Figure 21.22. Final Graph



Example request

- POST http://localhost:7474/db/data/node/11/relationships
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "to" : "http://localhost:7474/db/data/node/10",
  "type" : "LOVES",
  "data" : {
    "foo" : "bar"
  }
}
```

Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8
- Location: http://localhost:7474/db/data/relationship/8

```
{
  "extensions" : { },
  "start" : "http://localhost:7474/db/data/node/11",
  "property" : "http://localhost:7474/db/data/relationship/8/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/8",
  "properties" : "http://localhost:7474/db/data/relationship/8/properties",
  "type" : "LOVES",
  "end" : "http://localhost:7474/db/data/node/10",
  "metadata" : {
    "id" : 8,
    "type" : "LOVES"
  },
  "data" : {
    "foo" : "bar"
  }
}
```

Delete relationship

Figure 21.23. Starting Graph

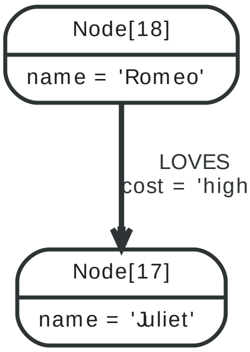


Figure 21.24. Final Graph

Example request

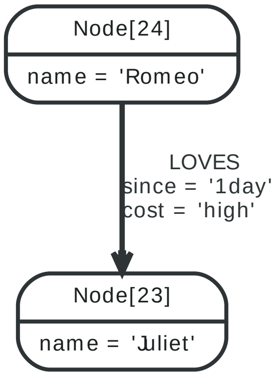
- DELETE `http://localhost:7474/db/data/relationship/6`
- `Accept: application/json; charset=UTF-8`

Example response

- `204: No Content`

Get all properties on a relationship

Figure 21.25. Final Graph



Example request

- GET `http://localhost:7474/db/data/relationship/9/properties`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

```
{
  "cost" : "high",
  "since" : "1day"
}
```

Set all properties on a relationship

Figure 21.26. Starting Graph

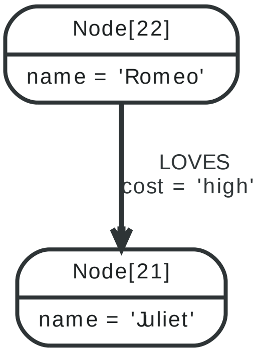
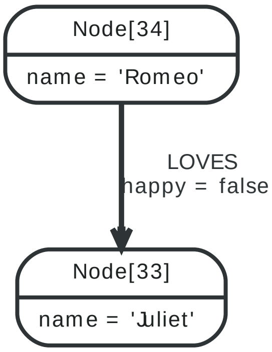


Figure 21.27. Final Graph



Example request

- PUT `http://localhost:7474/db/data/relationship/14/properties`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

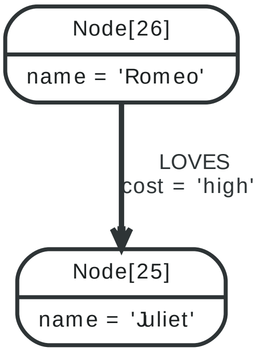
```
{
  "happy" : false
}
```

Example response

- `204: No Content`

Get single property on a relationship

Figure 21.28. Final Graph



Example request

- GET `http://localhost:7474/db/data/relationship/10/properties/cost`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

```
"high"
```

Set single property on a relationship

Figure 21.29. Starting Graph

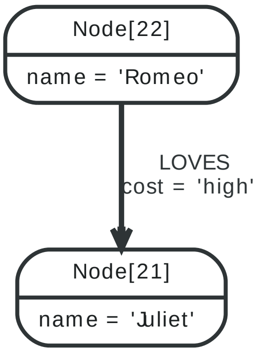
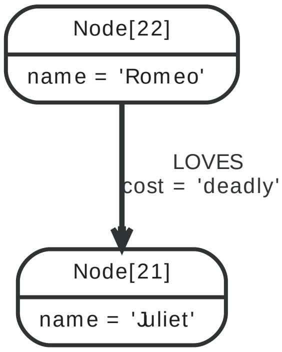


Figure 21.30. Final Graph



Example request

- PUT http://localhost:7474/db/data/relationship/8/properties/cost
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

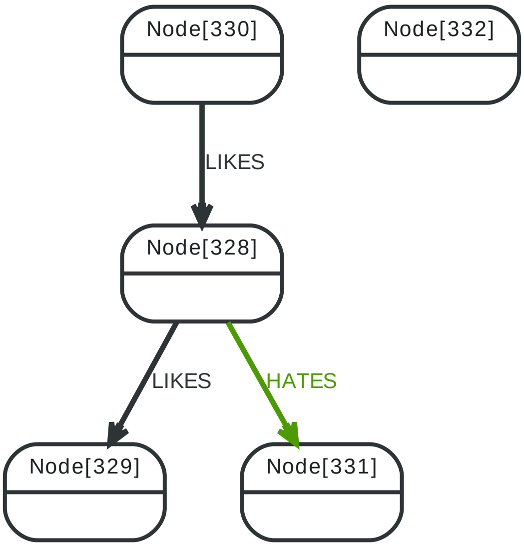
```
"deadly"
```

Example response

- 204: No Content

Get all relationships

Figure 21.31. Final Graph



Example request

- GET http://localhost:7474/db/data/node/328/relationships/all
- Accept: application/json; charset=UTF-8

Example response

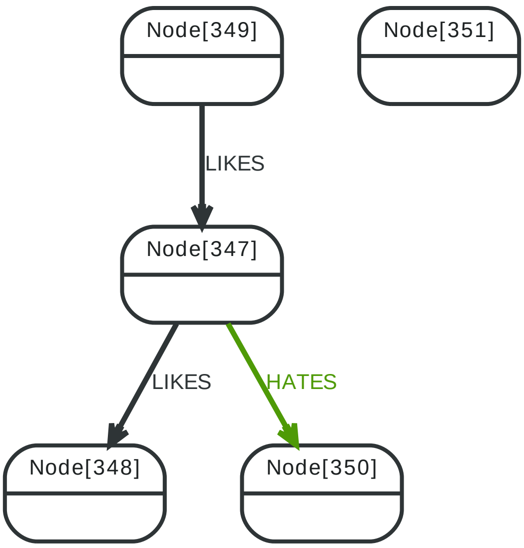
- 200: OK
- Content-Type: application/json; charset=UTF-8


```
[ {
  "start" : "http://localhost:7474/db/data/node/328",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/203",
  "property" : "http://localhost:7474/db/data/relationship/203/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/203/properties",
  "type" : "LIKES",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/329",
  "metadata" : {
    "id" : 203,
    "type" : "LIKES"
  }
}, {
  "start" : "http://localhost:7474/db/data/node/330",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/204",
```

```
  "property" : "http://localhost:7474/db/data/relationship/204/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/204/properties",
  "type" : "LIKES",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/328",
  "metadata" : {
    "id" : 204,
    "type" : "LIKES"
  }
}, {
  "start" : "http://localhost:7474/db/data/node/328",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/205",
  "property" : "http://localhost:7474/db/data/relationship/205/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/205/properties",
  "type" : "HATES",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/331",
  "metadata" : {
    "id" : 205,
    "type" : "HATES"
  }
} ]
```


Get incoming relationships

Figure 21.32. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/347/relationships/in`
- `Accept: application/json; charset=UTF-8`

Example response

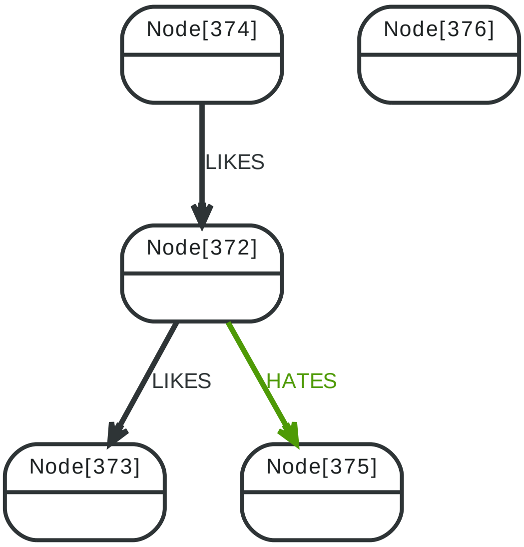
- `200: OK`
- `Content-Type: application/json; charset=UTF-8`


```
[ {
  "start" : "http://localhost:7474/db/data/node/349",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/215",
  "property" : "http://localhost:7474/db/data/relationship/215/properties/{key}",
```

```
  "properties" : "http://localhost:7474/db/data/relationship/215/properties",
  "type" : "LIKES",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/347",
  "metadata" : {
    "id" : 215,
    "type" : "LIKES"
  }
} ]
```


Get outgoing relationships

Figure 21.33. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/372/relationships/out`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

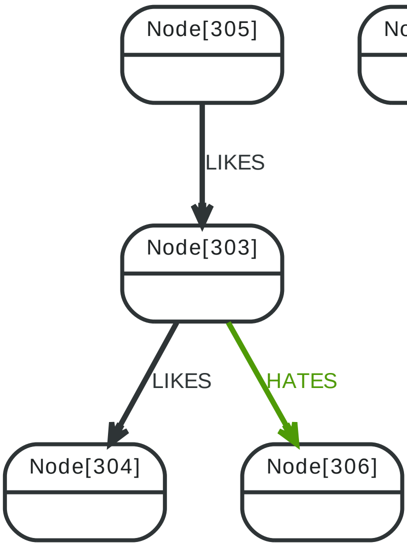

```
[ {
  "start" : "http://localhost:7474/db/data/node/372",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/229",
  "property" : "http://localhost:7474/db/data/relationship/229/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/229/properties",
  "type" : "LIKES",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/373",
  "metadata" : {
    "id" : 229,
    "type" : "LIKES"
  }
}, {
  "start" : "http://localhost:7474/db/data/node/372",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/231",
  "property" : "http://localhost:7474/db/data/relationship/231/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/231/properties",
```

```
  "type" : "HATES",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/375",
  "metadata" : {
    "id" : 231,
    "type" : "HATES"
  }
} ]
```


Get typed relationships

Note that the "&" needs to be encoded like "%26" for example when using cURL[2] from the terminal.

Figure 21.34. Final Graph



Example request

- `GET http://localhost:7474/db/data/node/303/relationships/all/LIKES&HATES`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

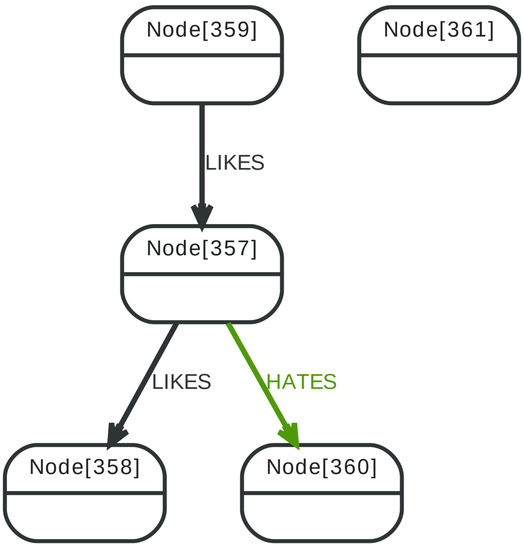

```
[ {
  "start" : "http://localhost:7474/db/data/node/303",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/188",
  "property" : "http://localhost:7474/db/data/relationship/188/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/188/properties",
  "type" : "LIKES",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/304",
  "metadata" : {
    "id" : 188,
    "type" : "LIKES"
  }
}, {
  "start" : "http://localhost:7474/db/data/node/305",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/189",
```


[2] http://curl.haxx.se/

```
  "property" : "http://localhost:7474/db/data/relationship/189/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/189/properties",
  "type" : "LIKES",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/303",
  "metadata" : {
    "id" : 189,
    "type" : "LIKES"
  }
}, {
  "start" : "http://localhost:7474/db/data/node/303",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/190",
  "property" : "http://localhost:7474/db/data/relationship/190/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/190/properties",
  "type" : "HATES",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/306",
  "metadata" : {
    "id" : 190,
    "type" : "HATES"
  }
} ]
```


Get relationships on a node without relationships

Figure 21.35. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/361/relationships/all`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`


```
[ ]
```

21.10. Relationship types

Get relationship types

Example request

- GET http://localhost:7474/db/data/relationship/types
- Accept: application/json; charset=UTF-8

Example response

- 200: OK
- Content-Type: application/json

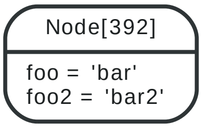
```
[ "know", "LOVES", "KNOWS" ]
```

21.11. Node properties

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 21.36. Final Graph



Example request

- PUT `http://localhost:7474/db/data/node/392/properties/foo`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
"bar"
```

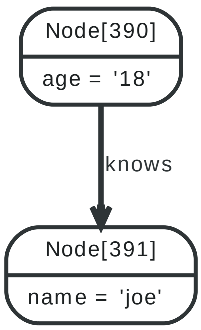
Example response

- `204: No Content`

Update node properties

This will replace all existing properties on the node with the new set of attributes.

Figure 21.37. Final Graph



Example request

- PUT `http://localhost:7474/db/data/node/390/properties`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

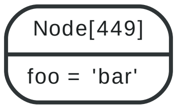
```
{
  "age" : "18"
}
```

Example response

- `204: No Content`

Get properties for node

Figure 21.38. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/449/properties`
- `Accept: application/json; charset=UTF-8`

Example response

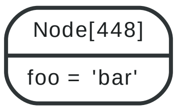
- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

```
{
  "foo" : "bar"
}
```

Get property for node

Get a single node property from a node.

Figure 21.39. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/448/properties/foo`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

```
"bar"
```

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; charset=UTF-8`
- `Content-Type: application/json`

```
{
  "foo" : null
}
```

Example response

- 400: Bad Request
- Content-Type: application/json; charset=UTF-8

```
{
  "message": "Could not set property \"foo\", unsupported type: null",
  "exception": "PropertyValueException",
  "fullname": "org.neo4j.server.rest.web.PropertyValueException",
  "stackTrace": [
    "org.neo4j.server.rest.domain.PropertySettingStrategy.setProperty(PropertySettingStrategy.java:141)",
    "org.neo4j.server.rest.domain.PropertySettingStrategy.setProperties(PropertySettingStrategy.java:88)",
    "org.neo4j.server.rest.web.DatabaseActions.createNode(DatabaseActions.java:207)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.createNode(RestfulGraphDatabase.java:239)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "java.lang.Thread.run(Thread.java:745)"
  ],
  "errors": [
    {
      "message": "Could not set property \"foo\", unsupported type: null",
      "code": "Neo.ClientError.Statement.InvalidArguments"
    }
  ]
}
```

Property values can not be nested

Nesting properties is not supported. You could for example store the nested JSON as a string instead.

Example request

- POST http://localhost:7474/db/data/node/
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "foo" : {
    "bar" : "baz"
  }
}
```

Example response

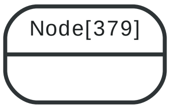
- 400: Bad Request
- Content-Type: application/json; charset=UTF-8

```
{
  "message": "Could not set property \"foo\", unsupported type: {bar\u003dbaz}",
  "exception": "PropertyValueException",
  "fullname": "org.neo4j.server.rest.web.PropertyValueException",
  "stackTrace": [
    "org.neo4j.server.rest.domain.PropertySettingStrategy.setProperty(PropertySettingStrategy.java:141)",
    "org.neo4j.server.rest.domain.PropertySettingStrategy.setProperties(PropertySettingStrategy.java:88)",
    "org.neo4j.server.rest.web.DatabaseActions.createNode(DatabaseActions.java:207)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.createNode(RestfulGraphDatabase.java:239)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "java.lang.Thread.run(Thread.java:745)"
  ],
  "errors": [
    {
      "message": "Could not set property \"foo\", unsupported type: {bar\u003dbaz}",
      "code": "Neo.ClientError.Statement.InvalidArguments"
```

```
    }
  ]
}
```

Delete all properties from node

Figure 21.40. Final Graph



Example request

- DELETE `http://localhost:7474/db/data/node/379/properties`
- `Accept: application/json; charset=UTF-8`

Example response

- `204: No Content`

Delete a named property from a node

To delete a single property from a node, see the example below.

Figure 21.41. Starting Graph

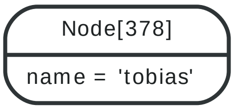
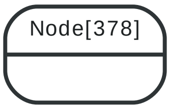


Figure 21.42. Final Graph



Example request

- DELETE `http://localhost:7474/db/data/node/378/properties/name`
- `Accept: application/json; charset=UTF-8`

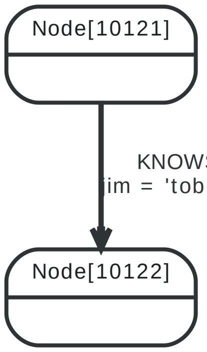
Example response

- `204: No Content`

21.12. Relationship properties

Update relationship properties

Figure 21.43. Final Graph



Example request

- PUT `http://localhost:7474/db/data/relationship/60/properties`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

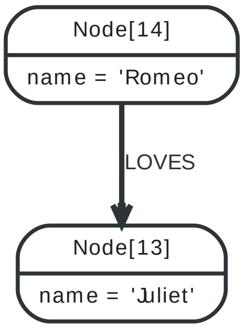
```
{
  "jim" : "tobias"
}
```

Example response

- `204: No Content`

Remove properties from a relationship

Figure 21.44. Final Graph



Example request

- DELETE `http://localhost:7474/db/data/relationship/4/properties`
- Accept: `application/json; charset=UTF-8`

Example response

- `204: No Content`

Remove property from a relationship

See the example request below.

Figure 21.45. Starting Graph

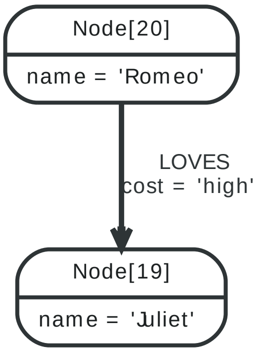
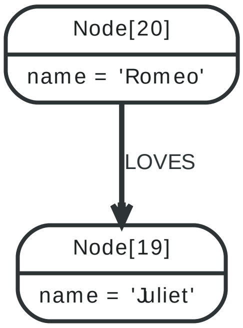


Figure 21.46. Final Graph



Example request

- `DELETE http://localhost:7474/db/data/relationship/7/properties/cost`
- `Accept: application/json; charset=UTF-8`

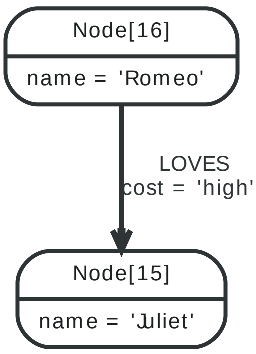
Example response

- `204: No Content`

Remove non-existent property from a relationship

Attempting to remove a property that doesn't exist results in an error.

Figure 21.47. Final Graph



Example request

- `DELETE http://localhost:7474/db/data/relationship/5/properties/non-existent`
- `Accept: application/json; charset=UTF-8`

Example response

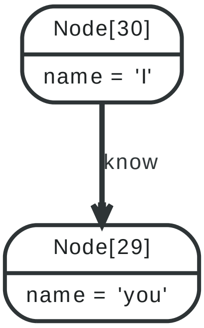
- 404: Not Found
- Content-Type: application/json; charset=UTF-8


```
{
  "message": "Relationship[5] does not have a property \"non-existent\"",
  "exception": "NoSuchPropertyException",
  "fullname": "org.neo4j.server.rest.web.NoSuchPropertyException",
  "stackTrace": [
    "org.neo4j.server.rest.web.DatabaseActions.removeRelationshipProperty(DatabaseActions.java:672)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteRelationshipProperty(RestfulGraphDatabase.java:801)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "java.lang.Thread.run(Thread.java:745)"
  ],
  "errors": [
    {
      "message": "Relationship[5] does not have a property \"non-existent\"",
      "code": "Neo.ClientError.Statement.NoSuchProperty"
    }
  ]
}
```


Remove properties from a non-existing relationship

Attempting to remove all properties from a relationship which doesn't exist results in an error.

Figure 21.48. Final Graph



Example request

- DELETE http://localhost:7474/db/data/relationship/1234/properties
- Accept: application/json; charset=UTF-8

Example response

- 404: Not Found
- Content-Type: application/json; charset=UTF-8

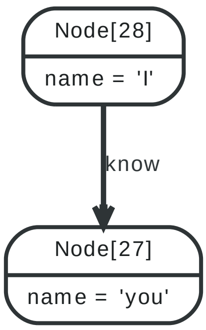

```
{
  "message": "org.neo4j.graphdb.NotFoundException: Relationship 1234 not found",
  "exception": "RelationshipNotFoundException",
  "fullname": "org.neo4j.server.rest.web.RelationshipNotFoundException",
  "stackTrace": [
    "org.neo4j.server.rest.web.DatabaseActions.relationship(DatabaseActions.java:190)",
    "org.neo4j.server.rest.web.DatabaseActions.removeAllRelationshipProperties(DatabaseActions.java:662)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteAllRelationshipProperties(RestfulGraphDatabase.java:781)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "java.lang.Thread.run(Thread.java:745)"
  ],
  "cause": {
```

```
    "message": "Relationship 1234 not found",
    "errors": [
      {
        "message": "Relationship 1234 not found",
        "code": "Neo.ClientError.Statement.EntityNotFound"
      }
    ],
    "cause": {
      "message": "Unable to load RELATIONSHIP with id 1234.",
      "errors": [
        {
          "message": "Unable to load RELATIONSHIP with id 1234.",
          "code": "Neo.ClientError.Statement.EntityNotFound"
        }
      ],
      "exception": "EntityNotFoundException",
      "fullname": "org.neo4j.kernel.api.exceptions.EntityNotFoundException",
      "stackTrace": [
        "org.neo4j.kernel.InternalAbstractGraphDatabase.getRelationshipById(InternalAbstractGraphDatabase.java:1086)",
        "org.neo4j.server.rest.web.DatabaseActions.relationship(DatabaseActions.java:186)",
        "org.neo4j.server.rest.web.DatabaseActions.removeAllRelationshipProperties(DatabaseActions.java:662)",
        "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteAllRelationshipProperties(RestfulGraphDatabase.java:781)",
        "java.lang.reflect.Method.invoke(Method.java:606)",

 "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
        "java.lang.Thread.run(Thread.java:745)"
      ]
    },
    "exception": "NotFoundException",
    "fullname": "org.neo4j.graphdb.NotFoundException",
    "stackTrace": [
      "org.neo4j.kernel.InternalAbstractGraphDatabase.getRelationshipById(InternalAbstractGraphDatabase.java:1086)",
      "org.neo4j.server.rest.web.DatabaseActions.relationship(DatabaseActions.java:186)",
      "org.neo4j.server.rest.web.DatabaseActions.removeAllRelationshipProperties(DatabaseActions.java:662)",
      "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteAllRelationshipProperties(RestfulGraphDatabase.java:781)",
      "java.lang.reflect.Method.invoke(Method.java:606)",
      "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
      "java.lang.Thread.run(Thread.java:745)"
    ]
  },
  "errors": [
    {
      "message": "org.neo4j.graphdb.NotFoundException: Relationship 1234 not found",
      "code": "Neo.ClientError.Statement.EntityNotFound"
    }
  ]
}
```


Remove property from a non-existing relationship

Attempting to remove a property from a relationship which doesn't exist results in an error.

Figure 21.49. Final Graph



Example request

- DELETE `http://localhost:7474/db/data/relationship/1234/properties/cost`
- Accept: `application/json; charset=UTF-8`

Example response

- 404: Not Found
- Content-Type: `application/json; charset=UTF-8`

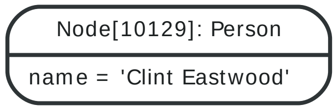

```
{
  "message": "org.neo4j.graphdb.NotFoundException: Relationship 1234 not found",
  "exception": "RelationshipNotFoundException",
  "fullname": "org.neo4j.server.rest.web.RelationshipNotFoundException",
  "stackTrace": [
    "org.neo4j.server.rest.web.DatabaseActions.relationship(DatabaseActions.java:190)",
    "org.neo4j.server.rest.web.DatabaseActions.removeRelationshipProperty(DatabaseActions.java:668)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteRelationshipProperty(RestfulGraphDatabase.java:801)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "java.lang.Thread.run(Thread.java:745)"
  ],
  "cause": {
    "message": "Relationship 1234 not found",
    "errors": [
      {
        "message": "Relationship 1234 not found",
        "code": "Neo.ClientError.Statement.EntityNotFound"
      }
    ],
    "cause": {
      "message": "Unable to load RELATIONSHIP with id 1234.",
      "errors": [
        {
          "message": "Unable to load RELATIONSHIP with id 1234.",
          "code": "Neo.ClientError.Statement.EntityNotFound"
        }
      ],
      "exception": "EntityNotFoundException",
      "fullname": "org.neo4j.kernel.api.exceptions.EntityNotFoundException",
      "stackTrace": [
        "org.neo4j.kernel.InternalAbstractGraphDatabase.getRelationshipById(InternalAbstractGraphDatabase.java:1086)",
        "org.neo4j.server.rest.web.DatabaseActions.relationship(DatabaseActions.java:186)",
        "org.neo4j.server.rest.web.DatabaseActions.removeRelationshipProperty(DatabaseActions.java:668)",
        "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteRelationshipProperty(RestfulGraphDatabase.java:801)",
        "java.lang.reflect.Method.invoke(Method.java:606)",

 "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
        "java.lang.Thread.run(Thread.java:745)"
      ]
    },
    "exception": "NotFoundException",
    "fullname": "org.neo4j.graphdb.NotFoundException",
    "stackTrace": [
      "org.neo4j.kernel.InternalAbstractGraphDatabase.getRelationshipById(InternalAbstractGraphDatabase.java:1086)",
      "org.neo4j.server.rest.web.DatabaseActions.relationship(DatabaseActions.java:186)",
      "org.neo4j.server.rest.web.DatabaseActions.removeRelationshipProperty(DatabaseActions.java:668)",
      "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteRelationshipProperty(RestfulGraphDatabase.java:801)",
      "java.lang.reflect.Method.invoke(Method.java:606)",
      "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
      "java.lang.Thread.run(Thread.java:745)"
    ]
  },
```

```
  "errors": [
    {
      "message": "org.neo4j.graphdb.NotFoundException: Relationship 1234 not found",
      "code": "Neo.ClientError.Statement.EntityNotFound"
    }
  ]
}
```

21.13. Node labels

Adding a label to a node

Figure 21.50. Final Graph



Example request

- POST http://localhost:7474/db/data/node/10129/labels
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

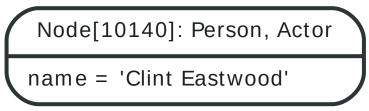
```
"Person"
```

Example response

- 204: No Content

Adding multiple labels to a node

Figure 21.51. Final Graph



Example request

- POST http://localhost:7474/db/data/node/10140/labels
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
[ "Person", "Actor" ]
```

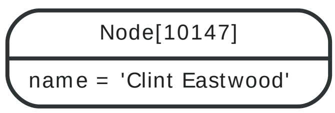
Example response

- 204: No Content

Adding a label with an invalid name

Labels with empty names are not allowed, however, all other valid strings are accepted as label names. Adding an invalid label to a node will lead to a HTTP 400 response.

Figure 21.52. Final Graph



Example request

- POST http://localhost:7474/db/data/node/10147/labels

- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
""
```

Example response

- 400: Bad Request
- Content-Type: application/json; charset=UTF-8

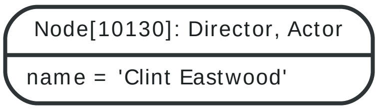

```
{
  "message": "Unable to add label, see nested exception.",
  "exception": "BadInputException",
  "fullname": "org.neo4j.server.rest.repr.BadInputException",
  "stackTrace": [
    "org.neo4j.server.rest.web.DatabaseActions.addLabelToNode(DatabaseActions.java:321)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.addNodeLabel(RestfulGraphDatabase.java:433)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "java.lang.Thread.run(Thread.java:745)"
  ],
  "cause": {
    "message": "Invalid label name \u0027\u0027.",
    "errors": [
      {
        "message": "Invalid label name \u0027\u0027.",
        "code": "Neo.ClientError.Schema.ConstraintViolation"
      }
    ],
    "cause": {
      "message": "\u0027\u0027 is not a valid token name. Only non-null, non-empty strings are allowed.",
      "errors": [
        {
          "message": "\u0027\u0027 is not a valid token name. Only non-null, non-empty strings are allowed.",
          "code": "Neo.ClientError.Schema.IllegalTokenName"
        }
      ],
      "exception": "IllegalTokenNameException",
      "fullname": "org.neo4j.kernel.api.exceptions.schema.IllegalTokenNameException",
      "stackTrace": [

 "org.neo4j.kernel.impl.api.DataIntegrityValidatingStatementOperations.checkValidTokenName(DataIntegrityValidatingStatementOperations.ja

 "org.neo4j.kernel.impl.api.DataIntegrityValidatingStatementOperations.labelGetOrCreateForName(DataIntegrityValidatingStatementOperation
        "org.neo4j.kernel.impl.api.OperationsFacade.labelGetOrCreateForName(OperationsFacade.java:544)",
        "org.neo4j.kernel.impl.core.NodeProxy.addLabel(NodeProxy.java:538)",
        "org.neo4j.server.rest.web.DatabaseActions.addLabelToNode(DatabaseActions.java:316)",
        "org.neo4j.server.rest.web.RestfulGraphDatabase.addNodeLabel(RestfulGraphDatabase.java:433)",
        "java.lang.reflect.Method.invoke(Method.java:606)",

 "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
        "java.lang.Thread.run(Thread.java:745)"
      ]
    },
    "exception": "ConstraintViolationException",
    "fullname": "org.neo4j.graphdb.ConstraintViolationException",
    "stackTrace": [
      "org.neo4j.kernel.impl.core.NodeProxy.addLabel(NodeProxy.java:549)",
      "org.neo4j.server.rest.web.DatabaseActions.addLabelToNode(DatabaseActions.java:316)",
      "org.neo4j.server.rest.web.RestfulGraphDatabase.addNodeLabel(RestfulGraphDatabase.java:433)",
      "java.lang.reflect.Method.invoke(Method.java:606)",
      "org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
      "java.lang.Thread.run(Thread.java:745)"
```

```
    ]
  },
  "errors": [
    {
      "message": "Unable to add label, see nested exception.",
      "code": "Neo.ClientError.Request.InvalidFormat"
    }
  ]
}
```


Replacing labels on a node

This removes any labels currently on a node, and replaces them with the labels passed in as the request body.

Figure 21.53. Final Graph



Example request

- PUT `http://localhost:7474/db/data/node/10130/labels`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

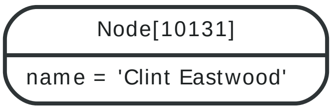
```
[ "Actor", "Director" ]
```

Example response

- `204: No Content`

Removing a label from a node

Figure 21.54. Final Graph



Example request

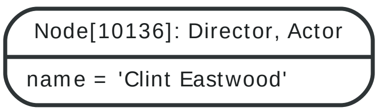
- DELETE `http://localhost:7474/db/data/node/10131/labels/Person`
- `Accept: application/json; charset=UTF-8`

Example response

- `204: No Content`

Listing labels for a node

Figure 21.55. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/10136/labels`
- `Accept: application/json; charset=UTF-8`

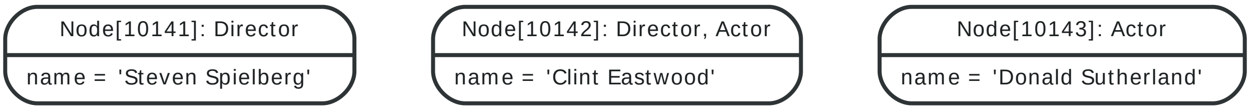
Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`


```
[ "Director", "Actor" ]
```


Get all nodes with a label

Figure 21.56. Final Graph



Example request

- GET `http://localhost:7474/db/data/label/Actor/nodes`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/10142/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/10142/labels",
  "data" : {
    "name" : "Clint Eastwood"
  },
  "traverse" : "http://localhost:7474/db/data/node/10142/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/10142/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/10142",
  "property" : "http://localhost:7474/db/data/node/10142/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/10142/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10142/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/10142/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/10142/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/10142/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/10142/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10142/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 10142,
    "labels" : [ "Director", "Actor" ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/10143/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/10143/labels",
  "data" : {
    "name" : "Donald Sutherland"
  },
  "traverse" : "http://localhost:7474/db/data/node/10143/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/10143/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/10143",
```

```
  "property" : "http://localhost:7474/db/data/node/10143/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/10143/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10143/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/10143/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/10143/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/10143/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/10143/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10143/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 10143,
    "labels" : [ "Actor" ]
  }
} ]
```

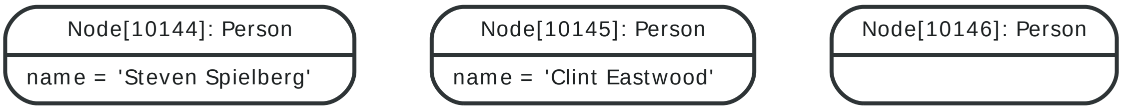

Get nodes by label and property

You can retrieve all nodes with a given label and property by passing one property as a query parameter. Notice that the property value is JSON-encoded and then URL-encoded.

If there is an index available on the label/property combination you send, that index will be used. If no index is available, all nodes with the given label will be filtered through to find matching nodes.

Currently, it is not possible to search using multiple properties.

Figure 21.57. Final Graph



Example request

- GET `http://localhost:7474/db/data/label/Person/nodes?name=%22Clint+Eastwood%22`
- Accept: `application/json; charset=UTF-8`

Example response

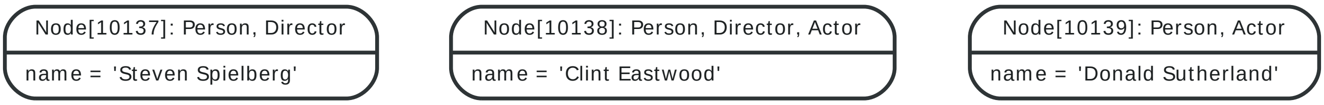
- 200: OK
- Content-Type: `application/json; charset=UTF-8`


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/10145/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/10145/labels",
  "data" : {
    "name" : "Clint Eastwood"
  },
  "traverse" : "http://localhost:7474/db/data/node/10145/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/10145/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/10145",
  "property" : "http://localhost:7474/db/data/node/10145/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/10145/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10145/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/10145/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/10145/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/10145/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/10145/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10145/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 10145,
    "labels" : [ "Person" ]
  }
```

```
} ]
```

List all labels

Figure 21.58. Final Graph



Example request

- GET `http://localhost:7474/db/data/labels`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

```
[ "AlbumRelease", "DecibelEntity", "AlbumGroup", "Person", "First", "Foo", "Director", "Bar", "Second", "Actor" ]
```

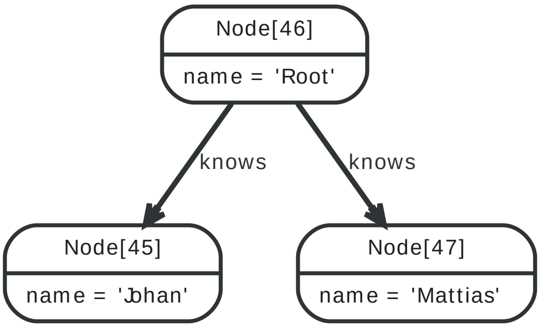
21.14. Node degree

The node degree is the number of relationships associated with a node. Neo4j stores the degree for each node, making this a useful mechanism to quickly get the number of relationships a node has. You can also optionally filter degree by direction and/or relationship type.

Get the degree of a node

Return the total number of relationships associated with a node.

Figure 21.59. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/46/degree/all`
- `Accept: application/json; charset=UTF-8`

Example response

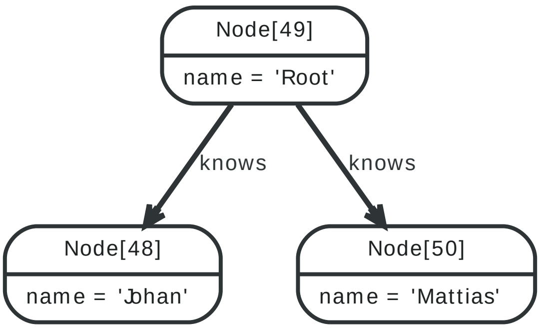
- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

```
2
```

Get the degree of a node by direction

Return the number of relationships of a particular direction for a node. Specify `all`, `in` or `out`.

Figure 21.60. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/49/degree/out`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`

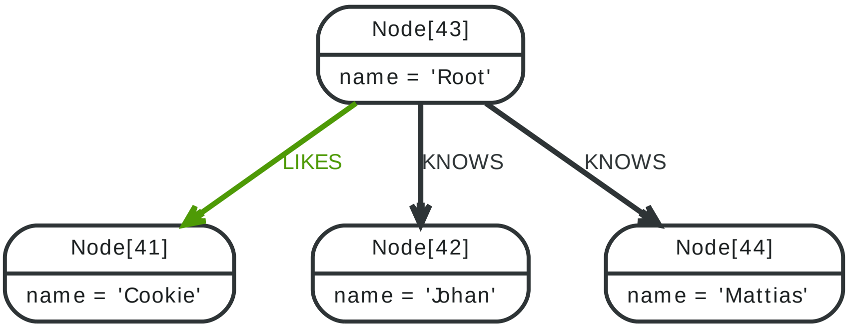
- `Content-Type: application/json; charset=UTF-8`

```
2
```

Get the degree of a node by direction and types

If you are only interested in the degree of a particular relationship type, or a set of relationship types, you specify relationship types after the direction. You can combine multiple relationship types by using the `&` character.

Figure 21.61. Final Graph



Example request

- `GET http://localhost:7474/db/data/node/43/degree/out/KNOWS&LIKES`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

```
3
```

21.15. Indexing

Note
This documents schema based indexes, a feature that was introduced in Neo4j 2.0, see Section 21.20, "Legacy indexing" [398] for legacy indexing.

For more details about indexes and the optional schema in Neo4j, see the section called "Schema" [9].

Create index

This will start a background job in the database that will create and populate the index. You can check the status of your index by listing all the indexes for the relevant label. The created index will show up, but have a state of `POPULATING` until the index is ready, where it is marked as `ONLINE`.

Example request

- `POST http://localhost:7474/db/data/schema/index/person`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
{
  "property_keys" : [ "name" ]
}
```

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

```
{
  "label" : "person",
  "property_keys" : [ "name" ]
}
```

List indexes for a label

Example request

- `GET http://localhost:7474/db/data/schema/index/user`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`

```
[ {
  "property_keys" : [ "name" ],
  "label" : "user"
} ]
```

Drop index

Drop index

Example request

- `DELETE http://localhost:7474/db/data/schema/index/SomeLabel/name`
- `Accept: application/json; charset=UTF-8`

Example response

- `204`: No Content

21.16. Constraints

Create uniqueness constraint

Create a uniqueness constraint on a property.

Example request

- POST `http://localhost:7474/db/data/schema/constraint/Person/uniqueness/`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "property_keys" : [ "name" ]
}
```

Example response

- `200: OK`
- Content-Type: `application/json; charset=UTF-8`

```
{
  "label" : "Person",
  "type" : "UNIQUENESS",
  "property_keys" : [ "name" ]
}
```

Get a specific uniqueness constraint

Get a specific uniqueness constraint for a label and a property.

Example request

- GET `http://localhost:7474/db/data/schema/constraint/User/uniqueness/name`
- Accept: `application/json; charset=UTF-8`

Example response

- `200: OK`
- Content-Type: `application/json; charset=UTF-8`

```
[ {
  "property_keys" : [ "name" ],
  "label" : "User",
  "type" : "UNIQUENESS"
} ]
```

Get all uniqueness constraints for a label

Example request

- GET `http://localhost:7474/db/data/schema/constraint/User/uniqueness/`
- Accept: `application/json; charset=UTF-8`

Example response

- `200: OK`
- Content-Type: `application/json; charset=UTF-8`

```
[ {
  "property_keys" : [ "name2" ],
  "label" : "User",
  "type" : "UNIQUENESS"
}, {
  "property_keys" : [ "name1" ],
  "label" : "User",
  "type" : "UNIQUENESS"
} ]
```


Get all constraints for a label

Example request

- GET `http://localhost:7474/db/data/schema/constraint/User`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`


```
[ {
  "property_keys" : [ "name2" ],
  "label" : "User",
  "type" : "UNIQUENESS"
}, {
  "property_keys" : [ "name1" ],
  "label" : "User",
  "type" : "UNIQUENESS"
} ]
```


Get all constraints

Example request

- GET `http://localhost:7474/db/data/schema/constraint`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`


```
[ {
  "property_keys" : [ "name2" ],
  "label" : "Prog",
  "type" : "UNIQUENESS"
}, {
  "property_keys" : [ "name1" ],
  "label" : "User",
  "type" : "UNIQUENESS"
} ]
```


Drop constraint

Drop uniqueness constraint for a label and a property.

Example request

- DELETE `http://localhost:7474/db/data/schema/constraint/SomeLabel/uniqueness/name`

- `Accept`: `application/json; charset=UTF-8`

Example response

- `204`: `No Content`

21.17. Traversals

Warning
The Traversal REST Endpoint executes arbitrary Javascript code under the hood as part of the evaluators definitions. In hosted and open environments, this can constitute a security risk. In these case, consider using declarative approaches like Part III, "Cypher Query Language" [101] or write your own server side plugin executing the interesting traversals with the Java API (see Section 34.1, "Server Plugins" [577]) or secure your server, see Chapter 27, *Security* [496].

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[3].
  - `depth_first`: see Depth-first search[4]

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[5]. 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 the section called "Traversal using a return filter" [357] 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 the section called "Traversal using a return filter" [357]), or use one of the built-in filters:
  - `all`
  - `all_but_start_node`

[3] http://en.wikipedia.org/wiki/Breadth-first_search
[4] http://en.wikipedia.org/wiki/Depth-first_search
[5] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/Uniqueness.html

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.

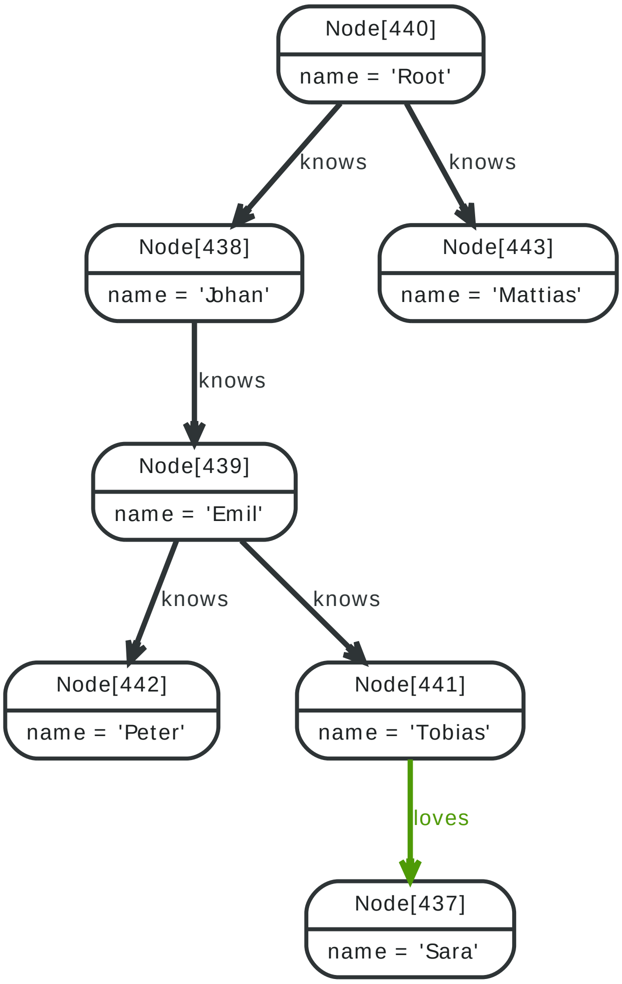
The `position` object in the body of the `return_filter` and `prune_evaluator` is a `Path`[6] object representing the path from the start node to the current traversal position.

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[7]. See the examples for the exact syntax of the request.

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 21.62. Final Graph



Example request

- `POST http://localhost:7474/db/data/node/440/traverse/node`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
{
  "order" : "breadth_first",
```

6 http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/Path.html
7 http://neo4j.com/docs/2.2.1/javadocs/

```
  "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

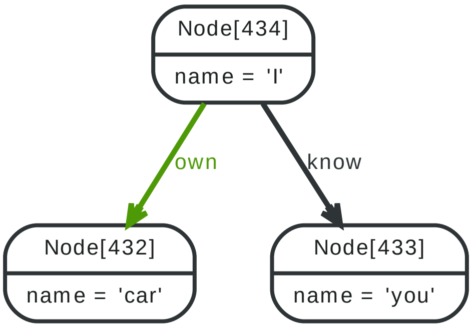
- 200: OK
- Content-Type: application/json; charset=UTF-8


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/440/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/440/labels",
  "data" : {
    "name" : "Root"
  },
  "traverse" : "http://localhost:7474/db/data/node/440/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/440/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/440",
  "property" : "http://localhost:7474/db/data/node/440/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/440/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/440/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/440/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/440/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/440/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/440/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/440/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 440,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/443/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/443/labels",
  "data" : {
    "name" : "Mattias"
  },
  "traverse" : "http://localhost:7474/db/data/node/443/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/443/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/443",
  "property" : "http://localhost:7474/db/data/node/443/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/443/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/443/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/443/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/443/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/443/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/443/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/443/relationships/in/{-list|&|types}",
```

```
    "metadata" : {
      "id" : 443,
      "labels" : [ ]
    }
  }, {
    "outgoing_relationships" : "http://localhost:7474/db/data/node/442/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/442/labels",
    "data" : {
      "name" : "Peter"
    },
    "traverse" : "http://localhost:7474/db/data/node/442/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/442/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/442",
    "property" : "http://localhost:7474/db/data/node/442/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/442/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/442/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/442/relationships/in",
    "extensions" : { },
    "create_relationship" : "http://localhost:7474/db/data/node/442/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/442/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/442/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/442/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 442,
      "labels" : [ ]
    }
  }, {
    "outgoing_relationships" : "http://localhost:7474/db/data/node/441/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/441/labels",
    "data" : {
      "name" : "Tobias"
    },
    "traverse" : "http://localhost:7474/db/data/node/441/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/441/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/441",
    "property" : "http://localhost:7474/db/data/node/441/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/441/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/441/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/441/relationships/in",
    "extensions" : { },
    "create_relationship" : "http://localhost:7474/db/data/node/441/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/441/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/441/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/441/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 441,
      "labels" : [ ]
    }
  } ]
```

Return relationships from a traversal

Figure 21.63. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/434/traverse/relationship`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "order" : "breadth_first",
  "uniqueness" : "none",
  "return_filter" : {
    "language" : "builtin",
    "name" : "all"
  }
}
```

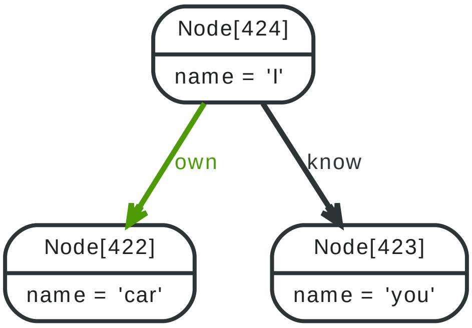
Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
[ {
  "start" : "http://localhost:7474/db/data/node/434",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/259",
  "property" : "http://localhost:7474/db/data/relationship/259/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/259/properties",
  "type" : "know",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/433",
  "metadata" : {
    "id" : 259,
    "type" : "know"
  }
}, {
  "start" : "http://localhost:7474/db/data/node/434",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/260",
  "property" : "http://localhost:7474/db/data/relationship/260/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/260/properties",
  "type" : "own",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/432",
  "metadata" : {
    "id" : 260,
    "type" : "own"
  }
} ]
```

Return paths from a traversal

Figure 21.64. Final Graph



Example request

- POST http://localhost:7474/db/data/node/424/traverse/path
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "order" : "breadth_first",
  "uniqueness" : "none",
  "return_filter" : {
    "language" : "builtin",
    "name" : "all"
  }
}
```

Example response

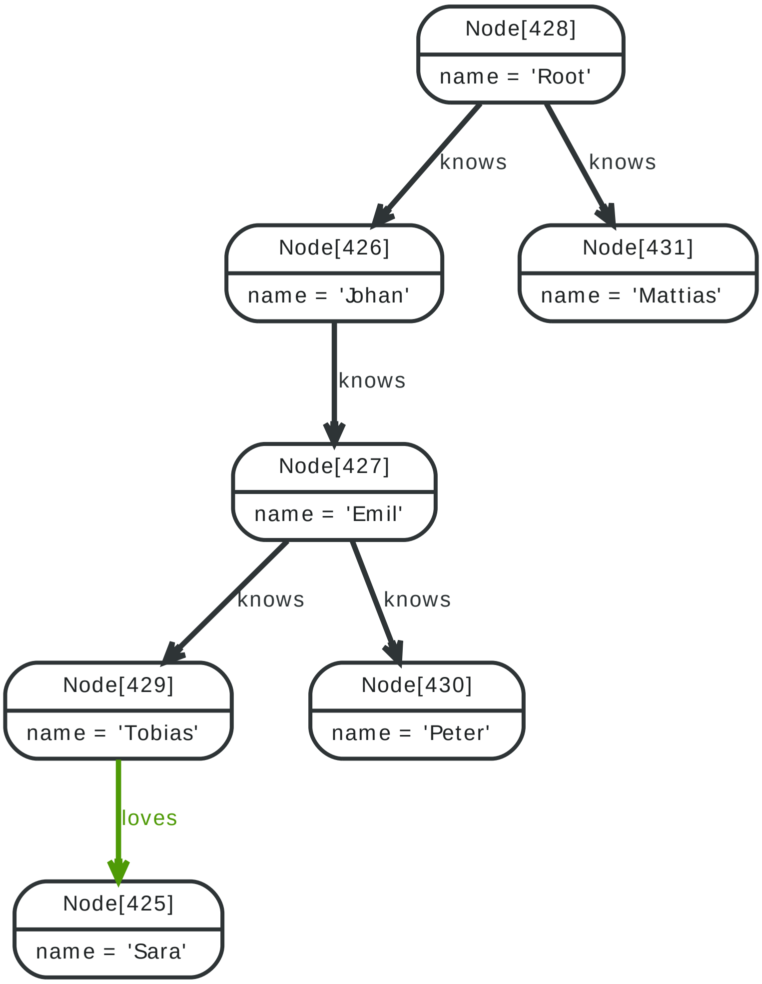
- 200: OK
- Content-Type: application/json; charset=UTF-8

```
[ {
  "directions" : [ ],
  "start" : "http://localhost:7474/db/data/node/424",
  "nodes" : [ "http://localhost:7474/db/data/node/424" ],
  "length" : 0,
  "relationships" : [ ],
  "end" : "http://localhost:7474/db/data/node/424"
}, {
  "directions" : [ "->" ],
  "start" : "http://localhost:7474/db/data/node/424",
  "nodes" : [ "http://localhost:7474/db/data/node/424", "http://localhost:7474/db/data/node/423" ],
  "length" : 1,
  "relationships" : [ "http://localhost:7474/db/data/relationship/251" ],
  "end" : "http://localhost:7474/db/data/node/423"
}, {
  "directions" : [ "->" ],
  "start" : "http://localhost:7474/db/data/node/424",
  "nodes" : [ "http://localhost:7474/db/data/node/424", "http://localhost:7474/db/data/node/422" ],
  "length" : 1,
  "relationships" : [ "http://localhost:7474/db/data/relationship/252" ],
  "end" : "http://localhost:7474/db/data/node/422"
} ]
```

Traversal returning nodes below a certain depth

Here, all nodes at a traversal depth below 3 are returned.

Figure 21.65. Final Graph



Example request

- POST http://localhost:7474/db/data/node/428/traverse/node
- Accept: application/json; charset=UTF-8
- 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; charset=UTF-8

```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/428/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/428/labels",
  "data" : {
    "name" : "Root"
```

```
  },
  "traverse" : "http://localhost:7474/db/data/node/428/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/428/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/428",
  "property" : "http://localhost:7474/db/data/node/428/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/428/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/428/relationships/out/{-list|&|types}",
  "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}",
  "metadata" : {
    "id" : 428,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/431/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/431/labels",
  "data" : {
    "name" : "Mattias"
  },
  "traverse" : "http://localhost:7474/db/data/node/431/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/431/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/431",
  "property" : "http://localhost:7474/db/data/node/431/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/431/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/431/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/431/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/431/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/431/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/431/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/431/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 431,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/426/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/426/labels",
  "data" : {
    "name" : "Johan"
  },
  "traverse" : "http://localhost:7474/db/data/node/426/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/426/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/426",
  "property" : "http://localhost:7474/db/data/node/426/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/426/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/426/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/426/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/426/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/426/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/426/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/426/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 426,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/427/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/427/labels",
  "data" : {
```

```
    "name" : "Emil"
  },
  "traverse" : "http://localhost:7474/db/data/node/427/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/427/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/427",
  "property" : "http://localhost:7474/db/data/node/427/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/427/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/427/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/427/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/427/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/427/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/427/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/427/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 427,
    "labels" : [ ]
  }
} ]
```


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/297/paged/traverse/node`
- `Accept: application/json; charset=UTF-8`
- `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; charset=UTF-8`
- `Location: http://localhost:7474/db/data/node/297/paged/traverse/node/0d668a91b9ee40ff94494ed604b8dd3b`


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/298/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/298/labels",
  "data" : {
    "name" : "1"
  },
  "traverse" : "http://localhost:7474/db/data/node/298/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/298/relationships/all/{-list|&|types}",
```

```
  "self" : "http://localhost:7474/db/data/node/298",
  "property" : "http://localhost:7474/db/data/node/298/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/298/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/298/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/298/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/298/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/298/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/298/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/298/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 298,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/307/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/307/labels",
  "data" : {
    "name" : "10"
  },
  "traverse" : "http://localhost:7474/db/data/node/307/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/307/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/307",
  "property" : "http://localhost:7474/db/data/node/307/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/307/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/307/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/307/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/307/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/307/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/307/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/307/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 307,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/308/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/308/labels",
  "data" : {
    "name" : "11"
  },
  "traverse" : "http://localhost:7474/db/data/node/308/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/308/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/308",
  "property" : "http://localhost:7474/db/data/node/308/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/308/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/308/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/308/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/308/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/308/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/308/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/308/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 308,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/309/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/309/labels",
  "data" : {
    "name" : "12"
  },
  "traverse" : "http://localhost:7474/db/data/node/309/traverse/{returnType}",
```

```
  "all_typed_relationships" : "http://localhost:7474/db/data/node/309/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/309",
  "property" : "http://localhost:7474/db/data/node/309/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/309/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/309/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/309/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/309/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/309/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/309/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/309/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 309,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/310/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/310/labels",
  "data" : {
    "name" : "13"
  },
  "traverse" : "http://localhost:7474/db/data/node/310/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/310/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/310",
  "property" : "http://localhost:7474/db/data/node/310/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/310/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/310/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/310/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/310/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/310/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/310/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/310/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 310,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/311/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/311/labels",
  "data" : {
    "name" : "14"
  },
  "traverse" : "http://localhost:7474/db/data/node/311/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/311/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/311",
  "property" : "http://localhost:7474/db/data/node/311/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/311/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/311/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/311/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/311/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/311/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/311/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/311/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 311,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/312/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/312/labels",
  "data" : {
    "name" : "15"
  },
```

```
  "traverse" : "http://localhost:7474/db/data/node/312/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/312/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/312",
  "property" : "http://localhost:7474/db/data/node/312/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/312/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/312/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/312/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/312/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/312/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/312/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/312/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 312,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/313/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/313/labels",
  "data" : {
    "name" : "16"
  },
  "traverse" : "http://localhost:7474/db/data/node/313/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/313/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/313",
  "property" : "http://localhost:7474/db/data/node/313/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/313/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/313/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/313/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/313/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/313/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/313/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/313/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 313,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/314/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/314/labels",
  "data" : {
    "name" : "17"
  },
  "traverse" : "http://localhost:7474/db/data/node/314/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/314/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/314",
  "property" : "http://localhost:7474/db/data/node/314/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/314/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/314/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/314/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/314/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/314/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/314/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/314/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 314,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/315/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/315/labels",
  "data" : {
    "name" : "18"
```

```
  },
  "traverse" : "http://localhost:7474/db/data/node/315/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/315/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/315",
  "property" : "http://localhost:7474/db/data/node/315/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/315/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/315/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/315/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/315/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/315/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/315/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/315/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 315,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/316/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/316/labels",
  "data" : {
    "name" : "19"
  },
  "traverse" : "http://localhost:7474/db/data/node/316/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/316/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/316",
  "property" : "http://localhost:7474/db/data/node/316/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/316/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/316/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/316/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/316/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/316/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/316/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/316/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 316,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/318/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/318/labels",
  "data" : {
    "name" : "21"
  },
  "traverse" : "http://localhost:7474/db/data/node/318/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/318/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/318",
  "property" : "http://localhost:7474/db/data/node/318/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/318/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/318/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/318/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/318/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/318/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/318/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/318/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 318,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/328/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/328/labels",
  "data" : {
```

```
    "name" : "31"
  },
  "traverse" : "http://localhost:7474/db/data/node/328/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/328/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/328",
  "property" : "http://localhost:7474/db/data/node/328/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/328/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/328/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/328/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/328/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/328/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/328/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/328/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 328,
    "labels" : [ ]
  }
} ]
```


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/330/paged/traverse/node/2a04aa756ab64814acf41826ae8198a4`
- `Accept: application/json`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/661/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/661/labels",
  "data" : {
    "name" : "331"
  },
  "traverse" : "http://localhost:7474/db/data/node/661/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/661/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/661",
  "property" : "http://localhost:7474/db/data/node/661/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/661/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/661/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/661/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/661/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/661/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/661/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/661/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 661,
```

```
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/671/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/671/labels",
  "data" : {
    "name" : "341"
  },
  "traverse" : "http://localhost:7474/db/data/node/671/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/671/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/671",
  "property" : "http://localhost:7474/db/data/node/671/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/671/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/671/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/671/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/671/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/671/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/671/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/671/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 671,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/681/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/681/labels",
  "data" : {
    "name" : "351"
  },
  "traverse" : "http://localhost:7474/db/data/node/681/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/681/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/681",
  "property" : "http://localhost:7474/db/data/node/681/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/681/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/681/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/681/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/681/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/681/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/681/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/681/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 681,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/691/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/691/labels",
  "data" : {
    "name" : "361"
  },
  "traverse" : "http://localhost:7474/db/data/node/691/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/691/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/691",
  "property" : "http://localhost:7474/db/data/node/691/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/691/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/691/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/691/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/691/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/691/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/691/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/691/relationships/in/{-list|&|types}",
  "metadata" : {
```

```
    "id" : 691,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/701/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/701/labels",
  "data" : {
    "name" : "371"
  },
  "traverse" : "http://localhost:7474/db/data/node/701/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/701/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/701",
  "property" : "http://localhost:7474/db/data/node/701/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/701/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/701/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/701/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/701/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/701/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/701/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/701/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 701,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/711/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/711/labels",
  "data" : {
    "name" : "381"
  },
  "traverse" : "http://localhost:7474/db/data/node/711/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/711/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/711",
  "property" : "http://localhost:7474/db/data/node/711/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/711/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/711/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/711/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/711/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/711/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/711/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/711/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 711,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/721/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/721/labels",
  "data" : {
    "name" : "391"
  },
  "traverse" : "http://localhost:7474/db/data/node/721/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/721/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/721",
  "property" : "http://localhost:7474/db/data/node/721/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/721/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/721/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/721/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/721/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/721/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/721/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/721/relationships/in/{-list|&|types}",
```

```
  "metadata" : {
    "id" : 721,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/731/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/731/labels",
  "data" : {
    "name" : "401"
  },
  "traverse" : "http://localhost:7474/db/data/node/731/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/731/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/731",
  "property" : "http://localhost:7474/db/data/node/731/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/731/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/731/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/731/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/731/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/731/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/731/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/731/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 731,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/740/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/740/labels",
  "data" : {
    "name" : "410"
  },
  "traverse" : "http://localhost:7474/db/data/node/740/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/740/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/740",
  "property" : "http://localhost:7474/db/data/node/740/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/740/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/740/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/740/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/740/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/740/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/740/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/740/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 740,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/741/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/741/labels",
  "data" : {
    "name" : "411"
  },
  "traverse" : "http://localhost:7474/db/data/node/741/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/741/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/741",
  "property" : "http://localhost:7474/db/data/node/741/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/741/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/741/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/741/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/741/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/741/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/741/relationships/all",
```

```
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/741/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 741,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/742/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/742/labels",
  "data" : {
    "name" : "412"
  },
  "traverse" : "http://localhost:7474/db/data/node/742/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/742/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/742",
  "property" : "http://localhost:7474/db/data/node/742/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/742/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/742/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/742/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/742/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/742/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/742/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/742/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 742,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/743/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/743/labels",
  "data" : {
    "name" : "413"
  },
  "traverse" : "http://localhost:7474/db/data/node/743/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/743/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/743",
  "property" : "http://localhost:7474/db/data/node/743/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/743/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/743/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/743/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/743/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/743/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/743/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/743/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 743,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/744/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/744/labels",
  "data" : {
    "name" : "414"
  },
  "traverse" : "http://localhost:7474/db/data/node/744/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/744/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/744",
  "property" : "http://localhost:7474/db/data/node/744/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/744/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/744/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/744/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/744/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/744/paged/traverse/{returnType}{?pageSize,leaseTime}",
```

```
  "all_relationships" : "http://localhost:7474/db/data/node/744/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/744/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 744,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/745/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/745/labels",
  "data" : {
    "name" : "415"
  },
  "traverse" : "http://localhost:7474/db/data/node/745/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/745/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/745",
  "property" : "http://localhost:7474/db/data/node/745/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/745/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/745/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/745/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/745/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/745/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/745/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/745/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 745,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/746/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/746/labels",
  "data" : {
    "name" : "416"
  },
  "traverse" : "http://localhost:7474/db/data/node/746/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/746/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/746",
  "property" : "http://localhost:7474/db/data/node/746/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/746/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/746/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/746/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/746/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/746/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/746/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/746/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 746,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/747/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/747/labels",
  "data" : {
    "name" : "417"
  },
  "traverse" : "http://localhost:7474/db/data/node/747/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/747/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/747",
  "property" : "http://localhost:7474/db/data/node/747/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/747/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/747/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/747/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/747/relationships",
```

```
  "paged_traverse" : "http://localhost:7474/db/data/node/747/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/747/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/747/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 747,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/748/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/748/labels",
  "data" : {
    "name" : "418"
  },
  "traverse" : "http://localhost:7474/db/data/node/748/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/748/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/748",
  "property" : "http://localhost:7474/db/data/node/748/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/748/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/748/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/748/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/748/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/748/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/748/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/748/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 748,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/749/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/749/labels",
  "data" : {
    "name" : "419"
  },
  "traverse" : "http://localhost:7474/db/data/node/749/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/749/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/749",
  "property" : "http://localhost:7474/db/data/node/749/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/749/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/749/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/749/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/749/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/749/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/749/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/749/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 749,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/751/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/751/labels",
  "data" : {
    "name" : "421"
  },
  "traverse" : "http://localhost:7474/db/data/node/751/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/751/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/751",
  "property" : "http://localhost:7474/db/data/node/751/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/751/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/751/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/751/relationships/in",
  "extensions" : { },
```

```
  "create_relationship" : "http://localhost:7474/db/data/node/751/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/751/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/751/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/751/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 751,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/761/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/761/labels",
  "data" : {
    "name" : "431"
  },
  "traverse" : "http://localhost:7474/db/data/node/761/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/761/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/761",
  "property" : "http://localhost:7474/db/data/node/761/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/761/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/761/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/761/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/761/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/761/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/761/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/761/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 761,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/771/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/771/labels",
  "data" : {
    "name" : "441"
  },
  "traverse" : "http://localhost:7474/db/data/node/771/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/771/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/771",
  "property" : "http://localhost:7474/db/data/node/771/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/771/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/771/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/771/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/771/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/771/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/771/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/771/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 771,
    "labels" : [ ]
  }
} ]
```


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/33/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; charset=UTF-8`
- `Location: http://localhost:7474/db/data/node/33/paged/traverse/node/c8d1e309d83d4e95a4083209c0ab92aa`

```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/34/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/34/labels",
  "data" : {
    "name" : "1"
  },
  "traverse" : "http://localhost:7474/db/data/node/34/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/34/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/34",
  "property" : "http://localhost:7474/db/data/node/34/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/34/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/34/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/34/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/34/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/34/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/34/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/34/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 34,
    "labels" : [ ]
  }
} ]
```

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/807/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; charset=UTF-8
- Location: http://localhost:7474/db/data/node/807/paged/traverse/node/78365a9e4f9e4d4fa8719477e9a8f974


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/808/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/808/labels",
  "data" : {
    "name" : "1"
  },
  "traverse" : "http://localhost:7474/db/data/node/808/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/808/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/808",
  "property" : "http://localhost:7474/db/data/node/808/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/808/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/808/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/808/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/808/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/808/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/808/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/808/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 808,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/817/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/817/labels",
  "data" : {
    "name" : "10"
  },
  "traverse" : "http://localhost:7474/db/data/node/817/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/817/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/817",
  "property" : "http://localhost:7474/db/data/node/817/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/817/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/817/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/817/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/817/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/817/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/817/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/817/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 817,
    "labels" : [ ]
  }
}, {
```

```
  "outgoing_relationships" : "http://localhost:7474/db/data/node/818/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/818/labels",
  "data" : {
    "name" : "11"
  },
  "traverse" : "http://localhost:7474/db/data/node/818/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/818/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/818",
  "property" : "http://localhost:7474/db/data/node/818/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/818/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/818/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/818/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/818/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/818/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/818/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/818/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 818,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/819/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/819/labels",
  "data" : {
    "name" : "12"
  },
  "traverse" : "http://localhost:7474/db/data/node/819/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/819/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/819",
  "property" : "http://localhost:7474/db/data/node/819/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/819/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/819/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/819/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/819/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/819/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/819/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/819/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 819,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/820/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/820/labels",
  "data" : {
    "name" : "13"
  },
  "traverse" : "http://localhost:7474/db/data/node/820/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/820/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/820",
  "property" : "http://localhost:7474/db/data/node/820/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/820/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/820/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/820/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/820/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/820/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/820/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/820/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 820,
    "labels" : [ ]
  }
```

```
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/821/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/821/labels",
  "data" : {
    "name" : "14"
  },
  "traverse" : "http://localhost:7474/db/data/node/821/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/821/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/821",
  "property" : "http://localhost:7474/db/data/node/821/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/821/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/821/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/821/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/821/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/821/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/821/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/821/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 821,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/822/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/822/labels",
  "data" : {
    "name" : "15"
  },
  "traverse" : "http://localhost:7474/db/data/node/822/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/822/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/822",
  "property" : "http://localhost:7474/db/data/node/822/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/822/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/822/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/822/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/822/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/822/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/822/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/822/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 822,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/823/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/823/labels",
  "data" : {
    "name" : "16"
  },
  "traverse" : "http://localhost:7474/db/data/node/823/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/823/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/823",
  "property" : "http://localhost:7474/db/data/node/823/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/823/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/823/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/823/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/823/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/823/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/823/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/823/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 823,
    "labels" : [ ]
```

```
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/824/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/824/labels",
  "data" : {
    "name" : "17"
  },
  "traverse" : "http://localhost:7474/db/data/node/824/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/824/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/824",
  "property" : "http://localhost:7474/db/data/node/824/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/824/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/824/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/824/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/824/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/824/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/824/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/824/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 824,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/825/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/825/labels",
  "data" : {
    "name" : "18"
  },
  "traverse" : "http://localhost:7474/db/data/node/825/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/825/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/825",
  "property" : "http://localhost:7474/db/data/node/825/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/825/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/825/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/825/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/825/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/825/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/825/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/825/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 825,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/826/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/826/labels",
  "data" : {
    "name" : "19"
  },
  "traverse" : "http://localhost:7474/db/data/node/826/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/826/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/826",
  "property" : "http://localhost:7474/db/data/node/826/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/826/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/826/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/826/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/826/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/826/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/826/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/826/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 826,
```

```
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/828/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/828/labels",
  "data" : {
    "name" : "21"
  },
  "traverse" : "http://localhost:7474/db/data/node/828/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/828/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/828",
  "property" : "http://localhost:7474/db/data/node/828/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/828/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/828/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/828/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/828/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/828/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/828/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/828/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 828,
    "labels" : [ ]
  }
}, {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/838/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/838/labels",
  "data" : {
    "name" : "31"
  },
  "traverse" : "http://localhost:7474/db/data/node/838/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/838/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/838",
  "property" : "http://localhost:7474/db/data/node/838/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/838/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/838/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/838/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/838/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/838/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/838/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/838/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 838,
    "labels" : [ ]
  }
} ]
```

21.18. 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. These are the parameters that can be used:

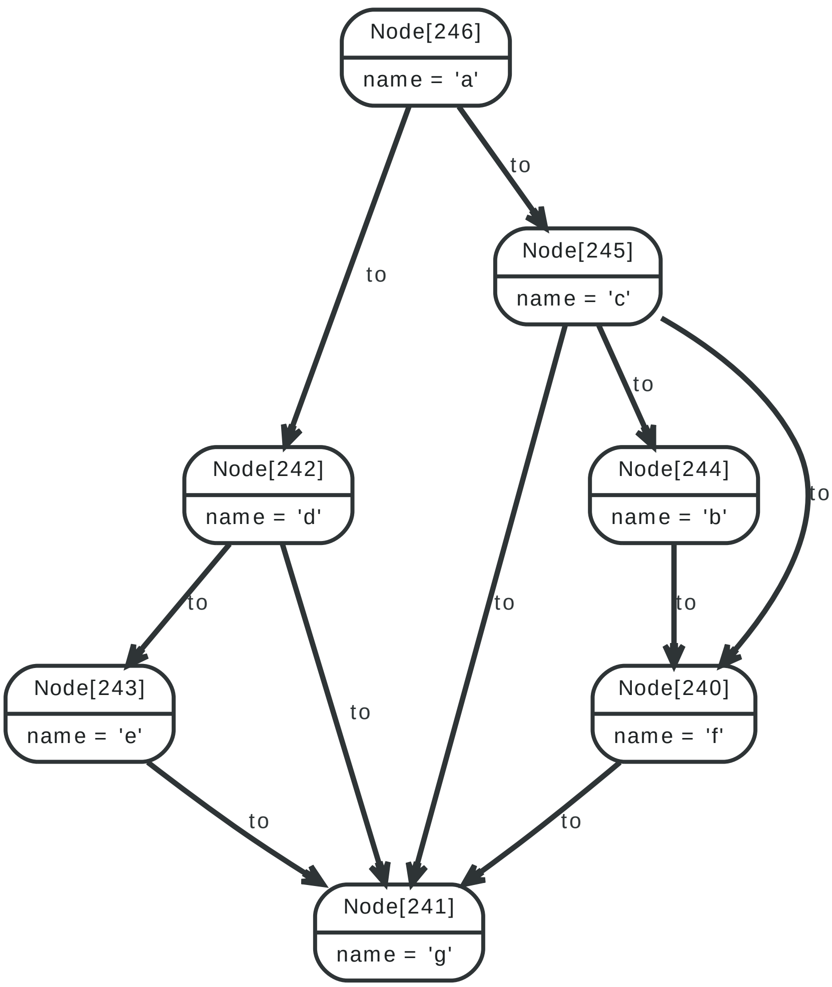
algorithm
: The algorithm to choose. If not set, default is `shortestPath`. `algorithm` can have one of these values:
  - `shortestPath`
  - `allSimplePaths`
  - `allPaths`
  - `dijkstra` (optionally with `cost_property` and `default_cost` parameters)

max_depth
: The maximum depth as an integer for the algorithms like `shortestPath`, where applicable. Default is `1`.

Find all shortest paths

The `shortestPath` algorithm can find multiple paths between the same nodes, like in this example.

Figure 21.66. Final Graph



Example request

- `POST http://localhost:7474/db/data/node/246/paths`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
{
  "to" : "http://localhost:7474/db/data/node/241",
  "max_depth" : 3,
  "relationships" : {
    "type" : "to",
    "direction" : "out"
  },
  "algorithm" : "shortestPath"
}
```

Example response

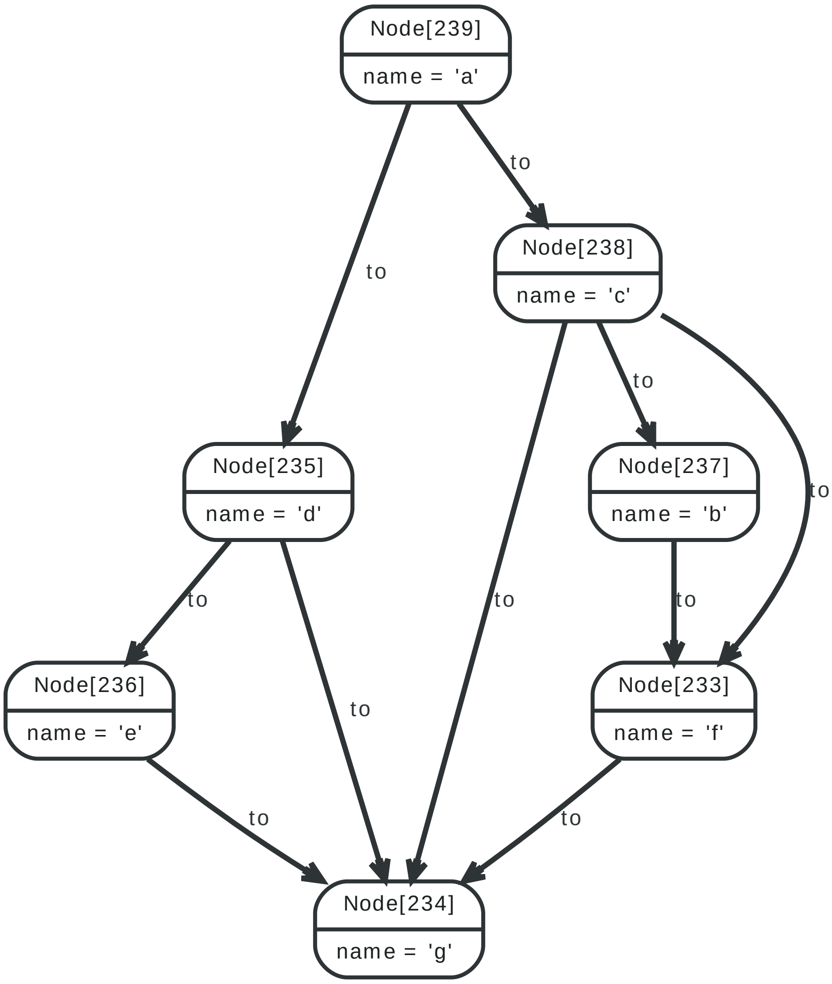
- `200:` OK
- `Content-Type:` application/json; charset=UTF-8

```
[ {
  "directions" : [ "->", "->" ],
  "start" : "http://localhost:7474/db/data/node/246",
  "nodes" : [ "http://localhost:7474/db/data/node/246", "http://localhost:7474/db/data/node/245", "http://localhost:7474/db/
data/node/241" ],
  "length" : 2,
  "relationships" : [ "http://localhost:7474/db/data/relationship/152", "http://localhost:7474/db/data/relationship/161" ],
  "end" : "http://localhost:7474/db/data/node/241"
}, {
  "directions" : [ "->", "->" ],
  "start" : "http://localhost:7474/db/data/node/246",
  "nodes" : [ "http://localhost:7474/db/data/node/246", "http://localhost:7474/db/data/node/242", "http://localhost:7474/db/
data/node/241" ],
  "length" : 2,
  "relationships" : [ "http://localhost:7474/db/data/relationship/153", "http://localhost:7474/db/data/relationship/159" ],
  "end" : "http://localhost:7474/db/data/node/241"
} ]
```

Find one of the shortest paths

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 a maximum of 3 linked nodes.

Figure 21.67. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/239/path`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "to" : "http://localhost:7474/db/data/node/234",
  "max_depth" : 3,
  "relationships" : {
    "type" : "to",
    "direction" : "out"
  },
  "algorithm" : "shortestPath"
}
```

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

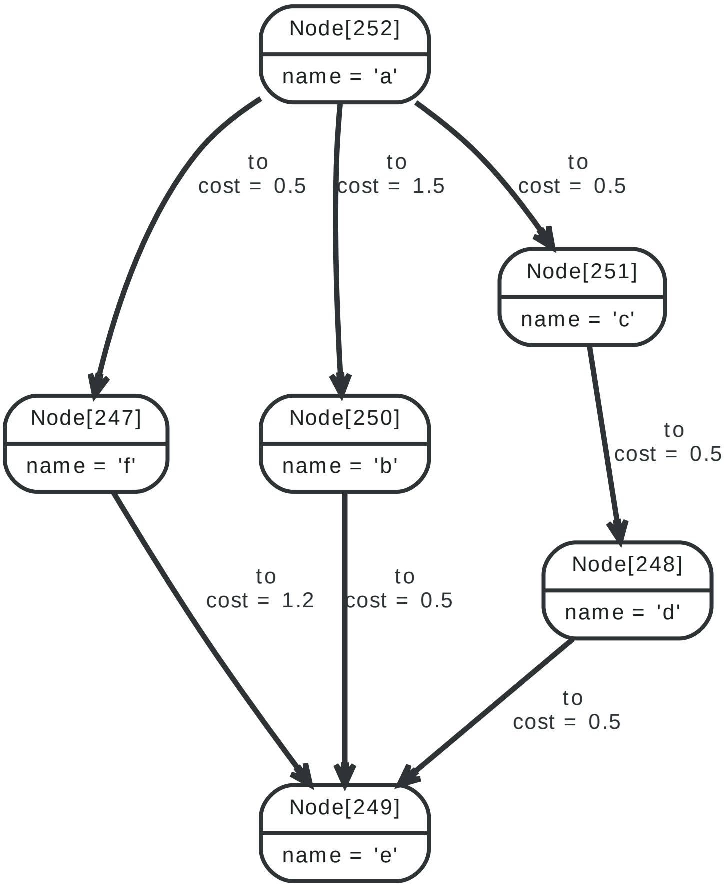
```
{
  "directions" : [ "->", "->" ],
  "start" : "http://localhost:7474/db/data/node/239",
  "nodes" : [ "http://localhost:7474/db/data/node/239", "http://localhost:7474/db/data/node/238", "http://localhost:7474/db/
data/node/234" ],
  "length" : 2,
```

```
  "relationships" : [ "http://localhost:7474/db/data/relationship/142", "http://localhost:7474/db/data/relationship/151" ],
  "end" : "http://localhost:7474/db/data/node/234"
}
```

Execute a Dijkstra algorithm and get a single path

This example is running a Dijkstra algorithm over a graph with different cost properties on different relationships. Note that the request URI ends with `/path` which means a single path is what we want here.

Figure 21.68. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/252/path`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "to" : "http://localhost:7474/db/data/node/249",
  "cost_property" : "cost",
  "relationships" : {
    "type" : "to",
    "direction" : "out"
  },
  "algorithm" : "dijkstra"
}
```

Example response

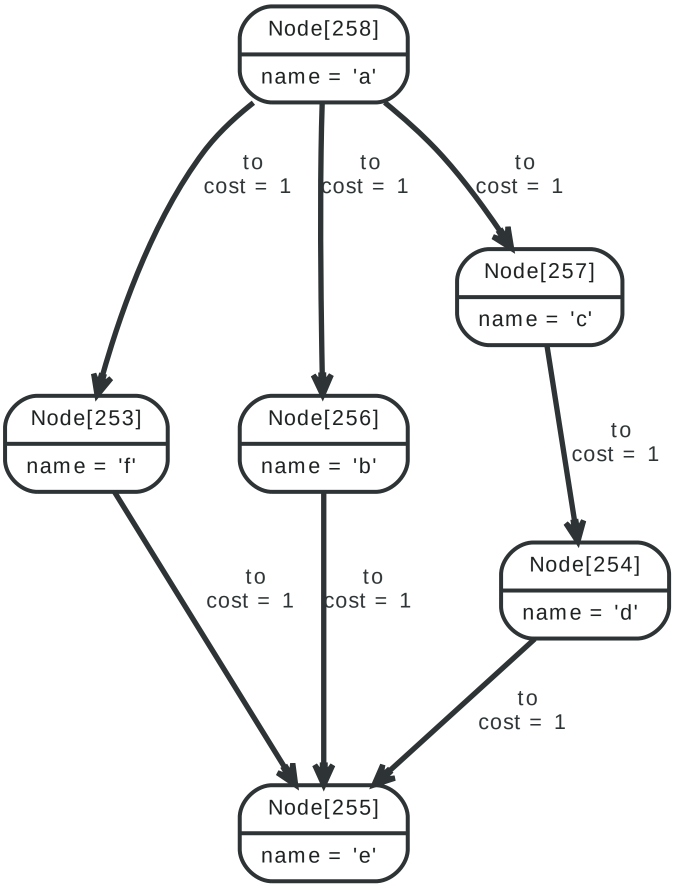
- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
{
  "directions" : [ "->", "->", "->" ],
  "weight" : 1.5,
  "start" : "http://localhost:7474/db/data/node/252",
  "nodes" : [ "http://localhost:7474/db/data/node/252", "http://localhost:7474/db/data/node/251", "http://localhost:7474/db/
data/node/248", "http://localhost:7474/db/data/node/249" ],
  "length" : 3,
  "relationships" : [ "http://localhost:7474/db/data/relationship/163", "http://localhost:7474/db/data/relationship/165",
 "http://localhost:7474/db/data/relationship/166" ],
  "end" : "http://localhost:7474/db/data/node/249"
}
```


Execute a Dijkstra algorithm with equal weights on relationships

The following is executing a Dijkstra search on a graph with equal weights on all relationships. This example is included to show the difference when the same graph structure is used, but the path weight is equal to the number of hops.

Figure 21.69. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/258/path`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "to" : "http://localhost:7474/db/data/node/255",
  "cost_property" : "cost",
  "relationships" : {
    "type" : "to",
    "direction" : "out"
```

```
  },
  "algorithm" : "dijkstra"
}
```

Example response

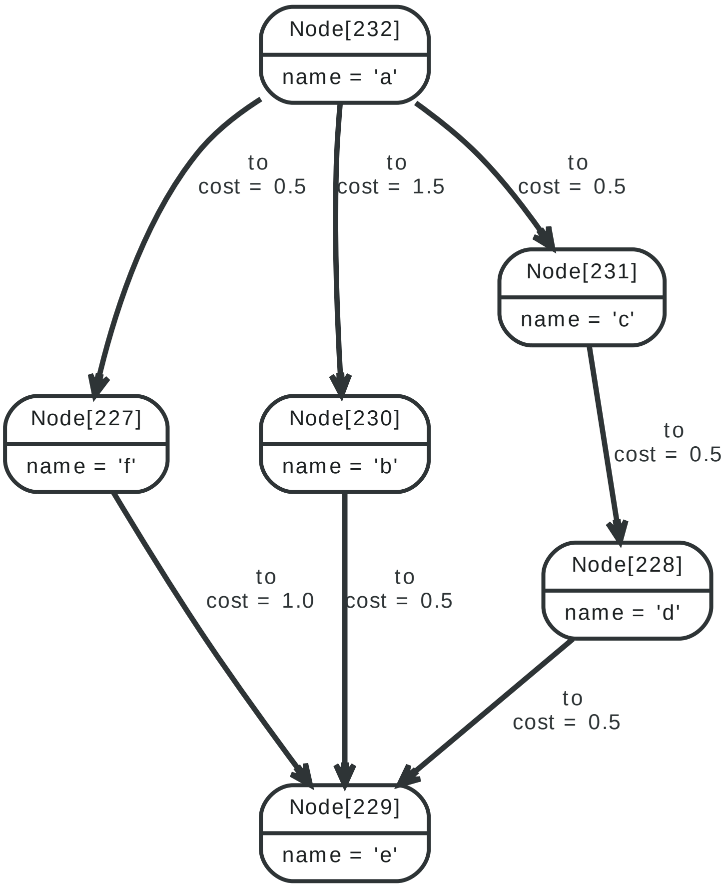
- `200: OK`
- `Content-Type: application/json; charset=UTF-8`


```
{
  "directions" : [ "->", "->" ],
  "weight" : 2.0,
  "start" : "http://localhost:7474/db/data/node/258",
  "nodes" : [ "http://localhost:7474/db/data/node/258", "http://localhost:7474/db/data/node/256", "http://localhost:7474/db/
data/node/255" ],
  "length" : 2,
  "relationships" : [ "http://localhost:7474/db/data/relationship/169", "http://localhost:7474/db/data/relationship/174" ],
  "end" : "http://localhost:7474/db/data/node/255"
}
```


Execute a Dijkstra algorithm and get multiple paths

This example is running a Dijkstra algorithm over a graph with different cost properties on different relationships. Note that the request URI ends with `/paths` which means we want multiple paths returned, in case they exist.

Figure 21.70. Final Graph



Example request

- `POST http://localhost:7474/db/data/node/232/paths`
- `Accept: application/json; charset=UTF-8`

- Content-Type: application/json


```
{
  "to" : "http://localhost:7474/db/data/node/229",
  "cost_property" : "cost",
  "relationships" : {
    "type" : "to",
    "direction" : "out"
  },
  "algorithm" : "dijkstra"
}
```


Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8


```
[ {
  "directions" : [ "->", "->", "->" ],
  "weight" : 1.5,
  "start" : "http://localhost:7474/db/data/node/232",
  "nodes" : [ "http://localhost:7474/db/data/node/232", "http://localhost:7474/db/data/node/231", "http://localhost:7474/db/
data/node/228", "http://localhost:7474/db/data/node/229" ],
  "length" : 3,
  "relationships" : [ "http://localhost:7474/db/data/relationship/136", "http://localhost:7474/db/data/relationship/138",
 "http://localhost:7474/db/data/relationship/139" ],
  "end" : "http://localhost:7474/db/data/node/229"
}, {
  "directions" : [ "->", "->" ],
  "weight" : 1.5,
  "start" : "http://localhost:7474/db/data/node/232",
  "nodes" : [ "http://localhost:7474/db/data/node/232", "http://localhost:7474/db/data/node/227", "http://localhost:7474/db/
data/node/229" ],
  "length" : 2,
  "relationships" : [ "http://localhost:7474/db/data/relationship/137", "http://localhost:7474/db/data/relationship/141" ],
  "end" : "http://localhost:7474/db/data/node/229"
} ]
```

21.19. Batch operations

Note: You cannot use this resource to execute Cypher queries with `USING PERIODIC COMMIT`.

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 `to` 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:

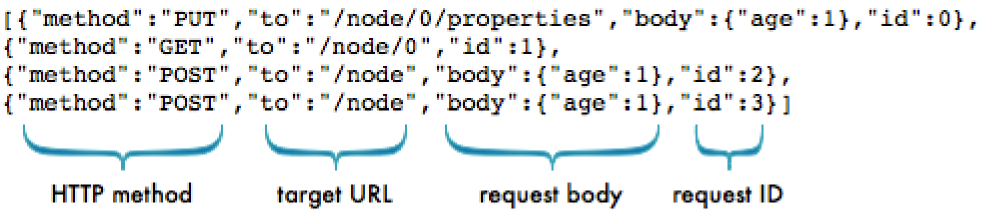
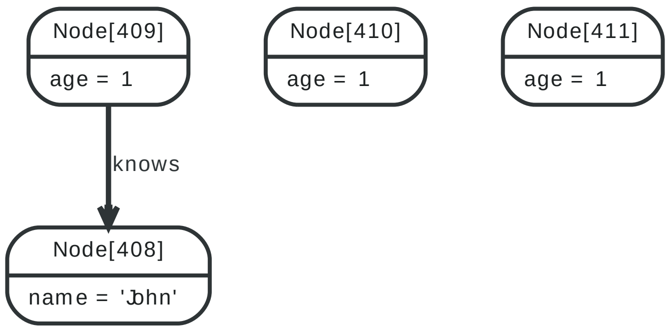


Figure 21.71. Final Graph



Example request

- `POST http://localhost:7474/db/data/batch`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
[ {
  "method" : "PUT",
  "to" : "/node/409/properties",
  "body" : {
    "age" : 1
  },
  "id" : 0
}, {
  "method" : "GET",
  "to" : "/node/409",
  "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; charset=UTF-8


```
[ {
  "id" : 0,
  "from" : "/node/409/properties"
}, {
  "id" : 1,
  "body" : {
    "extensions" : { },
    "outgoing_relationships" : "http://localhost:7474/db/data/node/409/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/409/labels",
    "traverse" : "http://localhost:7474/db/data/node/409/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/409/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/409",
    "property" : "http://localhost:7474/db/data/node/409/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/409/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/409/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/409/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/409/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/409/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/409/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/409/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 409,
      "labels" : [ ]
    },
    "data" : {
      "age" : 1
    }
  },
  "from" : "/node/409"
}, {
  "id" : 2,
  "location" : "http://localhost:7474/db/data/node/410",
  "body" : {
    "extensions" : { },
    "outgoing_relationships" : "http://localhost:7474/db/data/node/410/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/410/labels",
    "traverse" : "http://localhost:7474/db/data/node/410/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/410/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/410",
    "property" : "http://localhost:7474/db/data/node/410/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/410/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/410/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/410/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/410/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/410/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/410/relationships/all",
```

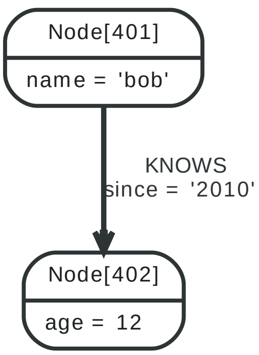
```
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/410/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 410,
      "labels" : [ ]
    },
    "data" : {
      "age" : 1
    }
  },
  "from" : "/node"
}, {
  "id" : 3,
  "location" : "http://localhost:7474/db/data/node/411",
  "body" : {
    "extensions" : { },
    "outgoing_relationships" : "http://localhost:7474/db/data/node/411/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/411/labels",
    "traverse" : "http://localhost:7474/db/data/node/411/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/411/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/411",
    "property" : "http://localhost:7474/db/data/node/411/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/411/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/411/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/411/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/411/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/411/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/411/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/411/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 411,
      "labels" : [ ]
    },
    "data" : {
      "age" : 1
    }
  },
  "from" : "/node"
} ]
```


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 21.72. Final Graph



Example request

- POST `http://localhost:7474/db/data/batch`

* Accept: application/json; charset=UTF-8
* 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; charset=UTF-8

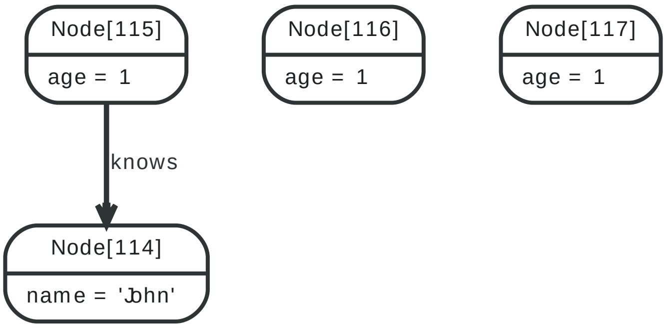
```
[ {
  "id" : 0,
  "location" : "http://localhost:7474/db/data/node/401",
  "body" : {
    "extensions" : { },
    "outgoing_relationships" : "http://localhost:7474/db/data/node/401/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/401/labels",
    "traverse" : "http://localhost:7474/db/data/node/401/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/401/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/401",
    "property" : "http://localhost:7474/db/data/node/401/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/401/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/401/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/401/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/401/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/401/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/401/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/401/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 401,
```

```
      "labels" : [ ]
    },
    "data" : {
      "name" : "bob"
    }
  },
  "from" : "/node"
}, {
  "id" : 1,
  "location" : "http://localhost:7474/db/data/node/402",
  "body" : {
    "extensions" : { },
    "outgoing_relationships" : "http://localhost:7474/db/data/node/402/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/402/labels",
    "traverse" : "http://localhost:7474/db/data/node/402/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/402/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/402",
    "property" : "http://localhost:7474/db/data/node/402/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/402/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/402/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/402/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/402/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/402/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/402/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/402/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 402,
      "labels" : [ ]
    },
    "data" : {
      "age" : 12
    }
  },
  "from" : "/node"
}, {
  "id" : 3,
  "location" : "http://localhost:7474/db/data/relationship/239",
  "body" : {
    "extensions" : { },
    "start" : "http://localhost:7474/db/data/node/401",
    "property" : "http://localhost:7474/db/data/relationship/239/properties/{key}",
    "self" : "http://localhost:7474/db/data/relationship/239",
    "properties" : "http://localhost:7474/db/data/relationship/239/properties",
    "type" : "KNOWS",
    "end" : "http://localhost:7474/db/data/node/402",
    "metadata" : {
      "id" : 239,
      "type" : "KNOWS"
    },
    "data" : {
      "since" : "2010"
    }
  },
  "from" : "http://localhost:7474/db/data/node/401/relationships"
}, {
  "id" : 4,
  "location" : "http://localhost:7474/db/data/index/relationship/my_rels/since/2010/239",
  "body" : {
    "extensions" : { },
    "start" : "http://localhost:7474/db/data/node/401",
    "property" : "http://localhost:7474/db/data/relationship/239/properties/{key}",
    "self" : "http://localhost:7474/db/data/relationship/239",
    "properties" : "http://localhost:7474/db/data/relationship/239/properties",
    "type" : "KNOWS",
    "end" : "http://localhost:7474/db/data/node/402",
```

```
    "metadata" : {
      "id" : 239,
      "type" : "KNOWS"
    },
    "data" : {
      "since" : "2010"
    },
    "indexed" : "http://localhost:7474/db/data/index/relationship/my_rels/since/2010/239"
  },
  "from" : "/index/relationship/my_rels"
} ]
```

Execute multiple operations in batch streaming

Figure 21.73. Final Graph



Example request

- POST `http://localhost:7474/db/data/batch`
- `Accept: application/json`
- `Content-Type: application/json`
- `X-Stream: true`

```
[ {
  "method" : "PUT",
  "to" : "/node/115/properties",
  "body" : {
    "age" : 1
  },
  "id" : 0
}, {
  "method" : "GET",
  "to" : "/node/115",
  "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; charset=UTF-8


```
[ {
  "id" : 0,
  "from" : "/node/115/properties",
  "body" : null,
  "status" : 204
}, {
  "id" : 1,
  "from" : "/node/115",
  "body" : {
    "extensions" : { },
    "outgoing_relationships" : "http://localhost:7474/db/data/node/115/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/115/labels",
    "traverse" : "http://localhost:7474/db/data/node/115/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/115/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/115",
    "property" : "http://localhost:7474/db/data/node/115/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/115/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/115/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/115/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/115/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/115/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/115/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/115/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 115,
      "labels" : [ ]
    },
    "data" : {
      "age" : 1
    }
  },
  "status" : 200
}, {
  "id" : 2,
  "from" : "/node",
  "body" : {
    "extensions" : { },
    "outgoing_relationships" : "http://localhost:7474/db/data/node/116/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/116/labels",
    "traverse" : "http://localhost:7474/db/data/node/116/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/116/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/116",
    "property" : "http://localhost:7474/db/data/node/116/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/116/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/116/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/116/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/116/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/116/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/116/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/116/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 116,
      "labels" : [ ]
    },
    "data" : {
      "age" : 1
    }
  },
  "location" : "http://localhost:7474/db/data/node/116",
  "status" : 201
}, {
```

```
  "id" : 3,
  "from" : "/node",
  "body" : {
    "extensions" : { },
    "outgoing_relationships" : "http://localhost:7474/db/data/node/117/relationships/out",
    "labels" : "http://localhost:7474/db/data/node/117/labels",
    "traverse" : "http://localhost:7474/db/data/node/117/traverse/{returnType}",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/117/relationships/all/{-list|&|types}",
    "self" : "http://localhost:7474/db/data/node/117",
    "property" : "http://localhost:7474/db/data/node/117/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/117/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/117/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/117/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/117/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/117/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/117/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/117/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 117,
      "labels" : [ ]
    },
    "data" : {
      "age" : 1
    }
  },
  "location" : "http://localhost:7474/db/data/node/117",
  "status" : 201
} ]
```

21.20. Legacy indexing

Note
This documents the legacy indexing in Neo4j, which is no longer the preferred way to handle indexes. Consider looking at Section 21.15, "Indexing" [351].

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 the section called "Create node index with configuration" [398].

Create node index

Note
Instead of creating the index this way, you can simply start to use it, and it will be created automatically with default configuration.

Example request

- POST http://localhost:7474/db/data/index/node/
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "name" : "favorites"
}
```

Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8
- Location: http://localhost:7474/db/data/index/node/favorites/

```
{
  "template" : "http://localhost:7474/db/data/index/node/favorites/{key}/{value}"
}
```

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 37.10, "Configuration and fulltext indexes" [644] for more information on index configuration.

Example request

- POST http://localhost:7474/db/data/index/node/

- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "name" : "fulltext",
  "config" : {
    "type" : "fulltext",
    "provider" : "lucene"
  }
}
```

Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8
- Location: http://localhost:7474/db/data/index/node/fulltext/

```
{
  "template" : "http://localhost:7474/db/data/index/node/fulltext/{key}/{value}",
  "type" : "fulltext",
  "provider" : "lucene"
}
```

Delete node index

Example request

- DELETE http://localhost:7474/db/data/index/node/kvnode
- Accept: application/json; charset=UTF-8

Example response

- 204: No Content

```

```

List node indexes

Example request

- GET http://localhost:7474/db/data/index/node/
- Accept: application/json; charset=UTF-8

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

```
{
  "node_auto_index" : {
    "template" : "http://localhost:7474/db/data/index/node/node_auto_index/{key}/{value}",
    "provider" : "lucene",
    "type" : "exact"
  },
  "favorites" : {
    "template" : "http://localhost:7474/db/data/index/node/favorites/{key}/{value}",
    "provider" : "lucene",
    "type" : "exact"
  }
}
```

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 encoded as `%20`.

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.

Example request

- `POST http://localhost:7474/db/data/index/node/favorites`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
{
  "value" : "some value",
  "uri" : "http://localhost:7474/db/data/node/266",
  "key" : "some-key"
}
```

Example response

- `201: Created`
- `Content-Type: application/json; charset=UTF-8`
- `Location: http://localhost:7474/db/data/index/node/favorites/some-key/some%20value/266`

```
{
  "extensions" : { },
  "outgoing_relationships" : "http://localhost:7474/db/data/node/266/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/266/labels",
  "traverse" : "http://localhost:7474/db/data/node/266/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/266/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/266",
  "property" : "http://localhost:7474/db/data/node/266/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/266/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/266/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/266/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/266/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/266/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/266/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/266/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 266,
    "labels" : [ ]
  },
  "data" : { },
  "indexed" : "http://localhost:7474/db/data/index/node/favorites/some-key/some%20value/266"
}
```

Remove all entries with a given node from an index

Example request

- `DELETE http://localhost:7474/db/data/index/node/kvnode/271`
- `Accept: application/json; charset=UTF-8`

Example response

- 204: No Content

Remove all entries with a given node and key from an index

Example request

- DELETE http://localhost:7474/db/data/index/node/kvnode/kvkey2/274
- Accept: application/json; charset=UTF-8

Example response

- 204: No Content

Remove all entries with a given node, key and value from an index

Example request

- DELETE http://localhost:7474/db/data/index/node/kvnode/kvkey1/value1/267
- Accept: application/json; charset=UTF-8

Example response

- 204: No Content

Find node by exact match

Note
Spaces in the URI have to be encoded as %20.

Example request

- GET http://localhost:7474/db/data/index/node/favorites/key/the%2520value
- Accept: application/json; charset=UTF-8

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8


```
[ {
  "indexed" : "http://localhost:7474/db/data/index/node/favorites/key/the%2520value/282",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/282/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/282/labels",
  "data" : { },
  "traverse" : "http://localhost:7474/db/data/node/282/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/282/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/282",
  "property" : "http://localhost:7474/db/data/node/282/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/282/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/282/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/282/relationships/in",
  "extensions" : { },
```

```
  "create_relationship" : "http://localhost:7474/db/data/node/282/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/282/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/282/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/282/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 282,
    "labels" : [ ]
  }
} ]
```


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 an example of a fuzzy search over multiple keys.

See: http://lucene.apache.org/core/3_6_2/queryparsersyntax.html

Getting the results with a predefined ordering requires adding the parameter

```
order=ordering
```

where ordering is one of index, relevance or score. In this case an additional field will be added to each result, named score, that holds the float value that is the score reported by the query result.

Example request

- `GET http://localhost:7474/db/data/index/node/bobTheIndex?query=Name:Build~0.1%20AND%20Gender:Male`
- `Accept: application/json; charset=UTF-8`

Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/281/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/281/labels",
  "data" : {
    "Name" : "Builder"
  },
  "traverse" : "http://localhost:7474/db/data/node/281/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/281/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/281",
  "property" : "http://localhost:7474/db/data/node/281/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/281/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/281/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/281/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/281/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/281/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/281/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/281/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 281,
    "labels" : [ ]
  }
} ]
```

21.21. Unique Indexing

Note
As of Neo4j 2.0, unique constraints have been added. These make Neo4j enforce the uniqueness, guaranteeing that uniqueness is maintained. See the section called “Constraints” [10] for details about this. For most cases, the unique constraints should be used rather than the features described below.

For uniqueness enforcements, there are two modes:

- URL Parameter `uniqueness=get_or_create`: Create a new node/relationship and index it if no existing one can be found. If an existing node/relationship is found, discard the sent data and return the existing node/relationship.
- URL Parameter `uniqueness=create_or_fail`: Create a new node/relationship if no existing one can be found in the index. If an existing node/relationship is found, return a conflict error.

For more information, see Section 18.6, “Creating unique nodes” [280].

Get or create unique node (create)

The node is created if it doesn’t exist in the unique index already.

Example request

- `POST http://localhost:7474/db/data/index/node/people?uniqueness=get_or_create`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
{
  "key" : "name",
  "value" : "Tobias",
  "properties" : {
    "name" : "Tobias",
    "sequence" : 1
  }
}
```

Example response

- `201: Created`
- `Content-Type: application/json; charset=UTF-8`
- `Location: http://localhost:7474/db/data/index/node/people/name/Tobias/280`

```
{
  "extensions" : { },
  "outgoing_relationships" : "http://localhost:7474/db/data/node/280/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/280/labels",
  "traverse" : "http://localhost:7474/db/data/node/280/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/280/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/280",
  "property" : "http://localhost:7474/db/data/node/280/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/280/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/280/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/280/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/280/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/280/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/280/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/280/relationships/in/{-list|&|types}",
```

```
  "metadata" : {
    "id" : 280,
    "labels" : [ ]
  },
  "data" : {
    "sequence" : 1,
    "name" : "Tobias"
  },
  "indexed" : "http://localhost:7474/db/data/index/node/people/name/Tobias/280"
}
```


Get or create unique node (existing)

Here, a node is not created but the existing unique node returned, since another node is indexed with the same data already. The node data returned is then that of the already existing node.

Example request

- POST `http://localhost:7474/db/data/index/node/people?uniqueness=get_or_create`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`


```
{
  "key" : "name",
  "value" : "Peter",
  "properties" : {
    "name" : "Peter",
    "sequence" : 2
  }
}
```


Example response

- `200: OK`
- `Content-Type: application/json; charset=UTF-8`
- `Location: http://localhost:7474/db/data/index/node/people/name/Peter/270`


```
{
  "extensions" : { },
  "outgoing_relationships" : "http://localhost:7474/db/data/node/270/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/270/labels",
  "traverse" : "http://localhost:7474/db/data/node/270/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/270/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/270",
  "property" : "http://localhost:7474/db/data/node/270/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/270/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/270/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/270/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/270/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/270/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/270/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/270/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 270,
    "labels" : [ ]
  },
  "data" : {
    "sequence" : 1,
    "name" : "Peter"
  },
  "indexed" : "http://localhost:7474/db/data/index/node/people/name/Peter/270"
}
```

Create a unique node or return fail (create)

Here, in case of an already existing node, an error should be returned. In this example, no existing indexed node is found and a new node is created.

Example request

- POST http://localhost:7474/db/data/index/node/people?uniqueness=create_or_fail
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "key" : "name",
  "value" : "Tobias",
  "properties" : {
    "name" : "Tobias",
    "sequence" : 1
  }
}
```

Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8
- Location: http://localhost:7474/db/data/index/node/people/name/Tobias/279


```
{
  "extensions" : { },
  "outgoing_relationships" : "http://localhost:7474/db/data/node/279/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/279/labels",
  "traverse" : "http://localhost:7474/db/data/node/279/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/279/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/279",
  "property" : "http://localhost:7474/db/data/node/279/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/279/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/279/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/279/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/279/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/279/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/279/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/279/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 279,
    "labels" : [ ]
  },
  "data" : {
    "sequence" : 1,
    "name" : "Tobias"
  },
  "indexed" : "http://localhost:7474/db/data/index/node/people/name/Tobias/279"
}
```


Create a unique node or return fail (fail)

Here, in case of an already existing node, an error should be returned. In this example, an existing node indexed with the same data is found and an error is returned.

Example request

- POST http://localhost:7474/db/data/index/node/people?uniqueness=create_or_fail
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "key" : "name",
  "value" : "Peter",
  "properties" : {
    "name" : "Peter",
    "sequence" : 2
  }
}
```


Example response

- 409: Conflict
- Content-Type: application/json; charset=UTF-8


```
{
  "extensions" : { },
  "outgoing_relationships" : "http://localhost:7474/db/data/node/264/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/264/labels",
  "traverse" : "http://localhost:7474/db/data/node/264/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/264/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/264",
  "property" : "http://localhost:7474/db/data/node/264/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/264/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/264/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/264/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/264/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/264/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/264/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/264/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 264,
    "labels" : [ ]
  },
  "data" : {
    "sequence" : 1,
    "name" : "Peter"
  },
  "indexed" : "http://localhost:7474/db/data/index/node/people/name/Peter/264"
}
```


Add an existing node to unique index (not indexed)

Associates a node with the given key/value pair in the given unique index.

In this example, we are using `create_or_fail` uniqueness.

Example request

- POST http://localhost:7474/db/data/index/node/favorites?uniqueness=create_or_fail
- Accept: application/json; charset=UTF-8
- Content-Type: application/json


```
{
  "value" : "some value",
  "uri" : "http://localhost:7474/db/data/node/275",
  "key" : "some-key"
}
```


Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8

- Location: http://localhost:7474/db/data/index/node/favorites/some-key/some%20value/275


```
{
  "extensions" : { },
  "outgoing_relationships" : "http://localhost:7474/db/data/node/275/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/275/labels",
  "traverse" : "http://localhost:7474/db/data/node/275/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/275/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/275",
  "property" : "http://localhost:7474/db/data/node/275/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/275/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/275/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/275/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/275/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/275/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/275/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/275/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 275,
    "labels" : [ ]
  },
  "data" : { },
  "indexed" : "http://localhost:7474/db/data/index/node/favorites/some-key/some%20value/275"
}
```


Add an existing node to unique index (already indexed)

In this case, the node already exists in the index, and thus we get a `HTTP 409` status response, as we have set the uniqueness to `create_or_fail`.

Example request

- POST http://localhost:7474/db/data/index/node/favorites?uniqueness=create_or_fail
- Accept: application/json; charset=UTF-8
- Content-Type: application/json


```
{
  "value" : "some value",
  "uri" : "http://localhost:7474/db/data/node/278",
  "key" : "some-key"
}
```


Example response

- 409: Conflict
- Content-Type: application/json; charset=UTF-8


```
{
  "extensions" : { },
  "outgoing_relationships" : "http://localhost:7474/db/data/node/277/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/277/labels",
  "traverse" : "http://localhost:7474/db/data/node/277/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/277/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/277",
  "property" : "http://localhost:7474/db/data/node/277/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/277/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/277/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/277/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/277/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/277/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/277/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/277/relationships/in/{-list|&|types}",
```

```
  "metadata" : {
    "id" : 277,
    "labels" : [ ]
  },
  "data" : {
    "some-key" : "some value"
  },
  "indexed" : "http://localhost:7474/db/data/index/node/favorites/some-key/some%20value/277"
}
```

Get or create unique relationship (create)

Create a unique relationship in an index. If a relationship matching the given key and value already exists in the index, it will be returned. If not, a new relationship will be created.

Note
The type and direction of the relationship is not regarded when determining uniqueness.

Example request

- POST `http://localhost:7474/db/data/index/relationship/MyIndex/?uniqueness=get_or_create`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
{
  "key" : "name",
  "value" : "Tobias",
  "start" : "http://localhost:7474/db/data/node/31",
  "end" : "http://localhost:7474/db/data/node/32",
  "type" : "knowledge"
}
```

Example response

- `201: Created`
- `Content-Type: application/json; charset=UTF-8`
- `Location: http://localhost:7474/db/data/index/relationship/MyIndex/name/Tobias/8`

```
{
  "extensions" : { },
  "start" : "http://localhost:7474/db/data/node/31",
  "property" : "http://localhost:7474/db/data/relationship/8/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/8",
  "properties" : "http://localhost:7474/db/data/relationship/8/properties",
  "type" : "knowledge",
  "end" : "http://localhost:7474/db/data/node/32",
  "metadata" : {
    "id" : 8,
    "type" : "knowledge"
  },
  "data" : {
    "name" : "Tobias"
  },
  "indexed" : "http://localhost:7474/db/data/index/relationship/MyIndex/name/Tobias/8"
}
```

Get or create unique relationship (existing)

Here, in case of an already existing relationship, the sent data is ignored and the existing relationship returned.

Example request

- POST http://localhost:7474/db/data/index/relationship/rels?uniqueness=get_or_create
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "key" : "name",
  "value" : "Peter",
  "start" : "http://localhost:7474/db/data/node/35",
  "end" : "http://localhost:7474/db/data/node/36",
  "type" : "KNOWS"
}
```

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

```
{
  "extensions" : { },
  "start" : "http://localhost:7474/db/data/node/33",
  "property" : "http://localhost:7474/db/data/relationship/9/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/9",
  "properties" : "http://localhost:7474/db/data/relationship/9/properties",
  "type" : "KNOWS",
  "end" : "http://localhost:7474/db/data/node/34",
  "metadata" : {
    "id" : 9,
    "type" : "KNOWS"
  },
  "data" : { },
  "indexed" : "http://localhost:7474/db/data/index/relationship/rels/name/Peter/9"
}
```

Create a unique relationship or return fail (create)

Here, in case of an already existing relationship, an error should be returned. In this example, no existing relationship is found and a new relationship is created.

Example request

- POST http://localhost:7474/db/data/index/relationship/rels?uniqueness=create_or_fail
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "key" : "name",
  "value" : "Tobias",
  "start" : "http://localhost:7474/db/data/node/43",
  "end" : "http://localhost:7474/db/data/node/44",
  "type" : "KNOWS"
}
```

Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8
- Location: http://localhost:7474/db/data/index/relationship/rels/name/Tobias/12

```
{
```

```
  "extensions" : { },
  "start" : "http://localhost:7474/db/data/node/43",
  "property" : "http://localhost:7474/db/data/relationship/12/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/12",
  "properties" : "http://localhost:7474/db/data/relationship/12/properties",
  "type" : "KNOWS",
  "end" : "http://localhost:7474/db/data/node/44",
  "metadata" : {
    "id" : 12,
    "type" : "KNOWS"
  },
  "data" : {
    "name" : "Tobias"
  },
  "indexed" : "http://localhost:7474/db/data/index/relationship/rels/name/Tobias/12"
}
```

Create a unique relationship or return fail (fail)

Here, in case of an already existing relationship, an error should be returned. In this example, an existing relationship is found and an error is returned.

Example request

- `POST` `http://localhost:7474/db/data/index/relationship/rels?uniqueness=create_or_fail`
- `Accept:` `application/json; charset=UTF-8`
- `Content-Type:` `application/json`

```
{
  "key" : "name",
  "value" : "Peter",
  "start" : "http://localhost:7474/db/data/node/23",
  "end" : "http://localhost:7474/db/data/node/24",
  "type" : "KNOWS"
}
```

Example response

- `409:` `Conflict`
- `Content-Type:` `application/json; charset=UTF-8`

```
{
  "extensions" : { },
  "start" : "http://localhost:7474/db/data/node/21",
  "property" : "http://localhost:7474/db/data/relationship/4/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/4",
  "properties" : "http://localhost:7474/db/data/relationship/4/properties",
  "type" : "KNOWS",
  "end" : "http://localhost:7474/db/data/node/22",
  "metadata" : {
    "id" : 4,
    "type" : "KNOWS"
  },
  "data" : { },
  "indexed" : "http://localhost:7474/db/data/index/relationship/rels/name/Peter/4"
}
```

Add an existing relationship to a unique index (not indexed)

If a relationship matching the given key and value already exists in the index, it will be returned. If not, an `HTTP 409` (conflict) status will be returned in this case, as we are using `create_or_fail`.

It's possible to use `get_or_create` uniqueness as well.

Note
The type and direction of the relationship is not regarded when determining uniqueness.

Example request

- POST http://localhost:7474/db/data/index/relationship/rels?uniqueness=create_or_fail
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "key" : "name",
  "value" : "Peter",
  "uri" : "http://localhost:7474/db/data/relationship/3"
}
```

Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8
- Location: http://localhost:7474/db/data/index/relationship/rels/name/Peter/3

```
{
  "extensions" : { },
  "start" : "http://localhost:7474/db/data/node/19",
  "property" : "http://localhost:7474/db/data/relationship/3/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/3",
  "properties" : "http://localhost:7474/db/data/relationship/3/properties",
  "type" : "KNOWS",
  "end" : "http://localhost:7474/db/data/node/20",
  "metadata" : {
    "id" : 3,
    "type" : "KNOWS"
  },
  "data" : { },
  "indexed" : "http://localhost:7474/db/data/index/relationship/rels/name/Peter/3"
}
```

Add an existing relationship to a unique index (already indexed)

Example request

- POST http://localhost:7474/db/data/index/relationship/rels?uniqueness=create_or_fail
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "key" : "name",
  "value" : "Peter",
  "uri" : "http://localhost:7474/db/data/relationship/6"
}
```

Example response

- 409: Conflict
- Content-Type: application/json; charset=UTF-8

```
{
  "extensions" : { },
```

```
  "start" : "http://localhost:7474/db/data/node/25",
  "property" : "http://localhost:7474/db/data/relationship/5/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/5",
  "properties" : "http://localhost:7474/db/data/relationship/5/properties",
  "type" : "KNOWS",
  "end" : "http://localhost:7474/db/data/node/26",
  "metadata" : {
    "id" : 5,
    "type" : "KNOWS"
  },
  "data" : { },
  "indexed" : "http://localhost:7474/db/data/index/relationship/rels/name/Peter/5"
}
```

21.22. Legacy Automatic Indexes

To enable automatic indexes, set up the database for that, see the section called "Configuration" [647]. With this feature enabled, you can then index and query nodes in these indexes.

Find node by exact match from an automatic index

Automatic index nodes can be found via exact lookups with normal Index REST syntax.

Example request

- GET http://localhost:7474/db/data/index/auto/node/name/I
- Accept: application/json; charset=UTF-8

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/9/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/9/labels",
  "data" : {
    "name" : "I"
  },
  "traverse" : "http://localhost:7474/db/data/node/9/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/9/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/9",
  "property" : "http://localhost:7474/db/data/node/9/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/9/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/9/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/9/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/9/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/9/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/9/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/9/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 9,
    "labels" : [ ]
  }
} ]
```


Find node by query from an automatic index

See Find node by query for the actual query syntax.

Example request

- GET http://localhost:7474/db/data/index/auto/node/?query=name:I
- Accept: application/json; charset=UTF-8

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8


```
[ {
  "outgoing_relationships" : "http://localhost:7474/db/data/node/0/relationships/out",
  "labels" : "http://localhost:7474/db/data/node/0/labels",
```

```
  "data" : {
    "name" : "I"
  },
  "traverse" : "http://localhost:7474/db/data/node/0/traverse/{returnType}",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/0/relationships/all/{-list|&|types}",
  "self" : "http://localhost:7474/db/data/node/0",
  "property" : "http://localhost:7474/db/data/node/0/properties/{key}",
  "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}",
  "metadata" : {
    "id" : 0,
    "labels" : [ ]
  }
} ]
```

21.23. Configurable Legacy Automatic Indexing

Out of the box auto-indexing supports exact matches since they are created with the default configuration (see Section 37.12, "Automatic Indexing" [647]) 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.

> **Warning**
> 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 *conf/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` settings 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 23.2, "Server Installation" [426].

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:

Create an auto index for nodes with specific configuration

Example request

- `POST http://localhost:7474/db/data/index/node/`
- `Accept: application/json; charset=UTF-8`
- `Content-Type: application/json`

```
{
  "name" : "node_auto_index",
  "config" : {
    "type" : "fulltext",
    "provider" : "lucene"
  }
}
```

Example response

- `201: Created`
- `Content-Type: application/json; charset=UTF-8`
- `Location: http://localhost:7474/db/data/index/node/node_auto_index/`

```
{
  "template" : "http://localhost:7474/db/data/index/node/node_auto_index/{key}/{value}",
  "type" : "fulltext",
  "provider" : "lucene"
}
```

If you require configured auto-indexes for relationships, the approach is similar:

Create an auto index for relationships with specific configuration

Example request

- POST http://localhost:7474/db/data/index/relationship/
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "name" : "relationship_auto_index",
  "config" : {
    "type" : "fulltext",
    "provider" : "lucene"
  }
}
```

Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8
- Location: http://localhost:7474/db/data/index/relationship/relationship_auto_index/

```
{
  "template" : "http://localhost:7474/db/data/index/relationship/relationship_auto_index/{key}/{value}",
  "type" : "fulltext",
  "provider" : "lucene"
}
```

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 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.

There are a few REST calls providing a REST interface to the AutoIndexer[8] component. The following REST calls work both, for `node` and `relationship` by simply changing the respective part of the URL.

Get current status for autoindexing on nodes

Example request

- GET http://localhost:7474/db/data/index/auto/node/status
- Accept: application/json; charset=UTF-8

Example response

[8] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/AutoIndexer.html

- `200`: `OK`
- `Content-Type`: `application/json; charset=UTF-8`

```
false
```

Enable node autoindexing

Example request

- `PUT` `http://localhost:7474/db/data/index/auto/node/status`
- `Accept`: `application/json; charset=UTF-8`
- `Content-Type`: `application/json`

```
true
```

Example response

- `204`: `No Content`

```
```

Lookup list of properties being autoindexed

Example request

- `GET` `http://localhost:7474/db/data/index/auto/node/properties`
- `Accept`: `application/json; charset=UTF-8`

Example response

- `200`: `OK`
- `Content-Type`: `application/json; charset=UTF-8`

```
[ "some-property" ]
```

Add a property for autoindexing on nodes

Example request

- `POST` `http://localhost:7474/db/data/index/auto/node/properties`
- `Accept`: `application/json; charset=UTF-8`
- `Content-Type`: `application/json`

```
myProperty1
```

Example response

- `204`: `No Content`

```
```

Remove a property for autoindexing on nodes

Example request

- `DELETE` `http://localhost:7474/db/data/index/auto/node/properties/myProperty1`
- `Accept`: `application/json; charset=UTF-8`

Example response

- `204`: `No Content`

21.24. WADL Support

The Neo4j REST API is a truly RESTful interface relying on hypermedia controls (links) to advertise permissible actions to users. Hypermedia is a dynamic interface style where declarative constructs (semantic markup) are used to inform clients of their next legal choices just in time.

Caution
RESTful APIs cannot be modeled by static interface description languages like WSDL or WADL.

However for some use cases, developers may wish to expose WADL descriptions of the Neo4j REST API, particularly when using tooling that expects such.

In those cases WADL generation may be enabled by adding to your server's *conf/neo4j.properties* file:

```
unsupported_wadl_generation_enabled=true
```

Caution
WADL is not an officially supported part of the Neo4j server API because WADL is insufficiently expressive to capture the set of potential interactions a client can drive with Neo4j server. Expect the WADL description to be incomplete, and in some cases contradictory to the real API. In any cases where the WADL description disagrees with the REST API, the REST API should be considered authoritative. WADL generation may be withdrawn at any point in the Neo4j release cycle.

21.25. Using the REST API from WebLogic

When deploying an application to WebLogic you may run into problems when Neo4j responds with an HTTP status of `204 No Content`. The response does not contain an entity body in such cases.

This can cause WebLogic to throw `java.net.SocketTimeoutException: Read timed out` for no obvious reason.

If you encounter this, please try setting `UseSunHttpHandler` to `true`. You can for example do this by adding the following to the WebLogic startup script:

```
-DUseSunHttpHandler=true
```

The WebLogic startup script is called `bin\startWebLogic.sh` (`bin/startWebLogic.cmd` on Windows).

Chapter 22. Deprecations

This section outlines deprecations in Neo4j 2.2.1 or earlier in order to help you find a smoother transition path to future releases.

Cypher ExecutionEngine	There's no need to use `ExecutionEngine` any more: instead, use the execute[1] methods on `GraphDatabaseService`. `ExecutionEngine` and the related classes/interfaces have been deprecated, see the javadocs for details.
Embedded Java API	See Deprecated list in Javadoc[2].
Graph Matching	The graph-matching component will be removed in future releases.

[1] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/GraphDatabaseService.html#execute(java.lang.String)
[2] http://neo4j.com/docs/2.2.1/javadocs/deprecated-list.html

Part V. Operations

This part describes how to install and 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 23. Installation & Deployment

Neo4j is accessed as a standalone server, either directly through a REST interface or through a language-specific driver.

Neo4j can be installed as a server, running either as a headless application or system service. For information on installing The Neo4j Server, see Section 23.2, “Server Installation” [426].

For running Neo4j in high availability mode, see Chapter 25, *High Availability* [468].

23.1. System Requirements

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

CPU

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

Minimum	Intel Core i3
Recommended	Intel Core i7

Memory

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

Minimum	2GB
Recommended	16—32GB or more

Disk

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

Minimum	10GB SATA
Recommended	SSD w/ SATA

Filesystem

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

Minimum	ext4 (or similar)
Recommended	ext4, ZFS

Software

Neo4j is Java-based.

Java	OpenJDK 7[1] or Oracle Java 7[2]
Operating Systems	Linux, HP UX, Windows 2008 for production; additionally Windows XP, Mac OS X for development.

[1] http://openjdk.java.net/
[2] http://www.oracle.com/technetwork/java/javase/downloads/index.html

23.2. Server Installation

Deployment Scenarios

As a developer, you may wish to download Neo4j and run it locally on your desktop computer. We recommend this as an easy way to discover Neo4j.

- For Windows, see the section called “Windows” [426].
- For Unix/Linux, see the section called “Linux” [427].
- For OSX, see the section called “Mac OSX” [427].

As a systems administrator, you may wish to install Neo4j using a packaging system so you can ensure that a cluster of machines have identical installs. See the section called “Linux Packages” [427] for more information on this.

For information on High Availability, please refer to Chapter 25, *High Availability* [468].

Prerequisites

With the exception of our Windows Installer, you'll need a Java Virtual Machine installed on your computer. We recommend that you install OpenJDK 7[3] or Oracle Java 7[4].

Windows

Windows Installer

1. Download the version that you want from http://neo4j.com/download/.
   - Select the appropriate version and architecture for your platform.
2. Double-click the downloaded installer file.
3. Follow the prompts.

Note
The installer will prompt to be granted Administrator privileges. Newer versions of Windows come with a SmartScreen feature that may prevent the installer from running — you can make it run anyway by clicking "More info" on the "Windows protected your PC" screen.

Tip
If you install Neo4j using the windows installer and you already have an existing Neo4j installed it will ask if you want to upgrade. This should proceed without issue although some users have reported a `JRE is damaged` exception. If you see this error simply install Neo4j into a different location.

Windows Console Application

1. Download the latest release from http://neo4j.com/download/.
   - Select the appropriate Zip distribution.
2. Right-click the downloaded file, click Extract All.
   - Refer to the top-level extracted directory as: `NEO4J_HOME`
3. Double-click on *%NEO4J_HOME%\bin\Neo4j.bat*
4. Stop the server by typing Ctrl-C in the console that got opened.

Note
Some users have reported problems on Windows when using the ZoneAlarm firewall. If you are having problems getting large responses from the server, or if the web interface does

[3] http://openjdk.java.net/
[4] http://www.oracle.com/technetwork/java/javase/downloads/index.html

not work, try disabling ZoneAlarm. Contact ZoneAlarm support to get information on how to resolve this.

Linux

Linux Packages

- For Debian packages, see the instructions at http://debian.neo4j.org.

After installation you may have to do some platform specific configuration and performance tuning. For that, refer to Section 24.12, “Linux Performance Guide” [464].

Unix Console Application

1. Download the latest release from http://neo4j.com/download/.
   - Select the appropriate tar.gz distribution for your platform.
2. Extract the contents of the archive, using: `tar -xf <filename>`
   - Refer to the top-level extracted directory as: `NEO4J_HOME`
3. Change directory to: `$NEO4J_HOME`
   - Run: `./bin/neo4j console`
4. Stop the server by typing Ctrl-C in the console.

Linux Service

The `neo4j` command can also be used with `start`, `stop`, `restart` or `status` instead of `console`. By using these actions, you can create a Neo4j service. See the neo4j man page for further details.

Caution
This approach to running Neo4j as a server is deprecated. We strongly advise you to run Neo4j from a package where feasible.

You can build your own `init.d` script. See for instance the Linux Standard Base specification on system initialization[5], or one of the many samples[6] and tutorials[7].

Mac OSX

Running Neo4j from the Terminal

The server can be started in the background from the terminal with the command `neo4j start`, and then stopped again with `neo4j stop`. The server can also be started in the foreground with `neo4j console` — then it’s log output will be printed to the terminal.

The `neo4j-shell` command can be used to interact with Neo4j from the command line using Cypher. It will automatically connect to any server that is running on localhost with the default port, otherwise it will show a help message. You can alternatively start the shell with an embedded Neo4j instance, by using the `-path path/to/data` argument — note that only a single instance of Neo4j can access the database files at a time.

OSX Service

Use the standard OSX system tools to create a service based on the `neo4j` command.

A note on Java on OS X Mavericks

Unlike previous versions, OS X Mavericks does not come with Java pre-installed. You might encounter that the first time you run Neo4j, where OS X will trigger a popup offering you to install Java SE 6.

5 http://refspecs.linuxfoundation.org/LSB_3.1.0/LSB-Core-generic/LSB-Core-generic/tocsysinit.html
6 https://gist.github.com/chrisvest/7673244
7 http://www.linux.com/learn/tutorials/442412-managing-linux-daemons-with-init-scripts

Java SE 6 is incompatible with Neo4j 2.2.1, so we strongly advise you to skip installing Java SE 6 if you have no other uses for it. Instead, for Neo4j 2.2.1 we recommend you install Java SE 7 from Oracle (http://www.oracle.com/technetwork/java/javase/downloads/index.html) as that is what we support for production use.

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.

For how to set this up, see the section called “Alternative setup: Creating a local cluster for testing” [482]. Just use the Neo4j edition of your choice, follow the guide and remember to not set the servers to run in HA mode.

23.3. Server Installation in the Cloud

Neo4j can be hosted on various cloud services. See http://www.neo4j.org/develop/cloud for more information.

23.4. Upgrading

Important
This section describes upgrading a single Neo4j instance. Upgrading a Neo4j HA cluster (Neo4j Enterprise) requires a very specific process be followed. Please refer to Section 25.5, “Upgrade of a Neo4j HA Cluster” [477].

A database created by an older version of Neo4j will be upgraded during startup when opened by Neo4j 2.2.1. Whilst this upgrade will occur automatically for minor changes, larger migrations require explicit configuration before Neo4j will start.

Each Neo4j version supports upgrading from a limited number of previous versions. These upgrades are either automatic, or require explicit configuration to allow them.

In this release of Neo4j, the following upgrades are permitted.

1.9.x → 2.2.1	Explicit configuration is required
2.0.x → 2.2.1	Explicit configuration is required
2.1.x → 2.2.1	Automatically performs any store upgrade

Note
Downgrade is only supported between Neo4j versions that allow for automatic store upgrades. This typically means only within patch releases of the same Neo4j version.

Automatic Store Upgrade

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

1. Cleanly shut down the older version of Neo4j, if it is running.
2. Install Neo4j 2.2.1, and set it up to use the same database store directory (typically `data/graph.db`).
3. Make a copy of the database.

   Important
   It is strongly advised to make a copy of the database store directory at this time, to use as a backup in case rollback/downgrade is required. This is not necessary if a backup has been made using the online backup tool, available with Neo4j Enterprise.
4. Start up Neo4j.
5. Any database store upgrade required will occur during startup.

Explicit Store Upgrade

To perform an explicit store upgrade (required for significant changes to the database store):

1. Install Neo4j 2.2.1, and set it up to use the same database store directory (typically *data/graph.db*).
2. Cleanly shut down the older version of Neo4j, if it is running.
3. Set the Neo4j configuration parameter `allow_store_upgrade=true` in your *conf/neo4j.properties* file. Neo4j will fail to start without this configuration set.
4. Start up Neo4j.
5. The database store upgrade will occur during startup.
6. The `allow_store_upgrade` configuration parameter should be removed, set to `false` or commented out.
7. Information about the upgrade and a progress indicator are logged into the *messages.log* file, inside the database store directory.

Caution
An explicit upgrade will require substantial free disk space, as it must make an entire copy of the database store. The upgraded store version may also require larger store files overall. It

is suggested to have available free disk space equivalent to at least 1.5 times the size of the existing store.

Note
Cypher compatibility: The Cypher language is rapidly evolving, and may change between Neo4j versions (although not between patch releases). However, Neo4j supports compatibility directives for Cypher, that allow explicitly selecting a language version. This is possible to do for individual statements, or globally, as described in the Cypher Compatibility section.

23.5. 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=-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005 \
   -Xdebug-Xnoagent-Djava.compiler=NONE\
   -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5005
```

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.

23.6. 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.

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: 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.
- Registration id: For registered server instances.
- Tags about the execution context (e.g. test, language, web-container, app-container, spring, ejb).
- Neo4j Edition (community, enterprise).
- A hash of the current cluster name (if any).
- Distribution information for Linux (rpm, dpkg, unknown).
- User-Agent header for tracking usage of REST client drivers
- MAC address to uniquely identify instances behind firewalls.
- The number of processors on the server.
- The amount of memory on the server.
- The JVM heap size.
- The number of nodes, relationships, labels and properties in the database.

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.

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>2.2.1</version>
  <exclusions>
    <exclusion>
      <groupId>org.neo4j</groupId>
      <artifactId>neo4j-udc</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

3. Lastly, if you are using a packaged version of Neo4j, and do not want to make any change to the jars, see below for the setting to use in *conf/neo4j.properties* to disable UDC.

Usage Data Collector configuration settings

- neo4j.ext.udc.enabled: Enable the UDC extension.

neo4j.ext.udc.enabled

Description	Enable the UDC extension.
Valid values	is a boolean.
Default value	`true`

Chapter 24. 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.

24.1. 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 24.8, "JVM Settings" [458]). 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).
- Give the Neo4j page cache as much memory as you can spare. After configuring the JVM heap size, you can leave 2-4GBs for the operating system (assuming the machine is dedicated to running Neo4j), and assign the rest to the Neo4j page cache with the `dbms.pagecache.memory` setting, unless you know your store will be small enough to fit in less.

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, see the section called "Starting an embedded database with configuration settings" [592] for an example.

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.MainClass
```

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 24.2, "Server Configuration" [437] for how to add configuration settings for the database to the server.

24.2. Server Configuration

Quick info

- The server's primary configuration file is found under *conf/neo4j-server.properties*
- Low-level performance tuning parameters and configuration of legacy indexes and the remote shell are found in *conf/neo4j.properties*
- Configuration of the daemonizing wrapper is found in *conf/neo4j-wrapper.properties*
- HTTP logging configuration is found in *conf/neo4j-http-logging.xml*

Important server configuration 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 Chapter 27, *Security* [496]

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=-Djava.net.preferIPv4Stack=true
```

Specify the number of threads used by the Neo4j Web server to control the level of concurrent HTTP requests that the server will service.

```
org.neo4j.server.webserver.maxthreads=200
```

Note
The default value is the number of CPUs reported available by the JVM, limited to a maximum of 500. The limit can be exceeded by specifying a larger value.

The server guards against orphaned transactions by using a timeout. If there are no requests for a given transaction within the timeout period, the server will roll it back. You can configure the timeout period by setting the following property to the number of seconds before timeout. The default timeout is 60 seconds.

```
org.neo4j.server.transaction.timeout=60
```

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.

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 24.3, "Server Performance Tuning" [441]. 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.

Server logging configuration

Application events within Neo4j server are processed with `java.util.logging`[1] and configured in the file *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`[2].

Apart from log statements originating from the Neo4j server, other libraries report their messages through various frameworks.

[1] http://download.oracle.com/javase/6/docs/technotes/guides/logging/overview.html
[2] http://download.oracle.com/javase/7/docs/api/java/util/logging/FileHandler.html

HTTP logging configuration

As well as logging events happening within the Neo4j server, it is possible to log the HTTP requests and responses that the server consumes and produces. Configuring HTTP logging requires operators to enable and configure the logger and where it will log; and then to optionally configure the log format.

Important

By default the HTTP logger uses `Common Log Format`[3] meaning that most Web server tooling can automtically consume such logs. In general users should only enable HTTP logging, select an output directory, and if necessary alter the rollover and retention policies.

To enable HTTP logging, edit the *conf/neo4j-server.properties* file to resemble the following:

```
org.neo4j.server.http.log.enabled=true
org.neo4j.server.http.log.config=conf/neo4j-http-logging.xml
```

Using `org.neo4j.server.http.log.enabled=true` tells the server that HTTP logging is enabled. HTTP logging can be disabled completely by setting this property to `false`. The setting `org.neo4j.server.http.log.config=conf/neo4j-http-logging.xml` specifies the logging format and rollover policy file that governs how HTTP log output is presented and archived. The defaults provided with Neo4j server uses an hourly log rotation and Common Log Format.

If logging is set up to use log files then the server will check that the log file directory exists and is writable. If this check fails, then the server will not start and will report the failure to another available channel like standard out.

Tip

Neo4j server now has **experimental** support for logging full request and response bodies. It is enabled by setting the following property in *neo4j-server.properties*:

```
org.neo4j.server.http.unsafe.content_log.enabled=true
```

The following logging pattern must also be specified in *neo4j-http-logging.xml*:

```
<pattern>%fullRequest\n\n%fullResponse</pattern>
```

This functionality fully duplicates HTTP requests and responses, logging them out to disk. As such it is **strongly advised** to not run this in a production setting because of the potential to constrain performance. However it can prove useful in testing and pre-production environments.

Using X-Forwarded-Proto and X-Forwarded-Host to parameterize the base URI for REST responses

There are occasions, for example when you want to host Neo4j server behind a proxy (e.g. one that handles HTTPS traffic), and still have Neo4j respect the base URI of that externally visible proxy.

Ordinarily Neo4j uses the `HOST` header of the HTTP request to construct URIs in its responses. Where a proxy is involved however, this is often undesirable. Instead Neo4j uses the `X-Forwarded-Host` and `X-Forwarded-Proto` headers provided by proxies to parameterize the URIs in the responses from the database's REST API. From the outside it looks as if the proxy generated that payload. If an `X-Forwarded-Host` header value contains more than one address (`X-Forwarded-Host` allows comma-and-space separated lists of addresses), Neo4j picks the first, which represents the client request.

In order to take advantage of this functionality your proxy server must be configured to transmit these headers to the Neo4j server. Failure to transmit both `X-Forwarded-Host` and `X-Forwarded-Proto` headers will result in the original base URI being used.

[3] http://en.wikipedia.org/wiki/Common_Log_Format

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 the following value in *conf/neo4j-wrapper.conf*:

```
wrapper.java.additional=-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 the option to.

Disabling console types in Webadmin

You may, for security reasons, want to disable the the Neo4j Shell in Webadmin. Shells allow arbitrary code execution, and so they could constitute a security risk if you do not trust all users of your Neo4j Server.

In the *conf/neo4j-server.properties* file:

```
# To disable all shells:
org.neo4j.server.manage.console_engines=

# To enable only the Neo4j Shell:
org.neo4j.server.manage.console_engines=shell
```

24.3. 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 *conf/neo4j.properties* file is a standard configuration file that databases load in order to tune their memory use and caching strategies.
- See Section 24.6, “Caches in Neo4j” [452] for more information.

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.

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 24.8, “JVM Settings” [458].

24.4. Performance Guide

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

Try this first

The first thing is to make sure the JVM is running well and not spending too much time in garbage collection. Monitoring heap usage of 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 a 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 out some different heap sizes. Make sure a parallel/concurrent garbage collector is running (`-XX:+UseConcMarkSweepGC` works well in most use-cases).

Finally make sure that the OS has some memory left to manage proper file system caches. This means, if your server has 8GB of RAM don't use all of that RAM for heap (unless you have turned off memory mapped buffers), but leave a good part of it to the OS. For more information on configuration see Chapter 24, *Configuration & Performance* [435].

For Linux specific tweaks, see Section 24.12, "Linux Performance Guide" [464].

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. The first time, after loading a node or relationship, that any property is accessed all the properties are loaded for that entity. If any of those properties contain an array larger than a few elements or a long string such values are loaded on demand when requesting them individually. Similarly, relationships of a node will only be loaded the first time they are requested for that node.

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 a 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.

Configuring Neo4j

See Chapter 24, *Configuration & Performance* [435] and Section 24.8, "JVM Settings" [458] for information on how to configure Neo4j and the JVM. These settings have a lot of impact on performance.

Disks, RAM and other tips

As always, as with any persistence solution, performance depends a lot 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 the available amount of RAM is too small or the configuration for caches and memory mapping is bad. 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.

Use tools like `vmstat` or equivalent to gather information 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 it's a sign that you need to tweak your Java heap, Neo4j cache and memory mapping 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 that is 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 slowing 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. For details about transaction management in Neo4j, please see Chapter 18, *Transaction Management* [272].

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 modifying 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.

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 Section 24.12, "Linux Performance Guide" [464] 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 compared 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 extent 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 have 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 with this is that now we need to invalidate the cached property whenever `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.

24.5. Configuration Settings

On this page you'll find the main configuration settings you can use with Neo4j. They can be set in the *conf/neo4j.properties* file when using the Neo4j Server (see Section 24.2, "Server Configuration" [437]). If you use the embedded database, you can pass them in as a map (see the section called "Starting an embedded database with configuration settings" [592]).

For additional configuration settings, see:

- Section 24.2, "Server Configuration" [437]
- Settings for the remote shell extension [529]
- High Performance Cache configuration settings [455]
- High Availability configuration settings [470]
- Cluster configuration settings [472]
- Online backup configuration settings [491]
- Consistency check configuration settings [491]
- Usage Data Collector configuration settings [434]

List of configuration settings

- allow_file_urls: Determines if Cypher will allow using file URLs when loading data using `LOAD CSV`.
- allow_store_upgrade: Whether to allow a store upgrade in case the current version of the database starts against an older store version.
- batched_writes: Whether or not transactions are appended to the log in batches.
- cache_type: The type of cache to use for nodes and relationships.
- cypher_parser_version: Set this to specify the default parser.
- dbms.cypher.min_replan_interval: The minimum lifetime of a query plan before a query is considered for replanning.
- dbms.cypher.planner: Set this to specify the default planner.
- dbms.pagecache.memory: The amount of memory to use for mapping the store files, in bytes (or kilobytes with the *k* suffix, megabytes with *m* and gigabytes with *g*).
- dbms.querylog.enabled: Log executed queries that takes longer than the configured threshold.
- dbms.querylog.filename: The file where queries will be recorded.
- dbms.querylog.threshold: If the execution of query takes more time than this threshold, the query is logged - provided query logging is enabled.
- dense_node_threshold: Relationship count threshold for considering a node to be dense.
- dump_configuration: Print out the effective Neo4j configuration after startup.
- index_background_sampling_enabled: Enable or disable background index sampling.
- index_sampling_buffer_size: Size of buffer used by index sampling.
- index_sampling_update_percentage: Percentage of index updates of total index size required before sampling of a given index is triggered.
- keep_logical_logs: Make Neo4j keep the logical transaction logs for being able to backup the database.
- logical_log_rotation_threshold: Specifies at which file size the logical log will auto-rotate.
- lucene_searcher_cache_size: The maximum number of open Lucene index searchers.
- node_auto_indexing: Controls the auto indexing feature for nodes.
- node_keys_indexable: A list of property names (comma separated) that will be indexed by default.
- query_cache_size: The number of Cypher query execution plans that are cached.
- read_only: Only allow read operations from this Neo4j instance.
- relationship_auto_indexing: Controls the auto indexing feature for relationships.
- relationship_grab_size: How many relationships to read at a time during iteration.

- relationship_keys_indexable: A list of property names (comma separated) that will be indexed by default.
- remote_logging_enabled: Whether to enable logging to a remote server or not.
- remote_logging_host: Host for remote logging using Logback SocketAppender.
- remote_logging_port: Port for remote logging using Logback SocketAppender.
- store_dir: The directory where the database files are located.

Deprecated settings

- log_mapped_memory_stats: Log memory mapping statistics regularly.
- log_mapped_memory_stats_filename: The file where memory mapping statistics will be recorded.
- log_mapped_memory_stats_interval: The number of records to be loaded between regular logging of memory mapping statistics.
- neostore.nodestore.db.mapped_memory: The size to allocate for memory mapping the node store.
- neostore.propertystore.db.arrays.mapped_memory: The size to allocate for memory mapping the array property store.
- neostore.propertystore.db.index.keys.mapped_memory: The size to allocate for memory mapping the store for property key strings.
- neostore.propertystore.db.index.mapped_memory: The size to allocate for memory mapping the store for property key indexes.
- neostore.propertystore.db.mapped_memory: The size to allocate for memory mapping the property value store.
- neostore.propertystore.db.strings.mapped_memory: The size to allocate for memory mapping the string property store.
- neostore.relationshipstore.db.mapped_memory: The size to allocate for memory mapping the relationship store.
- use_memory_mapped_buffers: Use memory mapped buffers for accessing the native storage layer.

allow_file_urls

Description	Determines if Cypher will allow using file URLs when loading data using `LOAD CSV`. Setting this value to `false` will cause Neo4j to fail `LOAD CSV` clauses that load data from the file system.
Valid values	is a boolean.
Default value	`true`

allow_store_upgrade

Description	Whether to allow a store upgrade in case the current version of the database starts against an older store version. Setting this to `true` does not guarantee successful upgrade, it just allows an upgrade to be performed.
Valid values	is a boolean.
Default value	`false`

batched_writes

Description	Whether or not transactions are appended to the log in batches.
Valid values	is a boolean.
Default value	`true`

cache_type

Description	The type of cache to use for nodes and relationships. Note that the Neo4j Enterprise Edition has the additional `hpc` cache type (High-Performance Cache). See the chapter on caches in the manual for more information.

Valid values	is one of soft, none, weak, strong, experimental-off.
Default value	soft

cypher_parser_version

Description	Set this to specify the default parser.
Valid values	is one of +<<config_1.9,1.9>>+, +<<config_2.0,2.0>>+, +<<config_2.1,2.1>>+, +<<config_2.2,2.2>>+.

dbms.cypher.min_replan_interval

Description	The minimum lifetime of a query plan before a query is considered for replanning.
Valid values	is a duration (valid units are ms, s, m).
Default value	1s

dbms.cypher.planner

Description	Set this to specify the default planner.
Valid values	is one of COST, RULE.

dbms.pagecache.memory

Description	The amount of memory to use for mapping the store files, in bytes (or kilobytes with the *k* suffix, megabytes with *m* and gigabytes with *g*). If Neo4j is running on a dedicated server, then it is generally recommended to leave about 2-4 gigabytes for the operating system, give the JVM enough heap to hold all your transaction state and query context, and then leave the rest for the page cache. The default page cache memory assumes the machine is dedicated to running Neo4j, and is heuristically set to 75% of RAM minus the max Java heap size.
Valid values	is a byte size (valid multipliers are k, m, g, K, M, G).
Default value	2303044608

dbms.querylog.enabled

Description	Log executed queries that takes longer than the configured threshold.
Valid values	is a boolean.
Default value	false

dbms.querylog.filename

Description	The file where queries will be recorded.
Valid values	is a path which is relative to .
Default value	queries.log

dbms.querylog.threshold

Description	If the execution of query takes more time than this threshold, the query is logged - provided query logging is enabled. Defaults to 0 seconds, that is all queries are logged.
Valid values	is a duration (valid units are ms, s, m).
Default value	0s

dense_node_threshold

Description	Relationship count threshold for considering a node to be dense.
Valid values	is an integer which is minimum 1.
Default value	50

dump_configuration

Description	Print out the effective Neo4j configuration after startup.
Valid values	is a boolean.
Default value	`false`

index_background_sampling_enabled

Description	Enable or disable background index sampling.
Valid values	is a boolean.
Default value	`true`

index_sampling_buffer_size

Description	Size of buffer used by index sampling.
Valid values	is a byte size (valid multipliers are `k`, `m`, `g`, `K`, `M`, `G`) which is minimum `1048576`, and is maximum `2147483647`.
Default value	`64m`

index_sampling_update_percentage

Description	Percentage of index updates of total index size required before sampling of a given index is triggered.
Valid values	is an integer which is minimum `0`.
Default value	`5`

keep_logical_logs

Description	Make Neo4j keep the logical transaction logs for being able to backup the database. Can be used for specifying the threshold to prune logical logs after. For example "10 days" will prune logical logs that only contains transactions older than 10 days from the current time, or "100k txs" will keep the 100k latest transactions and prune any older transactions.
Valid values	is a string which must be `true`/`false` or of format *<number><optional unit> <type>* for example `100M size` for limiting logical log space on disk to 100Mb, or `200k txs` for limiting the number of transactions to keep to 200 000.
Default value	`7 days`

log_mapped_memory_stats

Description	Log memory mapping statistics regularly.
Valid values	is a boolean.
Default value	`false`
Deprecated	This is no longer used.

log_mapped_memory_stats_filename

Description	The file where memory mapping statistics will be recorded.
Valid values	is a path which is relative to .
Default value	`mapped_memory_stats.log`
Deprecated	This is no longer used.

log_mapped_memory_stats_interval

Description	The number of records to be loaded between regular logging of memory mapping statistics.

Valid values	is an integer.
Default value	`1000000`
Deprecated	This is no longer used.

logical_log_rotation_threshold

Description	Specifies at which file size the logical log will auto-rotate. `0` means that no rotation will automatically occur based on file size.
Valid values	is a byte size (valid multipliers are `k`, `m`, `g`, `K`, `M`, `G`) which is minimum `1048576`.
Default value	`250M`

lucene_searcher_cache_size

Description	The maximum number of open Lucene index searchers.
Valid values	is an integer which is minimum `1`.
Default value	`2147483647`

neostore.nodestore.db.mapped_memory

Description	The size to allocate for memory mapping the node store.
Valid values	is a byte size (valid multipliers are `k`, `m`, `g`, `K`, `M`, `G`).
Deprecated	Replaced by the setting.

neostore.propertystore.db.arrays.mapped_memory

Description	The size to allocate for memory mapping the array property store.
Valid values	is a byte size (valid multipliers are `k`, `m`, `g`, `K`, `M`, `G`).
Deprecated	Replaced by the setting.

neostore.propertystore.db.index.keys.mapped_memory

Description	The size to allocate for memory mapping the store for property key strings.
Valid values	is a byte size (valid multipliers are `k`, `m`, `g`, `K`, `M`, `G`).
Deprecated	Replaced by the setting.

neostore.propertystore.db.index.mapped_memory

Description	The size to allocate for memory mapping the store for property key indexes.
Valid values	is a byte size (valid multipliers are `k`, `m`, `g`, `K`, `M`, `G`).
Deprecated	Replaced by the setting.

neostore.propertystore.db.mapped_memory

Description	The size to allocate for memory mapping the property value store.
Valid values	is a byte size (valid multipliers are `k`, `m`, `g`, `K`, `M`, `G`).
Deprecated	Replaced by the setting.

neostore.propertystore.db.strings.mapped_memory

Description	The size to allocate for memory mapping the string property store.
Valid values	is a byte size (valid multipliers are `k`, `m`, `g`, `K`, `M`, `G`).
Deprecated	Replaced by the setting.

neostore.relationshipstore.db.mapped_memory

Description	The size to allocate for memory mapping the relationship store.

Valid values	is a byte size (valid multipliers are `k`, `m`, `g`, `K`, `M`, `G`).
Deprecated	Replaced by the setting.

node_auto_indexing

Description	Controls the auto indexing feature for nodes. Setting it to `false` shuts it down, while `true` enables it by default for properties listed in the setting.
Valid values	is a boolean.
Default value	`false`

node_keys_indexable

Description	A list of property names (comma separated) that will be indexed by default. This applies to *nodes* only.
Valid values	is a string which must be a comma-separated list of keys to be indexed.

query_cache_size

Description	The number of Cypher query execution plans that are cached.
Valid values	is an integer which is minimum `0`.
Default value	`1000`

read_only

Description	Only allow read operations from this Neo4j instance. This mode still requires write access to the directory for lock purposes.
Valid values	is a boolean.
Default value	`false`

relationship_auto_indexing

Description	Controls the auto indexing feature for relationships. Setting it to `false` shuts it down, while `true` enables it by default for properties listed in the setting.
Valid values	is a boolean.
Default value	`false`

relationship_grab_size

Description	How many relationships to read at a time during iteration.
Valid values	is an integer which is minimum `1`.
Default value	`100`

relationship_keys_indexable

Description	A list of property names (comma separated) that will be indexed by default. This applies to *relationships* only.
Valid values	is a string which must be a comma-separated list of keys to be indexed.

remote_logging_enabled

Description	Whether to enable logging to a remote server or not.
Valid values	is a boolean.
Default value	`false`

remote_logging_host

Description	Host for remote logging using Logback SocketAppender.

Valid values	is a string which must be a valid hostname.
Default value	`127.0.0.1`

remote_logging_port

Description	Port for remote logging using Logback SocketAppender.
Valid values	is an integer which must be a valid port number (is in the range `0` to `65535`).
Default value	`4560`

store_dir

Description	The directory where the database files are located.
Valid values	is a path.

use_memory_mapped_buffers

Description	Use memory mapped buffers for accessing the native storage layer.
Valid values	is a boolean.
Default value	`true`
Deprecated	This setting has been obsoleted. Neo4j no longer relies on the memory-mapping capabilities of the operating system.

24.6. Caches in Neo4j

For how to provide custom configuration to Neo4j, see Section 24.1, "Introduction" [436].

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 fast graph traversal.

File buffer cache

Quick info

- The file buffer cache is sometimes called *low level cache*, *file system cache* or *the page cache*.
- It caches the Neo4j data as stored on the durable media.
- The default configuration of the cache relies on heuristics and assumes that the machine is dedicated to running Neo4j, so you might want to tune it yourself to get the most out of the available memory.
- There is just one setting for the file buffer cache: `dbms.pagecache.memory` - it specifies how much memory Neo4j is allowed to use for this cache.

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 writes. This behavior is safe since all transactions are always durably written to the transaction log, which can be used to recover the store files in the event of a crash. It also improves write performance by batching up many small writes into fewer page-sized writes.

Since the file buffer cache is caching the contents of the store files, you can calculate the appropriate size for it by summing up the space usage of all the store files. For instance, on a posix system you can look at the total of running `$ du -hc *store.db*` in your `data/graph.db` directory. If you configure the file buffer cache to have less memory than the size of the store, then the cache will automatically swap pages in and out on demand, such that the most popular data stays in memory.

Configuration

Parameter	Possible values	Effect
`dbms.pagecache.memory`	The maximum amount of memory to use for the file buffer cache, either in bytes, or greater byte-like units, such as `100m` for 100 mega-bytes, or `4g` for 4 giga-bytes.	The amount of memory to use for mapping the store files, in a unit of bytes. This will automatically be rounded down to the nearest whole page. This value cannot be zero. For extremely small and memory constrained deployments, it is recommended to still reserve at least a couple of megabytes for file buffering.
`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 configuring the amount of memory allowed for the file buffers and the JVM heap, make sure to also leave room for the operating systems page cache, and other programs and services the system might want to run. It is important to configure the memory usage, such that the Neo4j JVM process won't need to use any swap memory, as this will cause a significant drag on the performance of the system.

When reading the configuration parameters on startup Neo4j will automatically configure the parameters that are not specified. The cache size will be configured based on the available memory on the computer, with the assumption that the machine is dedicated to running Neo4j. Specifically, Neo4j will look at how much memory the machine has, subtract the JVM heap allocation from that, and then use 75% of what is left for the file buffer cache. This is the default configuration when nothing else is specified.

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. There are two different categories of object caches in Neo4j.

Firstly, there are the *reference caches*. With these caches, Neo4j will utilize as much of the allocated JVM heap memory as it can to hold nodes and relationships. It relies on garbage collection for eviction from the cache in an LRU manner. Note however that Neo4j is "competing" for the heap space with other objects in the same JVM, such as a your application (if deployed in embedded mode) or intermediate objects produced by Cypher queries, and Neo4j will yield to the application or query by using less memory for caching.

Note
The High-Performance Cache described below is only available in the Neo4j Enterprise Edition.

The other is the *High-Performance Cache* which gets assigned a certain maximum amount of space on the JVM heap and will purge objects whenever it grows bigger than that. Objects are evicted from the high performance cache when the maximum size is about to be reached, instead of relying on garbage collection (GC) to make that decision. With the high-performance cache, GC-pauses can be better controlled. The overhead of the High-Performance Cache is also much smaller as well as insert/lookup times faster than for reference caches.

Tip
The use of heap memory is subject to the Java Garbage Collector — depending on the cache type some tuning might be needed to play well with the GC at large heap sizes. Therefore, assigning a large heap for Neo4j's sake isn't always the best strategy as it may lead to long GC pauses. Instead leave some space for Neo4j's file buffer cache. The file buffer cache is allocated outside of the Java heap, and thus does not impact the performance of the garbage collector.

The content of this cache are objects with a representation geared towards supporting the Neo4j object API and graph traversals. Reading from this cache may be faster than reading from the file buffer cache, depending on the workload. 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.

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. Note that there will exist two cache instances, one for nodes and one for relationships. The available cache types are:

`cache_type`	**Description**
`none`	Do not use a high level cache. No objects will be cached. This cache is useful for applications that otherwise suffer from high GC pause times.
`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 type in Neo4j Community Edition.
`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 hold on to **all data** that gets loaded to never release it again. Provides good performance if your graph is small enough to fit in memory.
`hpc`	The High-Performance Cache. Provides means of assigning a specific amount of memory to dedicate to caching loaded nodes and relationships. Small footprint and fast insert/lookup. Should be the best option for most scenarios. See below on how to configure it. This cache type is only available in the Neo4j Enterprise Edition, and is the default in that edition.

High-Performance Cache

How much memory to allocate to the High Performance Cache can be fine tuned to suit your use case. There are two ways to configure memory usage for it.

Standard configuration

For most use cases, simply specifying a percentage of the memory available for caching to use is enough to tune the High-Performance Cache.

Allocating more memory to the cache gives faster querying speed, but takes memory away from other components and may put strain on the JVM garbage collector.

The max amount of memory available for caching depends on which garbage collector you are using. For CMS and G1 collectors (CMS is the Neo4j default), it will be equal to the max size of the old generation. How big the old generation is is platform-dependent, but can for CMS and G1 be configured using the "NewRatio" JVM configuration option. For other collectors, it will be half of the total heap size.

`configuration option`	**Description (what it controls)**	**Example value**
`cache.memory_ratio`	Percentage, 0-100, of memory available for caching to use for caching. Default is 50%.	`50.0`

Advanced configuration

The advanced configuration gives more fine-grained control of how much memory to allocate specific parts of the cache. There are two aspects to this configuration.

- The size of the array referencing the objects that are put in the cache.

- The maximum size of all the objects in the cache.

See below for further details.

High Performance Cache configuration settings

- cache.memory_ratio: Set how much of the memory available for caching to use for caching.
- node_cache_array_fraction: Fraction of the heap to dedicate to the array holding the nodes in the cache.
- node_cache_size: Maximum size of the heap memory to dedicate to the cached nodes.
- relationship_cache_array_fraction: Fraction of the heap to dedicate to the array holding the relationships in the cache.
- relationship_cache_size: Maximum size of the heap memory to dedicate to the cached relationships.

cache.memory_ratio

Description	Set how much of the memory available for caching to use for caching. It is recommended to not have this value exceed 70 percent.
Valid values	is a value between 0 and 100 which cannot be used in conjunction with other object cache size settings, and may not be set to more than 80% of the available heap space.

node_cache_array_fraction

Description	Fraction of the heap to dedicate to the array holding the nodes in the cache. Specifying `5` will let that array itself take up 5% out of the entire heap. Increasing this figure will reduce the chance of hash collisions at the expense of more heap used for it. More collisions means more redundant loading of objects from the low level cache.
Valid values	is a float which is in the range +<<config_1.0,1.0>>+ to +<<config_10.0,10.0>>+.
Default value	`1.0`

node_cache_size

Description	Maximum size of the heap memory to dedicate to the cached nodes. Right before the maximum size is reached a purge is performed. The purge will evict objects from the cache until the cache size gets below 90% of the maximum size. Optimal settings for the maximum size depends on the size of your graph. The configured maximum size should leave enough room for other objects to coexist in the same JVM. At the same time it should be large enough to keep loading from the low level cache at a minimum. Predicted load on the JVM as well as layout of domain level objects should also be taken into consideration.
Valid values	is a byte size (valid multipliers are `k`, `m`, `g`, `K`, `M`, `G`).

relationship_cache_array_fraction

Description	Fraction of the heap to dedicate to the array holding the relationships in the cache. See for more information.
Valid values	is a float which is in the range +<<config_1.0,1.0>>+ to +<<config_10.0,10.0>>+.
Default value	`1.0`

relationship_cache_size

Description	Maximum size of the heap memory to dedicate to the cached relationships. See for more information.
Valid values	is a byte size (valid multipliers are `k`, `m`, `g`, `K`, `M`, `G`).

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

- Understanding soft/weak references[4]
- How Hotspot Decides to Clear SoftReferences[5]
- HotSpot FAQ[6]

[4] http://weblogs.java.net/blog/enicholas/archive/2006/05/understanding_w.html
[5] http://jeremymanson.blogspot.com/2009/07/how-hotspot-decides-to-clear_07.html
[6] http://www.oracle.com/technetwork/java/hotspotfaq-138619.html#gc_softrefs

24.7. Logical logs

Logical logs in Neo4j are the journal of which operations happens and are the source of truth in scenarios where the database needs to be recovered after a crash or similar. Logs are rotated every now and then (defaults to when they surpass 25 Mb in size) and the amount of legacy logs to keep can be configured. Purpose of keeping a history of logical logs include being able to serve incremental backups as well as keeping an HA cluster running.

For any given configuration at least the latest non-empty logical log will be kept, but configuration can be supplied to control how much more to keep. There are several different means of controlling it and the format in which configuration is supplied is:

```
keep_logical_logs=<true/false>
keep_logical_logs=<amount> <type>
```

For example:

```
# Will keep logical logs indefinitely
keep_logical_logs=true

# Will keep only the most recent non-empty log
keep_logical_logs=false

# Will keep logical logs which contains any transaction committed within 30 days
keep_logical_logs=30 days

# Will keep logical logs which contains any of the most recent 500 000 transactions
keep_logical_logs=500k txs
```

Full list:

Type	Description	Example
files	Number of most recent logical log files to keep	"10 files"
size	Max disk size to allow log files to occupy	"300M size" or "1G size"
txs	Number of latest transactions to keep Keep	"250k txs" or "5M txs"
hours	Keep logs which contains any transaction committed within N hours from current time	"10 hours"
days	Keep logs which contains any transaction committed within N days from current time	"50 days"

24.8. JVM Settings

Background

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[7], 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.

Configuring heap size and GC

Note
When using Neo4j Server, JVM configuration goes into the `conf/neo4j-wrapper.conf` file, see Section 24.2, “Server Configuration” [437].

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.

[7] http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access

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.

Guidelines for heap size

Number of entities	RAM size	Heap configuration	Reserved RAM for the OS
10M	2GB	512MB	~1GB
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 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

24.9. 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 and less disk space required.

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.
- Hex (lower case), consisting of digits 0..9 and lower case letters a..f
- Hex (upper case), consisting of digits 0..9 and upper case letters a..f
- Upper case, consisting of upper case letters A..Z, and the punctuation space, underscore, period, dash, colon and slash.
- Lower case, like upper but with lower case letters a..z instead of upper case
- E-mail, consisting of lower case letters a..z and the punctuation comma, underscore, period, dash, plus and the at sign (@).
- URI, consisting of lower case letters a..z, digits 0..9 and most punctuation available.
- Alpha-numerical, consisting of both upper and lower case letters a..zA..z, digits 0..9 and punctuation space and underscore.
- Alpha-symbolical, consisting of both upper and lower case 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

Character count limits

String class	Character count limit
Numerical, Date and Hex	54
Uppercase, Lowercase and E-mail	43
URI, Alphanumerical and Alphasymbolical	36
European	31
Latin1	27
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.

24.10. 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.

24.11. Memory mapped IO settings

Introduction

Each file in the Neo4j store is accessed through the Neo4j page cache, when reading from, or writing to, the store files. Since there is only one page cache, there is only one setting for specifying how much memory Neo4j is allowed to use for page caching. The shared page cache ensures that memory is split across the various store files in the most optimal manner [8], depending on how the database is used and what data is popular.

The memory for the page cache is allocated outside the normal Java heap, so you need to take both the Java heap, and the page cache, into consideration in your capacity planning. 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 the page cache, 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.

Important
Make sure that your system is configured such that it will never need to swap. If memory belonging to the Neo4j process gets swapped out, it can lead to considerable performance degradation.

The amount of memory available to the page cache is configured using the `dbms.pagecache.memory` setting. With that setting, you specify the number of bytes available to the page cache, e.g. `150m` og `4g`. The default page memory setting is 75% of the machines memory, after subtracting the memory that is reserved for the Java heap.

For optimal performance, you will want to have as much of your data fit in the page cache as possible. You can sum up the size of all the `*store.db*` files in your store file directory, to figure out how big a page cache you need to fit all your data. Obviously the store files will grow as you add more nodes, relationships and properties, so configuring more page cache memory than you have data, is recommended when possible.

Batch insert example

Read general information on batch insertion in Chapter 38, *Batch Insertion* [650].

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. The important thing is that the page cache has enough memory to work with, that it doesn't slow down the BatchInserter:

```
dbms.pagecache.memory=4g
```

The configuration above will more or less fit the entire graph in memory. A rough formula to calculate the memory needed can look like this:

```
bytes_needed = number_of_nodes * 15
             + number_of_relationships * 34
             + number_of_properties * 64
```

Note that the size of the individual property very much depends on what data it contains. The numbers given in the above formula are only a rough estimate.

[8] This is an informal comparison to the store-specific memory mapping settings of previous versions of Neo4j. We are not claiming that our page replacement algorithms are optimal in the formal sense. Truly optimal page replacement algorithms require knowledge of events arbitrarily far into the future.

24.12. Linux Performance Guide

Introduction

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 24, *Configuration & Performance* [435], 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.

File system benchmark

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-contrib/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 | --noread | --nomemorymap]>>
```

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.

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).

> **Note**
> The file named `store` should be created and cached first (see the section called "Setup" [464]).

```
$ ./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.

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.

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.

Setting the number of open files

Linux platforms impose an upper limit on the number of concurrent files a user may have open. This number is reported for the current user and session with the command

```
user@localhost:~$ ulimit -n
1024
```

The usual default of 1024 is often not enough, especially when many indexes are used or a server installation sees too many connections (network sockets count against that limit as well). Users are therefore encouraged to increase that limit to a healthy value of 40000 or more, depending on usage patterns. Setting this value via the `ulimit` command is possible only for the root user and that for that session only. To set the value system wide you have to follow the instructions for your platform.

What follows is the procedure to set the open file descriptor limit to 40k for user neo4j under Ubuntu 10.04 and later. If you opted to run the neo4j service as a different user, change the first field in step 2 accordingly.

1. Become root since all operations that follow require editing protected system files.

   ```
   user@localhost:~$ sudo su -
   Password:
   root@localhost:~$
   ```

2. Edit `/etc/security/limits.conf` and add these two lines:

   ```
   neo4j   soft    nofile  40000
   neo4j   hard    nofile  40000
   ```

3. Edit `/etc/pam.d/su` and uncomment or add the following line:

   ```
   session    required   pam_limits.so
   ```

4. A restart is required for the settings to take effect.
   After the above procedure, the neo4j user will have a limit of 40000 simultaneous open files. If you continue experiencing exceptions on `Too many open files` or `Could not stat() directory` then you may have to raise that limit further.

Chapter 25. 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:

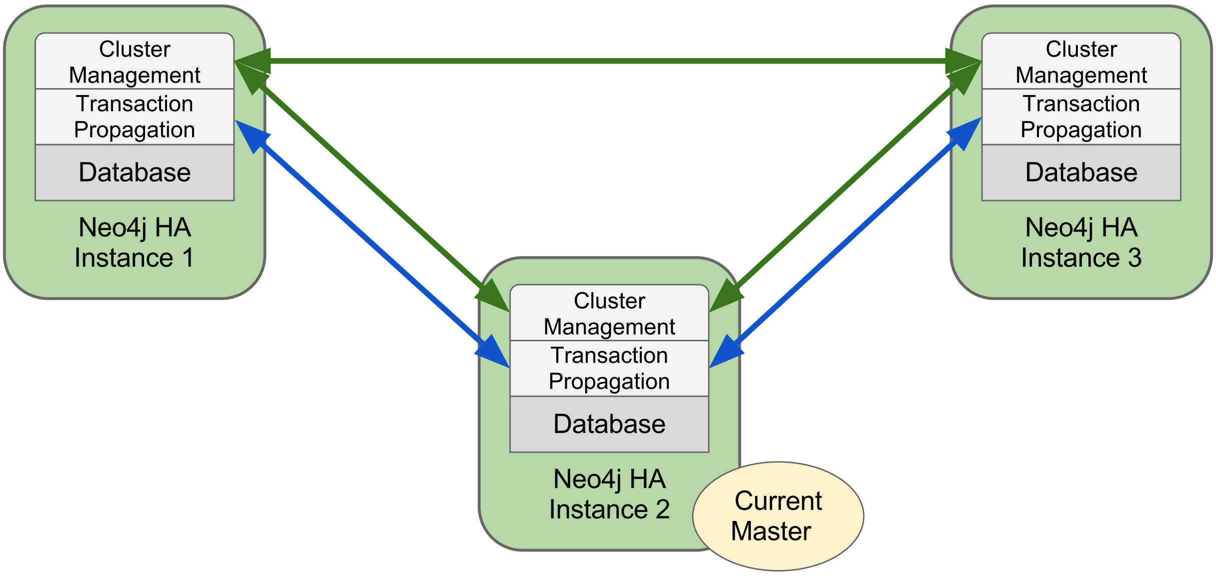
1. It enables a *fault-tolerant database architecture*, where several Neo4j slave databases can be configured to be exact replicas of a single Neo4j master database. This allows the end-user system to be fully functional and both read and write to the database in the event of hardware failure.
2. It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

25.1. 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 `GraphDatabaseFactory` to `HighlyAvailableGraphDatabaseFactory`. Since both implement the same interface, no additional changes are required.

Figure 25.1. 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 write requests on all machines so there is no need to redirect those to the master specifically.

A slave will handle writes by synchronizing with the master to preserve consistency. Writes to master can be configured to be optimistically pushed to 0 or more slaves. By optimistically we mean the master will try to push to slaves before the transaction completes but if it fails the transaction will still be successful (different from normal replication factor). All updates will however propagate from the master to other slaves eventually so a write from one slave may not be 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.

25.2. Setup and configuration

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

In HA mode, Neo4j instances form a cluster. The instances monitor each others' availability to take account of instances joining and leaving the cluster. They elect one instance to be the master, and designate the other instances to be slaves.

For installation instructions of a High Availability cluster see Section 25.6, "High Availability setup tutorial" [479].

Specifying cluster members

Specify the instances that should form the cluster by supplying `ha.initial_hosts`, a comma-separated list of URLs. When each instance starts, if it can contact any of the initial hosts, then it will form a cluster with them, otherwise it will start its own cluster.

Note that the parameter is called `ha.initial_hosts` because it's only used when instances initially join the cluster. This means that you can extend the cluster without changing the configuration of existing instances.

Server configuration

If you are running Neo4j server, specify `org.neo4j.server.database.mode=HA` in *conf/neo4j-server.properties*.

HA server configuration parameters

Parameter Name	Description	Example value	Required?
`org.neo4j.server.database.mode`	Whether to run as a single server or in HA mode.	`single` or `HA`	yes

Database configuration

HA configuration parameters should be supplied alongside general Neo4j parameters in *conf/neo4j.properties*. There are many configurable parameters. In most cases it isn't necessary to modify the default values. The only parameters that need to be specified are `ha.server_id` and `ha.initial_hosts`.

High Availability configuration settings

- ha.branched_data_policy: Policy for how to handle branched data.
- ha.com_chunk_size: Max size of the data chunks that flows between master and slaves in HA.
- ha.lock_read_timeout: Timeout for taking remote (write) locks on slaves.
- ha.max_concurrent_channels_per_slave: Maximum number of connections a slave can have to the master.
- ha.pull_interval: Interval of pulling updates from master.
- ha.read_timeout: How long a slave will wait for response from master before giving up.
- ha.server: Hostname and port to bind the HA server.
- ha.slave_only: Whether this instance should only participate as slave in cluster.
- ha.state_switch_timeout: Timeout for waiting for instance to become master or slave.
- ha.tx_push_factor: The amount of slaves the master will ask to replicate a committed transaction.
- ha.tx_push_strategy: Push strategy of a transaction to a slave during commit.

ha.branched_data_policy

Description	Policy for how to handle branched data.
Valid values	is one of `+<<config_keep_all,keep_all>>+`, `+<<config_keep_last,keep_last>>+`, `+<<config_keep_none,keep_none>>+`.

Default value	`keep_all`

ha.com_chunk_size

Description	Max size of the data chunks that flows between master and slaves in HA. Bigger size may increase throughput, but may also be more sensitive to variations in bandwidth, whereas lower size increases tolerance for bandwidth variations.
Valid values	is a byte size (valid multipliers are `k`, `m`, `g`, `K`, `M`, `G`) which is minimum `1024`.
Default value	`2M`

ha.lock_read_timeout

Description	Timeout for taking remote (write) locks on slaves. Defaults to .
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`20s`

ha.max_concurrent_channels_per_slave

Description	Maximum number of connections a slave can have to the master.
Valid values	is an integer which is minimum `1`.
Default value	`20`

ha.pull_interval

Description	Interval of pulling updates from master.
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`0s`

ha.read_timeout

Description	How long a slave will wait for response from master before giving up.
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`20s`

ha.server

Description	Hostname and port to bind the HA server.
Valid values	is a hostname and port.
Default value	`0.0.0.0:6001-6011`

ha.slave_only

Description	Whether this instance should only participate as slave in cluster. If set to `true`, it will never be elected as master.
Valid values	is a boolean.
Default value	`false`

ha.state_switch_timeout

Description	Timeout for waiting for instance to become master or slave.
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`120s`

ha.tx_push_factor

Description	The amount of slaves the master will ask to replicate a committed transaction.

Valid values	is an integer which is minimum `0`.
Default value	`1`

ha.tx_push_strategy

Description	Push strategy of a transaction to a slave during commit.
Valid values	is one of `+<<config_round_robin, round_robin>>+`, `fixed`.
Default value	`fixed`

Cluster configuration settings

- ha.allow_init_cluster: Whether to allow this instance to create a cluster if unable to join.
- ha.broadcast_timeout: Timeout for broadcasting values in cluster.
- ha.cluster_name: The name of a cluster.
- ha.cluster_server: Host and port to bind the cluster management communication.
- ha.configuration_timeout: Timeout for waiting for configuration from an existing cluster member during cluster join.
- ha.default_timeout: Default timeout used for clustering timeouts.
- ha.election_timeout: Timeout for waiting for other members to finish a role election.
- ha.heartbeat_interval: How often heartbeat messages should be sent.
- ha.heartbeat_timeout: Timeout for heartbeats between cluster members.
- ha.initial_hosts: A comma-separated list of other members of the cluster to join.
- ha.join_timeout: Timeout for joining a cluster.
- ha.learn_timeout: Timeout for learning values.
- ha.leave_timeout: Timeout for waiting for cluster leave to finish.
- ha.paxos_timeout: Default timeout for all Paxos timeouts.
- ha.phase1_timeout: Timeout for Paxos phase 1.
- ha.phase2_timeout: Timeout for Paxos phase 2.
- ha.server_id: Id for a cluster instance.

ha.allow_init_cluster

Description	Whether to allow this instance to create a cluster if unable to join.
Valid values	is a boolean.
Default value	`true`

ha.broadcast_timeout

Description	Timeout for broadcasting values in cluster. Must consider end-to-end duration of Paxos algorithm. This value is the default value for the and settings.
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`30s`

ha.cluster_name

Description	The name of a cluster.
Valid values	is a string which must be a valid cluster name.
Default value	`neo4j.ha`

ha.cluster_server

Description	Host and port to bind the cluster management communication.
Valid values	is a hostname and port.

Default value	`0.0.0.0:5001-5099`

ha.configuration_timeout

Description	Timeout for waiting for configuration from an existing cluster member during cluster join.
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`1s`

ha.default_timeout

Description	Default timeout used for clustering timeouts. Override specific timeout settings with proper values if necessary. This value is the default value for the , and settings.
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`5s`

ha.election_timeout

Description	Timeout for waiting for other members to finish a role election. Defaults to .
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`5s`

ha.heartbeat_interval

Description	How often heartbeat messages should be sent. Defaults to .
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`5s`

ha.heartbeat_timeout

Description	Timeout for heartbeats between cluster members. Should be at least twice that of .
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`11s`

ha.initial_hosts

Description	A comma-separated list of other members of the cluster to join.
Valid values	is a list separated by "," where items are a hostname and port.
Mandatory	The `ha.initial_hosts` configuration setting is mandatory.

ha.join_timeout

Description	Timeout for joining a cluster. Defaults to .
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`30s`

ha.learn_timeout

Description	Timeout for learning values. Defaults to .
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`5s`

ha.leave_timeout

Description	Timeout for waiting for cluster leave to finish. Defaults to .

Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`30s`

ha.paxos_timeout

Description	Default timeout for all Paxos timeouts. Defaults to . This value is the default value for the , and settings.
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`5s`

ha.phase1_timeout

Description	Timeout for Paxos phase 1. Defaults to .
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`5s`

ha.phase2_timeout

Description	Timeout for Paxos phase 2. Defaults to .
Valid values	is a duration (valid units are `ms`, `s`, `m`).
Default value	`5s`

ha.server_id

Description	Id for a cluster instance. Must be unique within the cluster.
Valid values	is an instance id, which has to be a valid integer.
Mandatory	The `ha.server_id` configuration setting is mandatory.

25.3. How Neo4j HA operates

A Neo4j HA cluster operates cooperatively — each database instance contains the logic needed in order to coordinate with the other members of the cluster. On startup a Neo4j HA database instance will try to connect to an existing cluster specified by configuration. If the cluster exists, the instance will join it as a slave. Otherwise the cluster will be created and the instance will become its 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 be committed on the master and then, if successful, on the slave. 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 be applied to a slave communicating with its master automatically.

Write transactions performed directly through the master will execute in the same way as running in normal non-HA mode. On success the transaction will be pushed out to a configurable number of slaves (default one slave). This is done optimistically meaning if the push fails the transaction will still be successful. It's also possible to configure push factor to 0 for higher write performance when writing directly through the master, although increasing the risk of losing any transaction not yet pulled by another slave if the master goes down.

Slaves can also be configured to pull updates asynchronously by setting the `ha.pull_interval` option.

Whenever a Neo4j database becomes unavailable, by means of for example hardware failure or network outages, the other database instances in the cluster will detect that and mark it as temporarily failed. A database instance that becomes available after being unavailable will automatically catch up with the cluster. If the master goes down another (best suited) member will be elected and have its role switched from slave to master after a quorum has been reached within the cluster. When the new master has performed its role switch it will broadcast its availability to all the other members of the cluster. Normally a new master is elected and started within just a few seconds and during this time no writes can take place (the writes will block or in rare cases throw an exception). The only time this is not true is when an old master had changes that did not get replicated to any other member before becoming unavailable. If the new master is elected and performs changes before the old master recovers, there will be two "branches" of the database after the point where the old master became unavailable. The old master will move away its database (its "branch") and download a full copy from the new master, to become available as a slave in the cluster.

All this can be summarized as:

- Write transactions can be performed on any database instance in a cluster.
- Neo4j HA is fault tolerant and can continue to operate from any number of machines down to a single machine.
- Slaves will be automatically synchronized with the master on write operations.
- If the master fails a new master will be elected automatically.
- The cluster automatically handles instances becoming unavailable (for example due to network issues), and also makes sure to accept them as members in the cluster when they are available again.
- Transactions are atomic, consistent and durable but eventually propagated out to other slaves.
- Updates to slaves are eventually consistent by nature but can be configured to be pushed optimistically from master during commit.
- If the master goes down any running write transaction will be rolled back and new transactions will block or fail until a new master has become available.
- Reads are highly available and the ability to handle read load scales with more database instances in the cluster.

25.4. Arbiter Instances

A typical deployment of Neo4j will use a cluster of 3 machines to provide fault-tolerance and read scalability. This setup is described in Section 25.6, “High Availability setup tutorial” [479].

While having at least 3 instances is necessary for failover to happen in case the master becomes unavailable, it is not required for all instances to run the full Neo4j stack, which includes the database engine. Instead, what is called arbiter instances can be deployed. They can be regarded as cluster participants in that their role is to take part in master elections with the single purpose of breaking ties in the election process. That makes possible a scenario where you have a cluster of 2 Neo4j database instances and an additional arbiter instance and still enjoy tolerance of a single failure of either of the 3 instances.

Arbiter instances are configured in the same way as Neo4j HA members are — through the *conf/neo4j.properties* file. Settings that are not cluster specific are of course ignored, so you can easily start up an arbiter instance in place of a properly configured Neo4j instance.

To start an arbiter instance, call

```
neo4j_home$ ./bin/neo4j-arbiter start
```

You can also stop, install and remove it as a service and ask for it’s status in exactly the same way as for Neo4j instances. See also Section 23.2, “Server Installation” [426].

25.5. Upgrade of a Neo4j HA Cluster

Caution
Before attempting any of the upgrades described here, please backup your database store!

To upgrade a Neo4j HA cluster to Neo4j 2.2.1 requires following a specific process which ensures that the cluster remains consistent and all cluster instances are able to join and participate following their upgrade.

Neo4j 2.2.1 does not support rolling upgrades, only the standard upgrade procedure is available.

Important
After upgrade is complete, existing backups will not be suitable for updating via the incremental online backup. Please ensure that the first backup after upgrading uses an empty target directory, and thus performs a full backup.

Standard upgrade

In order to perform a cluster upgrade to Neo4j 2.2.1, you need to first upgrade the database store on a single instance, and then allow that store to propagate out to slaves.

Steps

The following process is recommended:

Backup

- Before anything else, backup your database store!

Shut down the cluster

- Shut down all instances. This is usually best done one instance after the other, rather than all at once. It is also strongly recommended to shut down the master of the cluster last.

Upgrade the database store on the previous master

1. Install Neo4j 2.2.1 on the previous master, keeping the database store (typically *data/graph.db*) from the previous version.
2. Disable HA in the configuration, by setting `org.neo4j.server.database.mode=SINGLE`.
3. Upgrade as described for a single instance of Neo4j (this may involve configuring with `allow_store_upgrade=true`, as described in "Explicit Upgrades").
4. Once upgraded, shut down Neo4j again.
5. Re-enable HA in the configuration, by setting `org.neo4j.server.database.mode=HA` again.

Upgrade the slaves

- Install Neo4j 2.2.1 on all previous slaves **and remove their database store** (typically *data/graph.db*). *Slaves should not be started with a previous store present.*

Restart the cluster

1. Start the previous master instance.
2. Start each slave, one after the other. Once each slave has joined the cluster, it will sync the store from the master instance.

Tip
For larger databases, it is possible to manually copy the database store from the previous master *after it has completed the upgrade* to the slaves before starting them. This will avoid the need for them to sync from the master when starting.

Downgrading

Downgrade is only supported between Neo4j versions that for automatic upgrades. This typically means only within patch releases of the same Neo4j version.

Downgrade follows the same process as for standard upgrade.

25.6. High Availability setup tutorial

This guide will help you understand how to configure and deploy a Neo4j High Availability cluster. Two scenarios will be considered:

- Configuring 3 instances to be deployed on 3 separate machines, in a setting similar to what might be encountered in a production environment.
- Modifying the former to make it possible to run a cluster of 3 instances on the same physical machine, which is particularly useful during development.

Background

Each instance in a Neo4j HA cluster must be assigned an integer ID, which serves as its unique identifier. At startup, a Neo4j instance contacts the other instances specified in the `ha.initial_hosts` configuration option.

When an instance establishes a connection to any other, it determines the current state of the cluster and ensures that it is eligible to join. To be eligible the Neo4j instance must host the same database store as other members of the cluster (although it is allowed to be in an older state), or be a new deployment without a database store.

Explicitly configure IP Addresses/Hostnames for a cluster
Neo4j will attempt to configure IP addresses for itself in the absence of explicit configuration. However in typical operational environments where machines have multiple network cards and support IPv4 and IPv6 it is **strongly** recommended that the operator explicitly sets the IP address/hostname configuration for each machine in the cluster.

Let's examine the available settings and the values they accept.

ha.server_id

`ha.server_id` is the cluster identifier for each instance. It must be a positive integer and must be unique among all Neo4j instances in the cluster.

For example, `ha.server_id=1`.

ha.cluster_server

`ha.cluster_server` is an address/port setting that specifies where the Neo4j instance will listen for cluster communications (like hearbeat messages). The default port is `5001`. In the absence of a specified IP address, Neo4j will attempt to find a valid interface for binding. While this behavior typically results in a well-behaved server, it is **strongly** recommended that users explicitly choose an IP address bound to the network interface of their choosing to ensure a coherent cluster deployment.

For example, `ha.cluster_server=192.168.33.22:5001` will listen for cluster communications on the network interface bound to the 192.168.33.0 subnet on port 5001.

ha.initial_hosts

`ha.initial_hosts` is a comma separated list of address/port pairs, which specify how to reach other Neo4j instances in the cluster (as configured via their `ha.cluster_server` option). These hostname/ports will be used when the Neo4j instances starts, to allow it up to find and join the cluster. Specifying an instance's own address is permitted.

Warning
Do **not** use any whitespace in this configuration option.

For example, `ha.initial_hosts=192.168.33.22:5001,192.168.33.21:5001` will attempt to reach Neo4j instances listening on 192.168.33.22 on port 5001 and 192.168.33.21 on port 5001 on the 192.168.33.0 subnet.

ha.server

`ha.server` is an address/port setting that specifies where the Neo4j instance will listen for transactions (changes to the graph data) from the cluster master. The default port is `6001`. In the absence of a specified IP address, Neo4j will attempt to find a valid interface for binding. While this behavior typically results in a well-behaved server, it is **strongly** recommended that users explicitly choose an IP address bound to the network interface of their choosing to ensure a coherent cluster topology.

`ha.server` must user a different port to `ha.cluster_server`.

For example, `ha.server=192.168.33.22:6001` will listen for cluster communications on the network interface bound to the 192.168.33.0 subnet on port 6001.

Address/port format

The `ha.cluster_server` and `ha.server` configuration options are specified as <IP address>:<port>.

For `ha.server` the IP address MUST be the address assigned to one of the host's network interfaces.

For `ha.cluster_server` the IP address MUST be the address assigned to one of the host's network interfaces, or the value `0.0.0.0`, which will cause Neo4j to listen on every network interface.

Either the address or the port can be omitted, in which case the default for that part will be used. If the address is omitted, then the port must be preceded with a colon (eg. `:5001`).

The syntax for setting the port range is: <hostname>:<first port>[-<second port>]. In this case, Neo4j will test each port in sequence, and select the first that is unused. Note that this usage is not permitted when the hostname is specified as `0.0.0.0` (the "all interfaces" address).

Getting started: Setting up a production cluster

Download and unpack Neo4j Enterprise

Download Neo4j Enterprise from the Neo4j download site[1], and unpack on 3 separate machines.

Configure HA related settings for each installation

The following settings should be configured for each Neo4j installation. Note that all 3 installations have the same configuration, except for the `ha.server_id` property.

Neo4j instance #1 — neo4j-01.local

conf/neo4j.properties:

```
# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 1

# List of other known instances in this cluster
ha.initial_hosts = neo4j-01.local:5001,neo4j-02.local:5001,neo4j-03.local:5001
# Alternatively, use IP addresses:
#ha.initial_hosts = 192.168.0.20:5001,192.168.0.21:5001,192.168.0.22:5001
```

conf/neo4j-server.properties

```
# Let the webserver only listen on the specified IP.
org.neo4j.server.webserver.address=0.0.0.0

# HA - High Availability
```

[1] http://neo4j.com/download/

```
# SINGLE - Single mode, default.
org.neo4j.server.database.mode=HA
```

Neo4j instance #2 — neo4j-02.local

conf/neo4j.properties:

```
# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 2

# List of other known instances in this cluster
ha.initial_hosts = neo4j-01.local:5001,neo4j-02.local:5001,neo4j-03.local:5001
# Alternatively, use IP addresses:
#ha.initial_hosts = 192.168.0.20:5001,192.168.0.21:5001,192.168.0.22:5001
```

conf/neo4j-server.properties

```
# Let the webserver only listen on the specified IP.
org.neo4j.server.webserver.address=0.0.0.0

# HA - High Availability
# SINGLE - Single mode, default.
org.neo4j.server.database.mode=HA
```

Neo4j instance #3 — neo4j-03.local

conf/neo4j.properties:

```
# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 3

# List of other known instances in this cluster
ha.initial_hosts = neo4j-01.local:5001,neo4j-02.local:5001,neo4j-03.local:5001
# Alternatively, use IP addresses:
#ha.initial_hosts = 192.168.0.20:5001,192.168.0.21:5001,192.168.0.22:5001
```

conf/neo4j-server.properties

```
# Let the webserver only listen on the specified IP.
org.neo4j.server.webserver.address=0.0.0.0

# HA - High Availability
# SINGLE - Single mode, default.
org.neo4j.server.database.mode=HA
```

Start the Neo4j Servers

Start the Neo4j servers as normal. Note the startup order does not matter.

```
neo4j-01$ ./bin/neo4j start
```

```
neo4j-02$ ./bin/neo4j start
```

```
neo4j-03$ ./bin/neo4j start
```

Startup Time

When running in HA mode, the startup script returns immediately instead of waiting for the server to become available. This is because the instance does not accept any requests until a cluster has been formed. In the example above this happens when you startup the second instance. To keep track of the startup state you can follow the messages in console.log - the path to that is printed before the startup script returns.

Now, you should be able to access the 3 servers and check their HA status:

http://neo4j-01.local:7474/webadmin/#/info/org.neo4j/High%20Availability/

http://neo4j-02.local:7474/webadmin/#/info/org.neo4j/High%20Availability/

http://neo4j-03.local:7474/webadmin/#/info/org.neo4j/High%20Availability/

Tip
You can replace database #3 with an *arbiter* instance, see Arbiter Instances.

That's it! You now have a Neo4j HA cluster of 3 instances running. You can start by making a change on any instance and those changes will be propagated between them. For more HA related configuration options take a look at HA Configuration.

Alternative setup: Creating a local cluster for testing

If you want to start a cluster similar to the one described above, but for development and testing purposes, it is convenient to run all Neo4j instances on the same machine. This is easy to achieve, although it requires some additional configuration as the defaults will conflict with each other. Furthermore, the default `dbms.pagecache.memory` assumes that Neo4j has the machine to itself. If we in this example assume that the machine has 4 gigabytes of memory, and that each JVM consumes 500 megabytes of memory, then we can allocate 500 megabytes of memory to the page cache of each server.

Download and unpack Neo4j Enterprise

Download Neo4j Enterprise from the Neo4j download site[2], and unpack into 3 separate directories on your test machine.

Configure HA related settings for each installation

The following settings should be configured for each Neo4j installation.

Neo4j instance #1 — ~/neo4j-01

conf/neo4j.properties:

```
# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 1

# IP and port for this instance to bind to for communicating data with the
# other neo4j instances in the cluster.
ha.server = 127.0.0.1:6363
online_backup_server = 127.0.0.1:6366

# IP and port for this instance to bind to for communicating cluster information
# with the other neo4j instances in the cluster.
ha.cluster_server = 127.0.0.1:5001

# List of other known instances in this cluster
ha.initial_hosts = 127.0.0.1:5001,127.0.0.1:5002,127.0.0.1:5003

# Reduce the default page cache memory allocation
dbms.pagecache.memory=500m
```

conf/neo4j-server.properties

```
# database location
```

[2] http://neo4j.com/download/

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

# http port (for all data, administrative, and UI access)
org.neo4j.server.webserver.port=7474

# https port (for all data, administrative, and UI access)
org.neo4j.server.webserver.https.port=7484

# HA - High Availability
# SINGLE - Single mode, default.
org.neo4j.server.database.mode=HA
```

Neo4j instance #2 — ~/neo4j-02

conf/neo4j.properties:

```
# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 2

# IP and port for this instance to bind to for communicating data with the
# other neo4j instances in the cluster.
ha.server = 127.0.0.1:6364
online_backup_server = 127.0.0.1:6367

# IP and port for this instance to bind to for communicating cluster information
# with the other neo4j instances in the cluster.
ha.cluster_server = 127.0.0.1:5002

# List of other known instances in this cluster
ha.initial_hosts = 127.0.0.1:5001,127.0.0.1:5002,127.0.0.1:5003

# Reduce the default page cache memory allocation
dbms.pagecache.memory=500m
```

conf/neo4j-server.properties

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

# http port (for all data, administrative, and UI access)
org.neo4j.server.webserver.port=7475

# https port (for all data, administrative, and UI access)
org.neo4j.server.webserver.https.port=7485

# HA - High Availability
# SINGLE - Single mode, default.
org.neo4j.server.database.mode=HA
```

Neo4j instance #3 — ~/neo4j-03

conf/neo4j.properties:

```
# Unique server id for this Neo4j instance
# can not be negative id and must be unique
ha.server_id = 3

# IP and port for this instance to bind to for communicating data with the
# other neo4j instances in the cluster.
ha.server = 127.0.0.1:6365
online_backup_server = 127.0.0.1:6368

# IP and port for this instance to bind to for communicating cluster information
# with the other neo4j instances in the cluster.
```

```
ha.cluster_server = 127.0.0.1:5003

# List of other known instances in this cluster
ha.initial_hosts = 127.0.0.1:5001,127.0.0.1:5002,127.0.0.1:5003

# Reduce the default page cache memory allocation
dbms.pagecache.memory=500m
```

conf/neo4j-server.properties

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

# http port (for all data, administrative, and UI access)
org.neo4j.server.webserver.port=7476

# https port (for all data, administrative, and UI access)
org.neo4j.server.webserver.https.port=7486

# HA - High Availability
# SINGLE - Single mode, default.
org.neo4j.server.database.mode=HA
```

Start the Neo4j Servers

Start the Neo4j servers as normal. Note the startup order does not matter.

```
localhost:~/neo4j-01$ ./bin/neo4j start
```

```
localhost:~/neo4j-02$ ./bin/neo4j start
```

```
localhost:~/neo4j-03$ ./bin/neo4j start
```

Now, you should be able to access the 3 servers and check their HA status:

http://127.0.0.1:7474/webadmin/#/info/org.neo4j/High%20Availability/

http://127.0.0.1:7475/webadmin/#/info/org.neo4j/High%20Availability/

http://127.0.0.1:7476/webadmin/#/info/org.neo4j/High%20Availability/

25.7. REST endpoint for HA status information

Introduction

A common use case for Neo4j HA clusters is to direct all write requests to the master while using slaves for read operations, distributing the read load across the cluster and and gain failover capabilities for your deployment. The most common way to achieve this is to place a load balancer in front of the HA cluster, an example being shown with HA Proxy. As you can see in that guide, it makes use of a REST endpoint to discover which instance is the master and direct write load to it. In this section, we'll deal with this REST endpoint and explain its semantics.

The endpoints

Each HA instance comes with 3 endpoints regarding its HA status. They are complimentary but each may be used depending on your load balancing needs and your production setup. Those are:

- `/db/manage/server/ha/master`
- `/db/manage/server/ha/slave`
- `/db/manage/server/ha/available`

The `/master` and `/slave` endpoints can be used to direct write and non-write traffic respectively to specific instances. This is the optimal way to take advantage of Neo4j's scaling characteristics. The `/available` endpoint exists for the general case of directing arbitrary request types to instances that are available for transaction processing.

To use the endpoints, perform an HTTP GET operation on either and the following will be returned:

HA REST endpoint responses

Endpoint	Instance State	Returned Code	Body text
`/db/manage/server/ha/master`	Master	`200 OK`	`true`
	Slave	`404 Not Found`	`false`
	Unknown	`404 Not Found`	`UNKNOWN`
`/db/manage/server/ha/slave`	Master	`404 Not Found`	`false`
	Slave	`200 OK`	`true`
	Unknown	`404 Not Found`	`UNKNOWN`
`/db/manage/server/ha/available`	Master	`200 OK`	`master`
	Slave	`200 OK`	`slave`
	Unknown	`404 Not Found`	`UNKNOWN`

Examples

From the command line, a common way to ask those endpoints is to use `curl`. With no arguments, `curl` will do an HTTP `GET` on the URI provided and will output the body text, if any. If you also want to get the response code, just add the `-v` flag for verbose output. Here are some examples:

- Requesting `master` endpoint on a running master with verbose output

```
#> curl -v localhost:7474/db/manage/server/ha/master
* About to connect() to localhost port 7474 (#0)
*   Trying ::1...
* connected
* Connected to localhost (::1) port 7474 (#0)
> GET /db/manage/server/ha/master HTTP/1.1
```

```
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
> Host: localhost:7474
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Access-Control-Allow-Origin: *
< Transfer-Encoding: chunked
< Server: Jetty(6.1.25)
<
* Connection #0 to host localhost left intact
true* Closing connection #0
```

- Requesting `slave` endpoint on a running master without verbose output:

```
#> curl localhost:7474/db/manage/server/ha/slave
false
```

- Finally, requesting the `master` endpoint on a slave with verbose output

```
#> curl -v localhost:7475/db/manage/server/ha/master
* About to connect() to localhost port 7475 (#0)
*   Trying ::1...
* connected
* Connected to localhost (::1) port 7475 (#0)
> GET /db/manage/server/ha/master HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
> Host: localhost:7475
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Content-Type: text/plain
< Access-Control-Allow-Origin: *
< Transfer-Encoding: chunked
< Server: Jetty(6.1.25)
<
* Connection #0 to host localhost left intact
false* Closing connection #0
```

Unknown status

The `UNKNOWN` status exists to describe when a Neo4j instance is neither master nor slave. For example, the instance could be transitioning between states (master to slave in a recovery scenario or slave being promoted to master in the event of failure). If the `UNKNOWN` status is returned, the client should not treat the instance as a master or a slave and should instead pick another instance in the cluster to use, wait for the instance to transit from the `UNKNOWN` state, or undertake restorative action via systems admin.

25.8. 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.

For this tutorial we will assume a Linux environment with HAProxy already installed. See http://haproxy.1wt.eu/ for downloads and installation instructions.

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
    option httpchk GET /db/manage/server/ha/available
    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.

Optimizing for reads and writes

Neo4j provides a catalogue of *health check URLs* (see Section 25.7, "REST endpoint for HA status information" [485]) that HAProxy (or any load balancer for that matter) can use to distinguish machines using HTTP response codes. In the example above we used the `/available` endpoint, which directs requests to machines that are generally available for transaction processing (they are alive!).

However, it is possible to have requests directed to slaves only, or to the master only. If you are able to distinguish in your application between requests that write, and requests that only read, then you can take advantage of two (logical) load balancers: one that sends all your writes to the master, and one that sends all your read-only requests to a slave. In HAProxy you build logical load balancers by adding multiple `backend`s.

The trade-off here is that while Neo4j allows slaves to proxy writes for you, this indirection unnecessarily ties up resources on the slave and adds latency to your write requests. Conversely, you don't particularly want read traffic to tie up resources on the master; Neo4j allows you to scale out for reads, but writes are still constrained to a single instance. If possible, that instance should exclusively do writes to ensure maximum write performance.

The following example excludes the master from the set of machines using the `/slave` endpoint.

```
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 /db/manage/server/ha/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
```

Note
In practice, writing to a slave is uncommon. While writing to slaves has the benefit of ensuring that data is persisted in two places (the slave and the master), it comes at a cost. The cost is that the slave must immediately become consistent with the master by applying any missing transactions and then synchronously apply the new transaction with the master. This is a more expensive operation than writing to the master and having the master push changes to one or more slaves.

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 26. Backup

Note
The Backup features are only available in the Neo4j Enterprise Edition.

26.1. Introducing Backup

Backups are performed over the network, from a running Neo4j server and into a local copy of the database store (the backup). The backup is run using the `neo4j-backup` tool, which is provided alongside Neo4j Enterprise.

Important
Neo4j Server must be configured to run a backup service. This is enabled via the configuration parameter `online_backup_enabled`, and is enabled by default. The interface and port the backup service listens on is configured via the parameter `online_backup_server` and defaults to the loopback interface and port 6362. It is typical to reconfigure this to listen on an external interface, by setting `online_backup_server=<my-host-ip-address>:6362`. It can also be configured to listen on all interfaces by setting `online_backup_server=0.0.0.0:6362`.

Performing a backup requires specifying the target host, an optional port, and the backup location. The backup tool will automatically select a full or incremental backup, based on whether an existing backup is present at that location.

The relevant configuration settings are found below.

Online backup configuration settings

- online_backup_enabled: Enable support for running online backups.
- online_backup_server: Listening server for online backups.

online_backup_enabled

Description	Enable support for running online backups.
Valid values	is a boolean.
Default value	`true`

online_backup_server

Description	Listening server for online backups.
Valid values	is a hostname and port.
Default value	`0.0.0.0:6362-6372`

Consistency check configuration settings

- consistency_check_execution_order: Execution order of store cross-checks to be used when running consistency check.
- consistency_check_graph: Perform checks between nodes, relationships, properties, types and tokens.
- consistency_check_indexes: Perform checks on indexes.
- consistency_check_label_scan_store: Perform checks on the label scan store.
- consistency_check_property_owners: Perform optional additional checking on property ownership.
- consistency_check_report_file: File name for inconsistencies log file.

consistency_check_execution_order

Description	Execution order of store cross-checks to be used when running consistency check.
Valid values	is one of `SINGLE_THREADED`, `MULTI_PASS`.
Default value	`MULTI_PASS`

consistency_check_graph

Description	Perform checks between nodes, relationships, properties, types and tokens.

Valid values	is a boolean.
Default value	`true`

consistency_check_indexes

Description	Perform checks on indexes. Checking indexes is more expensive than checking the native stores, so it may be useful to turn off this check for very large databases.
Valid values	is a boolean.
Default value	`true`

consistency_check_label_scan_store

Description	Perform checks on the label scan store. Checking this store is more expensive than checking the native stores, so it may be useful to turn off this check for very large databases.
Valid values	is a boolean.
Default value	`true`

consistency_check_property_owners

Description	Perform optional additional checking on property ownership. This can detect a theoretical inconsistency where a property could be owned by multiple entities. However, the check is very expensive in time and memory, so it is skipped by default.
Valid values	is a boolean.
Default value	`false`

consistency_check_report_file

Description	File name for inconsistencies log file. If not specified, logs to a file in the store directory.
Valid values	is a path which is relative to .

26.2. Server and Embedded

```
# Performing a full backup: create a blank directory and run the backup tool
mkdir /mnt/backup/neo4j-backup
./bin/neo4j-backup -host 192.168.1.34 -to /mnt/backup/neo4j-backup

# Performing an incremental backup: just specify the location of your previous backup
./bin/neo4j-backup -host 192.168.1.34 -to /mnt/backup/neo4j-backup

# Performing an incremental backup where the service is listening on a non-default port
./bin/neo4j-backup -host 192.168.1.34 -port 9999 -to /mnt/backup/neo4j-backup
```

Incremental Backups

An incremental backup is performed whenever an existing backup directory is specified. The backup tool will then copy any new transactions from the Neo4j server and apply them to the backup. The result will be an updated backup that is consistent with the current server state.

However, the incremental backup may fail for a number of reasons:

- If the existing directory doesn't contain a valid backup.
- If the existing directory contains a backup of a different database store.
- If the existing directory contains a backup from a previous database version.

Note
Note that when copying the outstanding transactions, the backup tool needs access to the historical logical logs. These logical logs are kept by Neo4j and automatically removed after a period of time, based on the `keep_logical_logs` configuration. If the required logical logs have already been removed, the backup tool will do a full backup instead.

26.3. 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( "127.0.0.1" );
backup.full( backupPath.getPath() );
assertTrue( "Should be consistent", backup.isConsistent() );
backup.incremental( backupPath.getPath() );
```

For more information, please see the Javadocs for OnlineBackup[1]

[1] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/backup/OnlineBackup.html

26.4. 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. Just make sure the database isn't running while replacing the folder.

To restore from backup in a clustered environment, follow these steps:

1. Shut down all instances in the cluster
2. Restore the backup to the individual database folders
3. Restart the instances.

Make sure there's at least one *neostore.transaction.db.nnn* file included in the backup. If there isn't, start up one instance in stand-alone mode first, and issue one updating transaction against it (any sort of write, like creating a node). Then copy that database to all instances in your cluster.

Chapter 27. 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. See Section 27.1, “Securing access to the Neo4j Server” [497] for details.

27.1. Securing access to the Neo4j Server

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.

Server authentication and authorization

Neo4j requires clients to supply authentication credentials when accessing the REST API. Without valid credentials, access to the database will be forbidden.

The authentication and authorization data is stored under *data/dbms/auth*. If necessary, this file can be copied over to other neo4j instances to ensure they share the same username/password (see the section called “Copying security configuration from one instance to another” [300]).

Please refer to Section 21.3, “REST API Authentication and Authorization” [297] for additional details. When access Neo4j over unsecured networks, ensure HTTPS is configured and used for access (see the section called “HTTPS support” [497]).

If necessary, authentication may be disabled. This will allow any client to access the database without supplying authentication credentials.

```
# Disable authorization
dbms.security.auth_enabled=false
```

Warning
Disabling authentication is not recommended, and should only be done if the operator has a good understanding of their network security, including protection against cross-site scripting (XSS)[1] attacks via user browsers. Developers should not disable authentication if they have a local installation using the default listening ports.

HTTPS support

The Neo4j server includes built in support for SSL encrypted communication over HTTPS. The first time the server starts, it automatically generates a self-signed SSL certificate and a private key. Because the certificate is self signed, it is not safe to rely on for production use, instead, you should provide your own key and certificate for the server to use.

To provide your own key and certificate, replace the generated key and certificate, or change the *conf/neo4j-server.properties* file to set the location of your certificate and key:

```
# Certificate location (auto generated if the file does not exist)
org.neo4j.server.webserver.https.cert.location=ssl/snakeoil.cert

# Private key location (auto generated if the file does not exist)
```

[1] http://en.wikipedia.org/wiki/Cross-site_scripting

```
org.neo4j.server.webserver.https.key.location=ssl/snakeoil.key
```

Note that the key should be unencrypted. Make sure you set correct permissions on the private key, so that only the Neo4j server user can read/write it.

Neo4j also supports chained SSL certificates. This requires to have all certificates in PEM format combined in one file and the private key needs to be in DER format.

You can set what port the HTTPS connector should bind to in the same configuration file, as well as turn HTTPS off:

```
# Turn https-support on/off
org.neo4j.server.webserver.https.enabled=true

# https port (for all data, administrative, and UI access)
org.neo4j.server.webserver.https.port=443
```

Arbitrary code execution

By default, the Neo4j Server comes with some places where arbitrary code code execution can happen. These are the Section 21.17, “Traversals” [356] REST endpoints. To secure these, either disable them completely by removing offending plugins from the server classpath, or secure access to these URLs through proxies or Authorization Rules. Also, the Java Security Manager, see http://docs.oracle.com/javase/7/docs/technotes/guides/security/index.html can be used to secure parts of the codebase.

Server authorization rules

Administrators may require more fine-grained security policies in addition to the basic authorization and/or 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, security rules can be implemented and registered with the server. This makes scenarios like user and role based security and authentication against external lookup services possible. See `org.neo4j.server.rest.security.SecurityRule` in the javadocs downloadable from Maven Central (org.neo4j.app:neo4j-server)[2].

Caution
The use of Server Authorization Rules may interact unexpectedly with the built-in the section called “Server authentication and authorization” [497], if enabled.

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
```

[2] http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j.app%22%20AND%20a%3A%22neo4j-server%22

```
                // deployment-specific authorization logic here
    }

    @Override
    public String forUriPath()
    {
        return "/*";
    }

    @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; charset=UTF-8`

Example response

- 401: `Unauthorized`
- WWW-Authenticate: `Basic realm="WallyWorld"`

Using Wildcards to Target Security Rules

In this example, a security rule is registered to deny access to all URIs to the server by listing the rule(s) class(es) in *neo4j-server.properties*. In this case, the rule is registered using a wildcard URI path (where `*` characters can be used to signify any part of the path). For example `/users*` means the rule will be bound to any resources under the `/users` root path. Similarly `/users*type*` will bind the rule to resources matching URIs like `/users/fred/type/premium`.

```
org.neo4j.server.rest.security_rules=my.rules.PermanentlyFailingSecurityRuleWithWildcardPath
```

with the rule source code of:

```
public String forUriPath()
{
    return "/protected/*";
}
```

With this rule registered, any access to URIs under /protected/ will be denied by the server. Using wildcards allows flexible targeting of security rules to arbitrary parts of the server's API, including any unmanaged extensions or managed plugins that have been registered.

Example request

- GET `http://localhost:7474/protected/tree/starts/here/dummy/more/stuff`
- Accept: `application/json`

Example response

- 401: `Unauthorized`
- WWW-Authenticate: `Basic realm="WallyWorld"`

Using Complex Wildcards to Target Security Rules

In this example, a security rule is registered to deny access to all URIs matching a complex pattern. The config looks like this:

```
org.neo4j.server.rest.security_rules=my.rules.PermanentlyFailingSecurityRuleWithComplexWildcardPath
```

with the rule source code of:

```
public class PermanentlyFailingSecurityRuleWithComplexWildcardPath implements SecurityRule
{

    public static final String REALM = "WallyWorld"; // as per RFC2617 :-)

    @Override
    public boolean isAuthorized( HttpServletRequest request )
    {
        return false;
    }

    @Override
    public String forUriPath()
    {
        return "/protected/*/something/else/*/final/bit";
    }

    @Override
    public String wwwAuthenticateHeader()
    {
        return SecurityFilter.basicAuthenticationResponse(REALM);
    }
}
```

Example request

- GET `http://localhost:7474/protected/wildcard_replacement/x/y/z/something/else/more_wildcard_replacement/a/b/c/final/bit/more/stuff`
- `Accept: application/json`

Example response

- `401: Unauthorized`
- `WWW-Authenticate: Basic realm="WallyWorld"`

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.

Security in depth

Although the Neo4j server has a number of security features built-in (see the above chapters), for sensitive deployments it is often sensible to front against the outside world it with a proxy like Apache `mod_proxy` [3].

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

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` [4] plugin

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

[4] http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html

Chapter 28. Monitoring

Note
Most of the monitoring features are only available in the Enterprise edition 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 JMX[1].

[1] http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

28.1. Adjusting remote JMX access to the Neo4j Server

Per default, the Neo4j Enterprise Server edition do not allow remote 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/neo4j-wrapper.conf* files for details.

Make sure that *conf/jmx.password* has the correct file permissions. The owner of the file has to be the user that will run the service, and the permissions should be read only for that user. On Unix systems, this is `0600`.

On Windows, follow the tutorial at http://docs.oracle.com/javase/7/docs/technotes/guides/management/security-windows.html to set the correct permissions. If you are running the service under the Local System Account, the user that owns the file and has access to it should be SYSTEM.

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/neo4j-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://docs.oracle.com/javase/7/docs/technotes/guides/management/agent.html.

28.2. How to connect to a Neo4j instance using JMX and JConsole

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

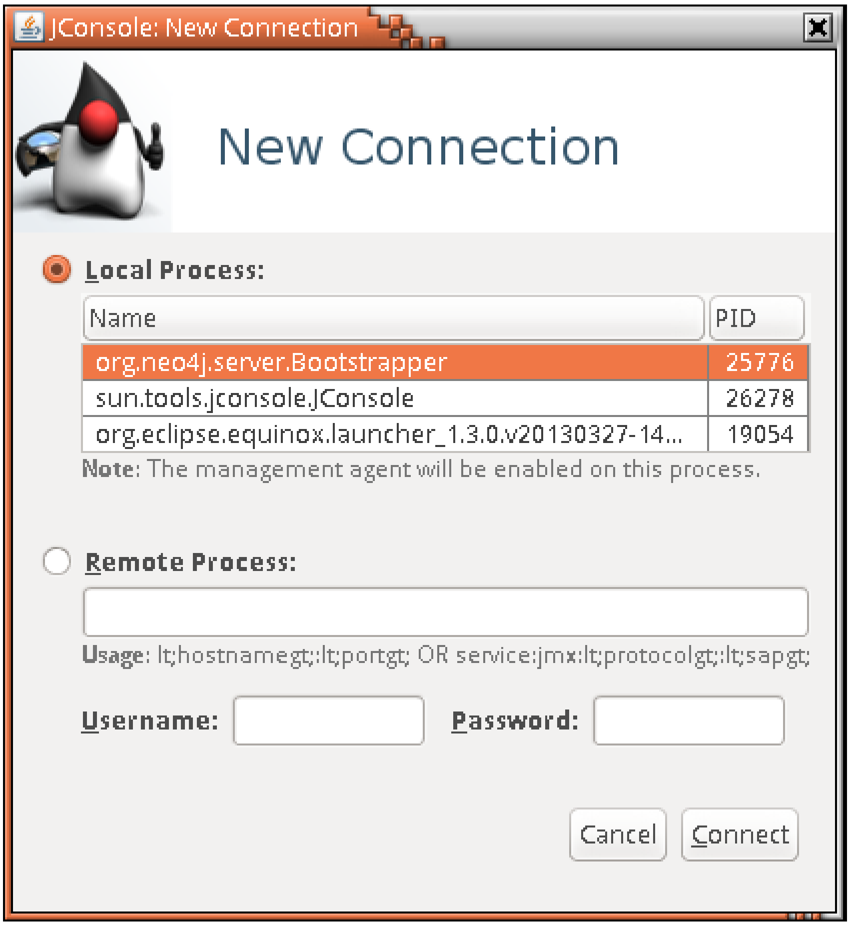
```
$NEO4j_HOME/bin/neo4j start
```

Now, start JConsole with

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

Connect to the process running your Neo4j database instance:

Figure 28.1. Connecting JConsole to the Neo4j Java process

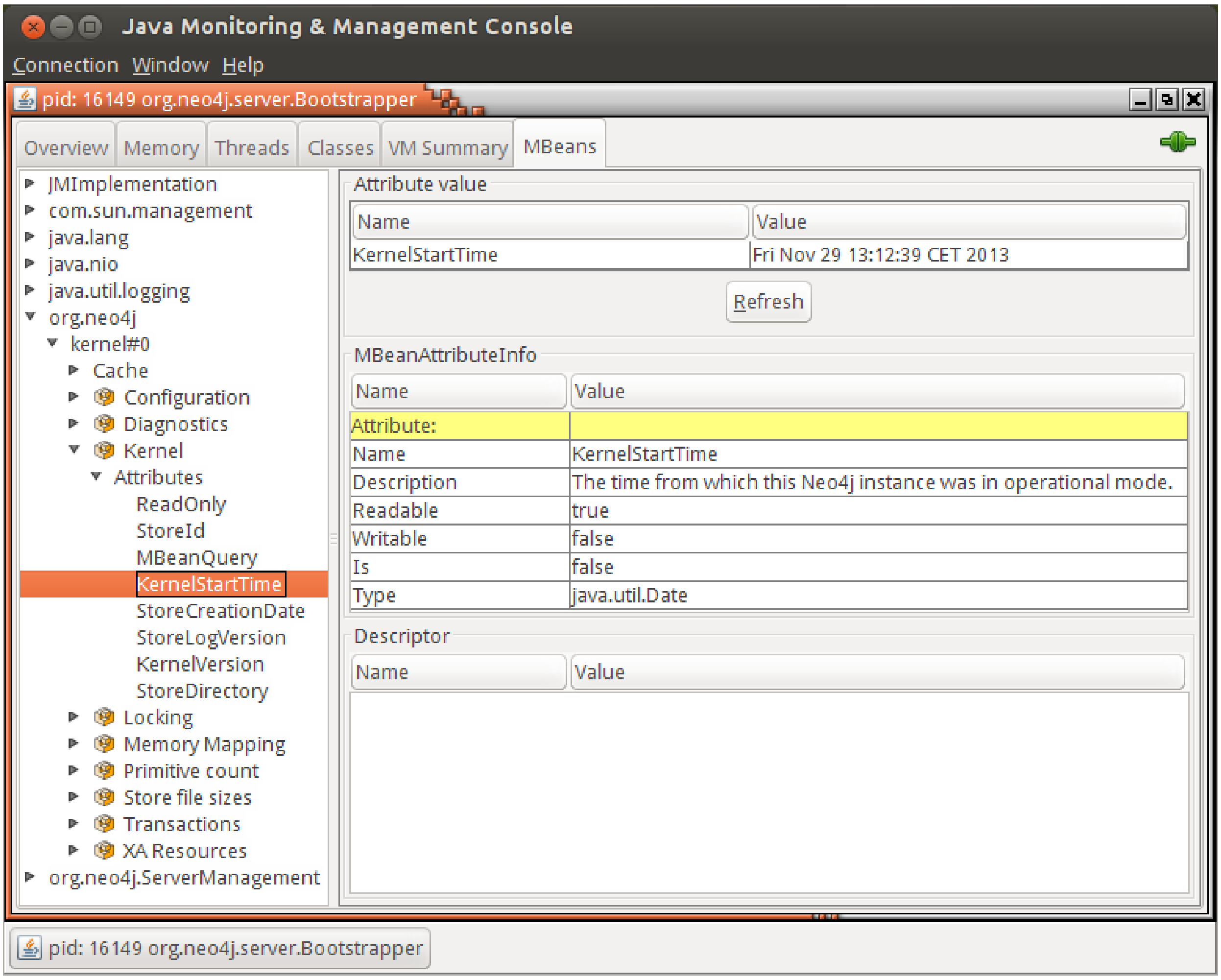


Now, beside the MBeans exposed by the JVM, you will see an `org.neo4j` section in the MBeans tab. Under that, you will have access to all the monitoring information exposed by Neo4j.

For opening JMX to remote monitoring access, please see Section 28.1, “Adjusting remote JMX access to the Neo4j Server” [503] and the JMX documention[2]. When using Neo4j in embedded mode, make sure to pass the `com.sun.management.jmxremote.port=portNum` or other configuration as JVM parameters to your running Java process.

[2] http://docs.oracle.com/javase/7/docs/technotes/guides/management/agent.html

Figure 28.2. Neo4j MBeans View



28.3. 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. See Section 35.11, “Reading a management attribute” [615] for an example.

Once you have access to this information, you can use it to for instance expose the values to SNMP[3] or other monitoring systems.

[3] http://en.wikipedia.org/wiki/Simple_Network_Management_Protocol

28.4. Reference of supported JMX MBeans

MBeans exposed by Neo4j

- Branched Store: Information about the branched stores present in this HA cluster member.
- Cache/NodeCache: Information about the caching in Neo4j.
- Cache/RelationshipCache: Information about the caching in Neo4j.
- Configuration: The configuration parameters used to configure Neo4j.
- Diagnostics: Diagnostics provided by Neo4j.
- High Availability: Information about an instance participating in a HA cluster.
- Index sampler: Handle index sampling.
- Kernel: Information about the Neo4j kernel.
- Locking: Information about the Neo4j lock status.
- Memory Mapping: The status of Neo4j memory mapping.
- Page cache: Information about the Neo4j page cache.
- Primitive count: Estimates of the numbers of different kinds of Neo4j primitives.
- Store file sizes: Information about the sizes of the different parts of the Neo4j graph store.
- Transactions: Information about the Neo4j transaction manager.

Note
For additional information on the primitive datatypes (`int`, `long` etc.) used in the JMX attributes, please see Property value types [597] in the section called "Properties" [7].

MBean Branched Store (org.neo4j.management.BranchedStore) Attributes

Name	Description	Type	Read	Write
Information about the branched stores present in this HA cluster member				
`BranchedStores`	A list of the branched stores	`org.neo4j.management.BranchedStoreInfo[] as CompositeData[]`	yes	no

MBean Cache/NodeCache (org.neo4j.management.Cache) Attributes

Name	Description	Type	Read	Write
Information about the caching in Neo4j				
`CacheSize`	The size of this cache (nr of entities or total size in bytes)	`long`	yes	no
`CacheType`	The type of cache used by Neo4j	`String`	yes	no
`HitCount`	The number of times a cache query returned a result	`long`	yes	no
`MissCount`	The number of times a cache query did not return a result	`long`	yes	no

MBean Cache/NodeCache (org.neo4j.management.Cache) Operations

Name	Description	ReturnType	Signature
`clear`	Clears the Neo4j caches	`void`	`(no parameters)`

MBean Cache/RelationshipCache (org.neo4j.management.Cache) Attributes

Name	Description	Type	Read	Write
Information about the caching in Neo4j				

Name	Description	Type	Read	Write
`CacheSize`	The size of this cache (nr of entities or total size in bytes)	`long`	yes	no
`CacheType`	The type of cache used by Neo4j	`String`	yes	no
`HitCount`	The number of times a cache query returned a result	`long`	yes	no
`MissCount`	The number of times a cache query did not return a result	`long`	yes	no

MBean Cache/RelationshipCache (org.neo4j.management.Cache) Operations

Name	Description	ReturnType	Signature
`clear`	Clears the Neo4j caches	`void`	`(no parameters)`

MBean Configuration (org.neo4j.jmx.impl.ConfigurationBean) Attributes

Name	Description	Type	Read	Write
	The configuration parameters used to configure Neo4j			
`cache.memory_ratio`	Set how much of the memory available for caching to use for caching. It is recommended to not have this value exceed 70 percent.	`String`	yes	no
`dbms.pagecache.memory`	Configuration attribute	`String`	yes	no
`ephemeral`	Configuration attribute	`String`	yes	no
`ha.initial_hosts`	Configuration attribute	`String`	yes	no
`ha.server_id`	Configuration attribute	`String`	yes	no
`jmx.port`	Configuration attribute	`String`	yes	no
`node_cache_array_fraction`	Fraction of the heap to dedicate to the array holding the nodes in the cache. Specifying `5` will let that array itself take up 5% out of the entire heap. Increasing this figure will reduce the chance of hash collisions at the expense of more heap used for it. More collisions means more redundant loading of objects from the low level cache.	`String`	yes	no
`node_cache_size`	Maximum size of the heap memory to dedicate to the cached nodes. Right before the maximum size is reached a purge is performed. The purge will evict objects from the cache until the cache size gets below 90% of the maximum size. Optimal settings for the maximum size depends on the size of your graph. The configured maximum size should leave enough room for other objects to coexist in the same JVM. At the same time it should be large enough to keep loading from the low level cache at a minimum. Predicted load on the JVM as	`String`	yes	no

Name	Description	Type	Read	Write
	well as layout of domain level objects should also be taken into consideration.			
`online_backup_enabled`	Enable support for running online backups	`String`	yes	no
`online_backup_server`	Listening server for online backups	`String`	yes	no
`relationship_cache_array_fraction`	Fraction of the heap to dedicate to the array holding the relationships in the cache. See node_cache_array_fraction for more information.	`String`	yes	no
`relationship_cache_size`	Maximum size of the heap memory to dedicate to the cached relationships. See node_cache_size for more information.	`String`	yes	no
`remote_shell_enabled`	Enable a remote shell server which Neo4j Shell clients can log in to.	`String`	yes	no
`remote_shell_host`	Remote host for shell. By default, the shell server listens only on the loopback interface, but you can specify the IP address of any network interface or use `0.0.0.0` for all interfaces.	`String`	yes	no
`remote_shell_name`	The name of the shell.	`String`	yes	no
`remote_shell_port`	The port the shell will listen on.	`String`	yes	no
`remote_shell_read_only`	Read only mode. Will only allow read operations.	`String`	yes	no
`store_dir`	Configuration attribute	`String`	yes	no

MBean Diagnostics (org.neo4j.management.Diagnostics) Attributes

Name	Description	Type	Read	Write
	Diagnostics provided by Neo4j			
`DiagnosticsProviders`	A list of the ids for the registered diagnostics providers.	`List (java.util.List)`	yes	no

MBean Diagnostics (org.neo4j.management.Diagnostics) Operations

Name	Description	ReturnType	Signature
`dumpAll`	Dump diagnostics information to JMX	`String`	`(no parameters)`
`dumpToLog`	Dump diagnostics information to the log.	`void`	`(no parameters)`
`dumpToLog`	Dump diagnostics information to the log.	`void`	`java.lang.String`
`extract`	Operation exposed for management	`String`	`java.lang.String`

MBean High Availability (org.neo4j.management.HighAvailability) Attributes

Name	Description	Type	Read	Write
	Information about an instance participating in a HA cluster			
`Alive`	Whether this instance is alive or not	`boolean`	yes	no
`Available`	Whether this instance is available or not	`boolean`	yes	no

Name	Description	Type	Read	Write
InstanceId	The identifier used to identify this server in the HA cluster	String	yes	no
InstancesInCluster	Information about all instances in this cluster	org.neo4j.management.ClusterMemberInfo[] as CompositeData[]	yes	no
LastCommittedTxId	The latest transaction id present in this instance's store	long	yes	no
LastUpdateTime	The time when the data on this instance was last updated from the master	String	yes	no
Role	The role this instance has in the cluster	String	yes	no

MBean High Availability (org.neo4j.management.HighAvailability) Operations

Name	Description	ReturnType	Signature
update	(If this is a slave) Update the database on this instance with the latest transactions from the master	String	(no parameters)

MBean Kernel (org.neo4j.jmx.Kernel) Attributes

Name	Description	Type	Read	Write
	Information about the Neo4j kernel			
KernelStartTime	The time from which this Neo4j instance was in operational mode.	Date (java.util.Date)	yes	no
KernelVersion	The version of Neo4j	String	yes	no
MBeanQuery	An ObjectName that can be used as a query for getting all management beans for this Neo4j instance.	javax.management.ObjectName	yes	no
ReadOnly	Whether this is a read only instance	boolean	yes	no
StoreCreationDate	The time when this Neo4j graph store was created.	Date (java.util.Date)	yes	no
StoreDirectory	The location where the Neo4j store is located	String	yes	no
StoreId	An identifier that, together with store creation time, uniquely identifies this Neo4j graph store.	String	yes	no
StoreLogVersion	The current version of the Neo4j store logical log.	long	yes	no

MBean Locking (org.neo4j.management.LockManager) Attributes

Name	Description	Type	Read	Write
	Information about the Neo4j lock status			
Locks	Information about all locks held by Neo4j	java.util.List<org.neo4j.kernel.info.LockInfo> as CompositeData[]	yes	no
NumberOfAvertedDeadlocks	The number of lock sequences that would have lead to a deadlock situation	long	yes	no

Name	Description	Type	Read	Write
	that Neo4j has detected and averted (by throwing DeadlockDetectedException).			

MBean Locking (org.neo4j.management.LockManager) Operations

Name	Description	ReturnType	Signature
`getContendedLocks`	getContendedLocks	`java.util.List<org.neo4j.kernel.info.LockInfo> as CompositeData[]`	`long`

MBean Memory Mapping (org.neo4j.management.MemoryMapping) Attributes

Name	Description	Type	Read	Write
The status of Neo4j memory mapping				
`MemoryPools`	Get information about each pool of memory mapped regions from store files with memory mapping enabled	`org.neo4j.management.WindowPoolInfo[] as CompositeData[]`	yes	no

MBean Page cache (org.neo4j.management.PageCache) Attributes

Name	Description	Type	Read	Write
Information about the Neo4j page cache				
`BytesRead`	Number of bytes read from durable storage	`long`	yes	no
`BytesWritten`	Number of bytes written to durable storage	`long`	yes	no
`EvictionExceptions`	Number of exceptions caught during page eviction	`long`	yes	no
`Evictions`	Number of page evictions	`long`	yes	no
`Faults`	Number of page faults	`long`	yes	no
`FileMappings`	Number of files that have been mapped into the page cache	`long`	yes	no
`FileUnmappings`	Number of files that have been unmapped from the page cache	`long`	yes	no
`Flushes`	Number of page flushes	`long`	yes	no
`Pins`	Number of page pins	`long`	yes	no
`Unpins`	Number of page unpins	`long`	yes	no

MBean Primitive count (org.neo4j.jmx.Primitives) Attributes

Name	Description	Type	Read	Write
Estimates of the numbers of different kinds of Neo4j primitives				
`NumberOfNodeIdsInUse`	An estimation of the number of nodes used in this Neo4j instance	`long`	yes	no
`NumberOfPropertyIdsInUse`	An estimation of the number of properties used in this Neo4j instance	`long`	yes	no
`NumberOfRelationshipIdsInUse`	An estimation of the number of relationships used in this Neo4j instance	`long`	yes	no

Name	Description	Type	Read	Write
NumberOf RelationshipTypeIds InUse	The number of relationship types used in this Neo4j instance	long	yes	no

MBean Store file sizes (org.neo4j.jmx.StoreFile) Attributes

Name	Description	Type	Read	Write
Information about the sizes of the different parts of the Neo4j graph store				
ArrayStoreSize	The amount of disk space used to store array properties, in bytes.	long	yes	no
LogicalLogSize	The amount of disk space used by the current Neo4j logical log, in bytes.	long	yes	no
NodeStoreSize	The amount of disk space used to store nodes, in bytes.	long	yes	no
PropertyStoreSize	The amount of disk space used to store properties (excluding string values and array values), in bytes.	long	yes	no
RelationshipStoreSize	The amount of disk space used to store relationships, in bytes.	long	yes	no
StringStoreSize	The amount of disk space used to store string properties, in bytes.	long	yes	no
TotalStoreSize	The total disk space used by this Neo4j instance, in bytes.	long	yes	no

MBean Transactions (org.neo4j.management.TransactionManager) Attributes

Name	Description	Type	Read	Write
Information about the Neo4j transaction manager				
LastCommittedTxId	The id of the latest committed transaction	long	yes	no
NumberOfCommitted Transactions	The total number of committed transactions	long	yes	no
NumberOfOpened Transactions	The total number started transactions	long	yes	no
NumberOfOpen Transactions	The number of currently open transactions	long	yes	no
NumberOfRolledBack Transactions	The total number of rolled back transactions	long	yes	no
PeakNumberOf Concurrent Transactions	The highest number of transactions ever opened concurrently	long	yes	no

MBean Index sampler (org.neo4j.management.IndexSamplingManager) Operations

Name	Description	ReturnType	Signature
triggerIndexSampling	triggerIndexSampling	void	java.lang.String, java.lang.String, boolean

Part VI. Tools

The Tools part describes available Neo4j tools and how to use them.

Chapter 29. Import Tool

The import tool is used to create a new Neo4j database from data in CSV files.

This chapter explains how to use the tool, format the input data and concludes with an example bringing everything together.

These are some things you'll need to keep in mind when creating your input files:

- Fields are comma separated by default but a different delimiter can be specified.
- All files must use the same delimiter.
- Multiple data sources can be used for both nodes and relationships.
- A data source can optionally be provided using multiple files.
- A header which provides information on the data fields must be on the first row of each data source.
- Fields without corresponding information in the header will not be read.
- UTF-8 encoding is used.

Tip
Indexes are not created during the import. Instead you'll need to add indexes afterwards (see the section called "Indexes" [10]).

Note
Data cannot be imported into an existing database using this tool.

- If you want to load small to medium sized CSV files see Section 11.6, "Load CSV" [177].
- If you want to bulk import into an existing database see Chapter 38, *Batch Insertion* [650].

29.1. CSV file Header format

The header row of each data source specifies how the fields should be interpreted. The same delimiter is used for the header row as for the rest of the data.

The header contains information for each field, with the format: `<name>:<field_type>`. The `<name>` is used as the property key for values, and ignored in other cases. The following `<field_type>` settings can be used for both nodes and relationships:

Property value	Use one of `int`, `long`, `float`, `double`, `boolean`, `byte`, `short`, `char`, `string` to designate the data type. If no data type is given, this defaults to `string`. To define an array type, append `[]` to the type. Array values are by default delimited by a `;`, but a different delimiter can be specified.
IGNORE	Ignore this field completely.

See below for the specifics of node and relationship data source headers.

Nodes

The following field types do additionally apply to node data sources:

ID	Each node must have a unique id which is used during the import. The ids are used to find the correct nodes when creating relationships. Note that the id has to be unique across all nodes in the import, even nodes with different labels.
LABEL	Read one or more labels from this field. For multiple labels, the values are separated by the array delimiter.

Relationships

For relationship data sources, there's three mandatory fields:

TYPE	The relationship type to use for the relationship.
START_ID	The id of the start node of the relationship to create.
END_ID	The id of the end node of the relationship to create.

ID Spaces

The import tool assumes that node identifiers are unique across node files. If this isn't the case then we can define an `id space`. Id spaces are defined in the `ID` field of node files.

For example, to specify the `Person` id space we would use the field type `ID(Person)` in our persons node file. We also need to reference that id space in our relationships file i.e. `START_ID(Person)` or `END_ID(Person)`.

29.2. Command Line Usage

Command

Under Unix/Linux/OSX, the command is named `neo4j-import`. Under Windows, the command is named `Neo4jImport.bat`.

Depending on the installation type, the tool is either available globally, or used by executing `./bin/neo4j-import` or `bin\Neo4jImport.bat` from inside the installation directory.

Options

--into <store-dir>	Database directory to import into. Must not contain existing database.
--nodes [:Label1:Label2] "<file1>,<file2>,…"	Node CSV header and data. Multiple files will be logically seen as one big file from the perspective of the importer. The first line must contain the header. Multiple data sources like these can be specified in one import, where each data source has its own header. Note that file groups must be enclosed in quotation marks.
--relationships [:RELATIONSHIP_TYPE] "<file1>,<file2>,…"	Relationship CSV header and data. Multiple files will be logically seen as one big file from the perspective of the importer. The first line must contain the header. Multiple data sources like these can be specified in one import, where each data source has its own header. Note that file groups must be enclosed in quotation marks.
--delimiter <delimiter-character>	Delimiter character, or *TAB*, between values in CSV data. The default option is `,`.
--array-delimiter <array-delimiter-character>	Delimiter character, or *TAB*, between array elements within a value in CSV data. The default option is `;`.
--quote <quotation-character>	Character to treat as quotation character for values in CSV data. The default option is `"`. Quotes inside quotes escaped like `"""Go away"", he said."` and `"\"Go away\", he said."` are supported. If you have set "`'`" to be used as the quotation character, you could write the previous example like this instead: `'"Go away", he said.'`
--id-type <id-type>	One out of [STRING, INTEGER, ACTUAL] and specifies how ids in node/relationship input files are treated. STRING: arbitrary strings for identifying nodes. INTEGER: arbitrary integer values for identifying nodes. ACTUAL: (advanced) actual node ids. The default option is `STRING`.Default value: STRING
--processors <max processor count>	(advanced) Max number of processors used by the importer. Defaults to the number of available processors reported by the JVM. There is a certain amount of minimum threads needed so for that reason there is no lower bound for this value. For optimal performance this value shouldn't be greater than the number of available processors.
--stacktrace	Enable printing of error stack traces.
--bad <file name>	Relationships that refer to nodes that cannot be found can, instead of making the import fail, be logged to a file specified by this option. Can be relative (to store directory) or absolute
--bad-tolerance <max number of bad entries>	Number of bad entries before the import is considered failed. This tolerance threshold is about relationships refering to missing nodes. Format errors in input data are still treated as errors. Default value: 1000
--input-encoding <character set>	Character set that input data is encoded in. Provided value must be one out of the available character sets in the JVM, as provided by

Charset#availableCharsets(). If no input encoding is provided, the default character set of the JVM will be used.

29.3. Basic Import Tool Example

Let's look at an example. We'll use a data set containing movies, actors and roles.

> **Tip**
> While you'll usually want to store your node identifier as a property on the node for looking it up later, it's not mandatory. If you don't want the identifier to be persisted then don't specify a property name in the `:ID` field.

First we'll look at the movies. Each movie has an id, which is used to refer to it in other data sources, a title and a year Along with these properties we'll also add the node labels `Movie` and `Sequel`.

By default the import tool expects CSV files to be comma delimited.

movies.csv

```
movieId:ID,title,year:int,:LABEL
tt0133093,"The Matrix",1999,Movie
tt0234215,"The Matrix Reloaded",2003,Movie;Sequel
tt0242653,"The Matrix Revolutions",2003,Movie;Sequel
```

Next up are the actors. They have an id - in this case a shorthand - and a name and all have the Actor label.

actors.csv

```
personId:ID,name,:LABEL
keanu,"Keanu Reeves",Actor
laurence,"Laurence Fishburne",Actor
carrieanne,"Carrie-Anne Moss",Actor
```

Finally we have the roles that an actor plays in a movie which will be represented by relationships in the database. In order to create a relationship between nodes we refer to the ids used in `actors.csv` and `movies.csv` in the `START_ID` and `END_ID` fields. We also need to provide a relationship type (in this case `ACTS_IN`) in the `:TYPE` field.

roles.csv

```
:START_ID,role,:END_ID,:TYPE
keanu,"Neo",tt0133093,ACTED_IN
keanu,"Neo",tt0234215,ACTED_IN
keanu,"Neo",tt0242653,ACTED_IN
laurence,"Morpheus",tt0133093,ACTED_IN
laurence,"Morpheus",tt0234215,ACTED_IN
laurence,"Morpheus",tt0242653,ACTED_IN
carrieanne,"Trinity",tt0133093,ACTED_IN
carrieanne,"Trinity",tt0234215,ACTED_IN
carrieanne,"Trinity",tt0242653,ACTED_IN
```

With all data in place, we execute the following command:

```
neo4j-import --into path_to_target_directory --nodes movies.csv --nodes actors.csv --relationships roles.csv
```

We're now ready to start up a database from the target directory. (see Section 23.2, "Server Installation" [426])

Once we've got the database up and running we can add appropriate indexes. (see Section 3.7, "Labels, Constraints and Indexes" [33].)

> **Tip**
> It is possible to import only nodes using the import tool - just don't specify a relationships file when calling `neo4j-import`. If you do this you'll need to create relationships later by another method - the import tool only works for initial graph population.

Customizing configuration options

We can customize the configuration options that the import tool uses (see the section called “Options” [517]) if our data doesn’t fit the default format. The following CSV files are delimited by `;`, use `|` as their array delimiter and use `'` for quotes.

movies2.csv

```
movieId:ID;title;year:int;:LABEL
tt0133093;'The Matrix';1999;Movie
tt0234215;'The Matrix Reloaded';2003;Movie|Sequel
tt0242653;'The Matrix Revolutions';2003;Movie|Sequel
```

actors2.csv

```
personId:ID;name;:LABEL
keanu;'Keanu Reeves';Actor
laurence;'Laurence Fishburne';Actor
carrieanne;'Carrie-Anne Moss';Actor
```

roles2.csv

```
:START_ID;role;:END_ID;:TYPE
keanu;'Neo';tt0133093;ACTED_IN
keanu;'Neo';tt0234215;ACTED_IN
keanu;'Neo';tt0242653;ACTED_IN
laurence;'Morpheus';tt0133093;ACTED_IN
laurence;'Morpheus';tt0234215;ACTED_IN
laurence;'Morpheus';tt0242653;ACTED_IN
carrieanne;'Trinity';tt0133093;ACTED_IN
carrieanne;'Trinity';tt0234215;ACTED_IN
carrieanne;'Trinity';tt0242653;ACTED_IN
```

We can then import these files with the following command line options:

```
neo4j-import --into path_to_target_directory --nodes movies2.csv --nodes actors2.csv --relationships roles2.csv --delimiter
 ";" --array-delimiter "|" --quote "'"
```

Using separate header files

When dealing with very large CSV files it’s more convenient to have the header in a separate file. This makes it easier to edit the header as you avoid having to open a huge data file just to change it.

Tip

`import-tool` can also process single file compressed archives. e.g. `--nodes nodes.csv.gz` or `--relationships rels.zip`

We’ll use the same data as in the previous example but put the headers in separate files.

movies3-header.csv

```
movieId:ID,title,year:int,:LABEL
```

movies3.csv

```
tt0133093,"The Matrix",1999,Movie
tt0234215,"The Matrix Reloaded",2003,Movie;Sequel
tt0242653,"The Matrix Revolutions",2003,Movie;Sequel
```

actors3-header.csv

```
personId:ID,name,:LABEL
```

actors3.csv

```
keanu,"Keanu Reeves",Actor
laurence,"Laurence Fishburne",Actor
carrieanne,"Carrie-Anne Moss",Actor
```

roles3-header.csv

```
:START_ID,role,:END_ID,:TYPE
```

roles3.csv

```
keanu,"Neo",tt0133093,ACTED_IN
keanu,"Neo",tt0234215,ACTED_IN
keanu,"Neo",tt0242653,ACTED_IN
laurence,"Morpheus",tt0133093,ACTED_IN
laurence,"Morpheus",tt0234215,ACTED_IN
laurence,"Morpheus",tt0242653,ACTED_IN
carrieanne,"Trinity",tt0133093,ACTED_IN
carrieanne,"Trinity",tt0234215,ACTED_IN
carrieanne,"Trinity",tt0242653,ACTED_IN
```

Note how the file groups are enclosed in quotation marks in the command:

```
neo4j-import --into path_to_target_directory --nodes movies3-header.csv,movies3.csv --nodes actors3-header.csv,actors3.csv --
relationships roles3-header.csv,roles3.csv
```

Multiple input files

As well as using a separate header file you can also provide multiple nodes or relationships files. This may be useful when processing the output from a Hadoop pipeline for example.

movies4-header.csv

```
movieId:ID,title,year:int,:LABEL
```

movies4-part1.csv

```
tt0133093,"The Matrix",1999,Movie
tt0234215,"The Matrix Reloaded",2003,Movie;Sequel
```

movies4-part2.csv

```
tt0242653,"The Matrix Revolutions",2003,Movie;Sequel
```

actors4-header.csv

```
personId:ID,name,:LABEL
```

actors4-part1.csv

```
keanu,"Keanu Reeves",Actor
laurence,"Laurence Fishburne",Actor
```

actors4-part2.csv

```
carrieanne,"Carrie-Anne Moss",Actor
```

roles4-header.csv

```
:START_ID,role,:END_ID,:TYPE
```

roles4-part1.csv

```
keanu,"Neo",tt0133093,ACTED_IN
keanu,"Neo",tt0234215,ACTED_IN
keanu,"Neo",tt0242653,ACTED_IN
laurence,"Morpheus",tt0133093,ACTED_IN
laurence,"Morpheus",tt0234215,ACTED_IN
```

roles4-part2.csv

```
laurence,"Morpheus",tt0242653,ACTED_IN
carrieanne,"Trinity",tt0133093,ACTED_IN
carrieanne,"Trinity",tt0234215,ACTED_IN
carrieanne,"Trinity",tt0242653,ACTED_IN
```

The call to `neo4j-import` would look like this:

```
neo4j-import --into path_to_target_directory --nodes movies4-header.csv,movies4-part1.csv,movies4-part2.csv --nodes actors4-header.csv,actors4-part1.csv,actors4-part2.csv --relationships roles4-header.csv,roles4-part1.csv,roles4-part2.csv
```

29.4. Types and Labels

Using the same label for every node

If you want to use the same node label(s) for every node in your nodes file you can do this by specifying the appropriate value as an option to `neo4j-import`. In this example we'll put the label `Movie` on every node specified in `movies5.csv`:

movies5.csv

```
movieId:ID,title,year:int
tt0133093,"The Matrix",1999
```

> **Tip**
> There's then no need to specify the `:LABEL` field in the node file if you pass it as a command line option. If you do then both the label provided in the file and the one provided on the command line will be added to the node.

And we'll put the labels `Movie` and `Sequel` on the nodes specified in `sequels5.csv`.

sequels5.csv

```
movieId:ID,title,year:int
tt0234215,"The Matrix Reloaded",2003
tt0242653,"The Matrix Revolutions",2003
```

actors5.csv

```
personId:ID,name
keanu,"Keanu Reeves"
laurence,"Laurence Fishburne"
carrieanne,"Carrie-Anne Moss"
```

roles5.csv

```
:START_ID,role,:END_ID,:TYPE
keanu,"Neo",tt0133093,ACTED_IN
keanu,"Neo",tt0234215,ACTED_IN
keanu,"Neo",tt0242653,ACTED_IN
laurence,"Morpheus",tt0133093,ACTED_IN
laurence,"Morpheus",tt0234215,ACTED_IN
laurence,"Morpheus",tt0242653,ACTED_IN
carrieanne,"Trinity",tt0133093,ACTED_IN
carrieanne,"Trinity",tt0234215,ACTED_IN
carrieanne,"Trinity",tt0242653,ACTED_IN
```

The call to `neo4j-import` would look like this:

```
neo4j-import --into path_to_target_directory --nodes:Movie movies5.csv --nodes:Movie:Sequel sequels5.csv --nodes:Actor
 actors5.csv --relationships roles5.csv
```

Using the same relationship type for every relationship

If you want to use the same relationship type for every relationship in your relationships file you can do this by specifying the appropriate value as an option to `neo4j-import`. In this example we'll put the relationship type `ACTS_IN` on every relationship specified in `roles6.csv`:

movies6.csv

```
movieId:ID,title,year:int,:LABEL
tt0133093,"The Matrix",1999,Movie
tt0234215,"The Matrix Reloaded",2003,Movie;Sequel
```

```
tt0242653,"The Matrix Revolutions",2003,Movie;Sequel
```

actors6.csv

```
personId:ID,name,:LABEL
keanu,"Keanu Reeves",Actor
laurence,"Laurence Fishburne",Actor
carrieanne,"Carrie-Anne Moss",Actor
```

roles6.csv

```
:START_ID,role,:END_ID
keanu,"Neo",tt0133093
keanu,"Neo",tt0234215
keanu,"Neo",tt0242653
laurence,"Morpheus",tt0133093
laurence,"Morpheus",tt0234215
laurence,"Morpheus",tt0242653
carrieanne,"Trinity",tt0133093
carrieanne,"Trinity",tt0234215
carrieanne,"Trinity",tt0242653
```

Tip
If you provide a relationship type on the command line and in the relationships file the one in the file will be applied.

The call to `neo4j-import` would look like this:

```
neo4j-import --into path_to_target_directory --nodes movies6.csv --nodes actors6.csv --relationships:ACTED_IN roles6.csv
```

Property Types

The type for properties specified in nodes and relationships files is defined in the header row. (see Section 29.1, "CSV file Header format" [516])

The following example creates a small graph containing one actor and one movie connected by an `ACTED_IN` relationship. There is a `roles` property on the relationship which contains an array of the characters played by the actor in a movie.

movies10.csv

```
movieId:ID,title,year:int,:LABEL
tt0099892,"Joe Versus the Volcano",1990,Movie
```

actors10.csv

```
personId:ID,name,:LABEL
meg,"Meg Ryan",Actor
```

roles10.csv

```
:START_ID,roles:string[],:END_ID,:TYPE
meg,"DeDe;Angelica Graynamore;Patricia Graynamore",tt0099892,ACTED_IN
```

The arguments to `neo4j-import` would be the following:

```
neo4j-import --into path_to_target_directory --nodes movies10.csv --nodes actors10.csv --relationships roles10.csv
```

29.5. ID handling

Each node processed by `neo4j-import` must provide a unique id. We use this id to find the correct nodes when creating relationships.

Working with sequential or auto incrementing identifiers

The import tool makes the assumption that identifiers are unique across node files. This may not be the case for data sets which use sequential, auto incremented or otherwise colliding identifiers. Those data sets can define id spaces where identifiers are unique within their respective id space.

For example if movies and people both use sequential identifiers then we would define `Movie` and `Actor` id spaces.

movies7.csv

```
movieId:ID(Movie),title,year:int,:LABEL
1,"The Matrix",1999,Movie
2,"The Matrix Reloaded",2003,Movie;Sequel
3,"The Matrix Revolutions",2003,Movie;Sequel
```

actors7.csv

```
personId:ID(Actor),name,:LABEL
1,"Keanu Reeves",Actor
2,"Laurence Fishburne",Actor
3,"Carrie-Anne Moss",Actor
```

We also need to reference the appropriate id space in our relationships file so it knows which nodes to connect together:

roles7.csv

```
:START_ID(Actor),role,:END_ID(Movie)
1,"Neo",1
1,"Neo",2
1,"Neo",3
2,"Morpheus",1
2,"Morpheus",2
2,"Morpheus",3
3,"Trinity",1
3,"Trinity",2
3,"Trinity",3
```

The command line arguments would remain the same as before:

```
neo4j-import --into path_to_target_directory --nodes movies7.csv --nodes actors7.csv --relationships:ACTED_IN roles7.csv
```

Missing nodes in relationships files

The import tool has a threshold of how many relationships can contain missing nodes (a bad relationship). By default it will allow `1000` bad relationships before the import process fails. If you want the import to immediately fail when it is unable to import a relationship then set this threshold to `0`.

In the following example there is a missing `emil` node referenced in the roles file.

movies9.csv

```
movieId:ID,title,year:int,:LABEL
tt0133093,"The Matrix",1999,Movie
tt0234215,"The Matrix Reloaded",2003,Movie;Sequel
tt0242653,"The Matrix Revolutions",2003,Movie;Sequel
```

actors9.csv

```
personId:ID,name,:LABEL
keanu,"Keanu Reeves",Actor
laurence,"Laurence Fishburne",Actor
carrieanne,"Carrie-Anne Moss",Actor
```

roles9.csv

```
:START_ID,role,:END_ID,:TYPE
keanu,"Neo",tt0133093,ACTED_IN
keanu,"Neo",tt0234215,ACTED_IN
keanu,"Neo",tt0242653,ACTED_IN
laurence,"Morpheus",tt0133093,ACTED_IN
laurence,"Morpheus",tt0234215,ACTED_IN
laurence,"Morpheus",tt0242653,ACTED_IN
carrieanne,"Trinity",tt0133093,ACTED_IN
carrieanne,"Trinity",tt0234215,ACTED_IN
carrieanne,"Trinity",tt0242653,ACTED_IN
emil,"Emil",tt0133093,ACTED_IN
```

The command line arguments would remain the same as before:

```
neo4j-import --into path_to_target_directory --nodes movies9.csv --nodes actors9.csv --relationships roles9.csv --bad bad-relationships-default-not-imported.bad.adoc
```

Since there was only one bad relationship the import process will complete successfully and a `not-imported.bad` file will be created and populated with the failed relationships.

not-imported.bad

```
emil in InputRelationship:
   source: /opt/teamcity-agent/work/db215c57ad4b0a74/target/community/import-tool/target/docs/ops/roles9.csv:11
   properties: [role, Emil]
   startNode: emil
   endNode: tt0133093
   type: ACTED_IN
```

Chapter 30. Web Interface

The Neo4j Web Interface is the primary user interface for Neo4j.

The tool is available at http://127.0.0.1:7474/ after you have installed the Neo4j Server.

See the tool itself for more information!

Chapter 31. Neo4j Shell

Neo4j shell is a command-line shell for running Cypher queries. There's also commands to get information about the database. In addition, you can browse 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's a nice tool for development and debugging. This guide will show you how to get it going!

31.1. Starting the shell

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

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

Under windows, the command is named `Neo4jShell.bat`.

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

To connect to a running Neo4j database, use the section called "Read-only mode" [530] for local databases and see the section called "Enabling the shell server" [529] for remote databases.

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

Enabling the shell server

Shell is enabled from the main configuration of Neo4j, see Section 24.5, "Configuration Settings" [445]. Here's the available settings:

Settings for the remote shell extension

- remote_shell_enabled: Enable a remote shell server which Neo4j Shell clients can log in to.
- remote_shell_host: Remote host for shell.
- remote_shell_name: The name of the shell.
- remote_shell_port: The port the shell will listen on.
- remote_shell_read_only: Read only mode.

remote_shell_enabled

Description	Enable a remote shell server which Neo4j Shell clients can log in to.
Valid values	is a boolean.
Default value	`false`

remote_shell_host

Description	Remote host for shell. By default, the shell server listens only on the loopback interface, but you can specify the IP address of any network interface or use `+<<config_0.0.0.0,0.0.0.0>>+` for all interfaces.
Valid values	is a string which must be a valid name.
Default value	`127.0.0.1`

remote_shell_name

Description	The name of the shell.
Valid values	is a string which must be a valid name.
Default value	`shell`

remote_shell_port

Description	The port the shell will listen on.
Valid values	is an integer which must be a valid port number (is in the range `0` to `65535`).
Default value	`1337`

remote_shell_read_only

Description	Read only mode. Will only allow read operations.
Valid values	is a boolean.

Default value	`false`

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.

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)$
```

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
```

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
```

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.

And even to another neo4j-shell, e.g. for importing a `dump` of another database or cypher result. When used with command mode the shell will not output a welcome message. So some examples of usage:

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

Pass Neo4j configuration options

By setting the `-config` switch, you can provide a properties file that will be used to configure your Neo4j instance, if started in embedded mode.

```
$ neo4j-shell -config conf/neo4j.properties -path mydb
```

Execute a file and then exit

To execute commands from a file and then exit just provide a `-file filename`. This is faster than piping to the shell which still handles the input as if it was user input.

For example reading a dump file directly from the command line and executing it against the given database. For example:

```
$ neo4j-shell -file export.cql > result.txt
```

Supplying `-` as the filename reads from `stdin` instead.

31.2. 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.

31.3. 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`.

31.4. 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[1] 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.

[1] http://www.json.org/

31.5. 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 (see the section called "Environment variables" [536]) and the default being `.*name.*,.*title.*`

31.6. 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:

Comments

Single line comments, which will be ignored, can be made by using the prefix `//`. Example:

```
// This is a comment
```

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. When the shell first starts you are not positioned on any entity, but you can `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 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.

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 31.4, "Filters" [533] and for information about how to specify direction see Section 31.3, "Enum options" [532]. Use `man ls` for more info.

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.

Setting, renaming and removing properties

Property operations are done with the `set`, `mv` and `rm` commands. These commands operates on the current node/relationship.

Use `set <key> <value>`, optionally with the `-t` option (for value type), to set 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.

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.

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. Allowed values are all parseable JSON strings, so maps `{age:10,name:"Mattias"}` and arrays `[1,2,3]` are also supported.

Variables can also be assigned to each other. E.g. `a=b` will result in `a` containing the value of `b`.

This becomes especially interesting as all shell variables are automatically passed to cypher statements as parameters. That makes it easy to query for certain start nodes or create nodes and relationships with certain provided properties (as maps).

Values are removed by setting them to `null` or an empty value. List environment variables using `env`

Executing groovy/python scripts

The shell has support for executing scripts, such as Groovy[2] and Python[3] (via Jython[4]). 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.

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

[2] http://groovy.codehaus.org
[3] http://www.python.org
[4] http://www.jython.org

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 31.4, “Filters” [533]. See Section 31.3, “Enum options” [532] for the traversal order option. Even relationship types/directions are supplied using the same format as filters.

Query with Cypher

You can use Cypher to query the graph. For that, use the `match` or `start` command. You can also use `create` statements to create nodes and relationships and use the `cypher VERSION` prefix to select a certain cypher version.

> **Tip**
> Cypher queries need to be terminated by a semicolon `;`.

Cypher commands are given all shell variables as parameters and the special `self` parameter for the current node or relationship.

- `start n = node(0) return n;` will give you a listing of the node with ID 0
- `cypher 1.9 start n = node(0) return n;` will execute the query with Cypher version 1.9
- `START n = node({self}) MATCH (n)-[:KNOWS]->(friend) RETURN friend;` will return the nodes connected to the current node.
- `START n=node({me}) CREATE (me)-[r:KNOWS]->(friend {props});` will create the friend and the relationship according to the variables available.

Listing Indexes and Constraints

The `schema` command allows to list all existing indexes and constraints together with their current status.

> **Note**
> This command does not list legacy indexes. For working with legacy indexes, please see the section called “Legacy Indexing” [537].

List all indexes and constraints:

```
schema
```

List indexes or constraints on `:Person` nodes for the property `name`:

```
schema -l :Person -p name
```

The `schema` command supports the following parameters:

- `-l :Label` only list indexes or constraints for the given label `:Label`
- `-p propertyKey` only list indexes or constraints for the given property key `propertyKey`
- `-v` if an index is in the `FAILED` state, print a verbose error cause if available

Indexes and constraints can be created or removed using Cypher or the Java Core API. They are updated automatically whenever the graph is changed. See the section called “Schema” [9] for more information.

Legacy Indexing

It’s possible to query and manipulate legacy indexes via the index command.

Example: `index -i persons name` (will index the name for the current node or relationship in the "persons" legacy index).

- `-g` will do exact lookup in the legacy index and display hits. You can supply `-c` with a command to be executed for each hit.
- `-q` will ask the legacy 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 into a legacy 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 a legacy index for the current node/relationship. Key and value are optional.
- `-t` will set the legacy index type to work with, for example `index -t Relationship --delete friends` will delete the `friends` relationship index.

Transactions

It is useful to be able to test changes, and then being able to commit or rollback said changes.

Transactions can be nested. With a nested transaction, a commit does not write any changes to disk, except for the top level transaction. A rollback, however works regardless of the level of the transaction. It will roll back all open transactions.

- `begin transaction` Starts a transaction.
- `commit` Commits a transaction.
- `rollback` Rollbacks all open transactions.

Dumping the database or Cypher statement results

Experimental feature
The `dump` command has incomplete functionality. It might not work for your use case or data size.

As a simple way of exporting a database or a subset of it, the `dump` command converts the graph of a Cypher result or the whole database into a single Cypher `create` statement.

Examples:

- `dump` dumps the whole database as single cypher create statement
- `dump START n=node({self}) MATCH p=(n)-[r:KNOWS*]->(m) RETURN n,r,m;` dumps the transitive friendship graph of the current node.
- `neo4j-shell -path db1 -c 'dump MATCH p=(n:Person {name:"Mattias"})-[r:KNOWS]->(m) RETURN p;' | neo4j-shell -path db2 -file -` imports the subgraph of the first database (db1) into the second (db2)

Example Dump Scripts

```
# create a new node and go to it
neo4j-sh (?)$ mknode --cd --np "{'name':'Neo'}"

# create a relationship
neo4j-sh (Neo,0)$ mkrel -c -d i -t LIKES --np "{'app':'foobar'}"

# Export the cypher statement results
neo4j-sh (Neo,0)$ dump MATCH (n)-[r]-(m) WHERE n = {self} return n,r,m;
```

```
begin
create (_0 {`name`:"Neo"})
create (_1 {`app`:"foobar"})
create _1-[:`LIKES`]->_0
;
commit
```

```
# create an index
neo4j-sh (?)$ create index on :Person(name);
+--------------------+
| No data returned. |
+--------------------+
Indexes added: 1
751 ms


# create one labeled node and a relationship
neo4j-sh (?)$ create (m:Person:Hacker {name:'Mattias'}), (m)-[:KNOWS]->(m);
+--------------------+
| No data returned. |
+--------------------+
Nodes created: 1
Relationships created: 1
Properties set: 1
Labels added: 2
405 ms


# Export the whole database including indexes
neo4j-sh (?)$ dump
begin
create index on :`Person`(`name`)
create (_0:`Person`:`Hacker` {`name`:"Mattias"})
create _0-[:`KNOWS`]->_0
;
commit
```

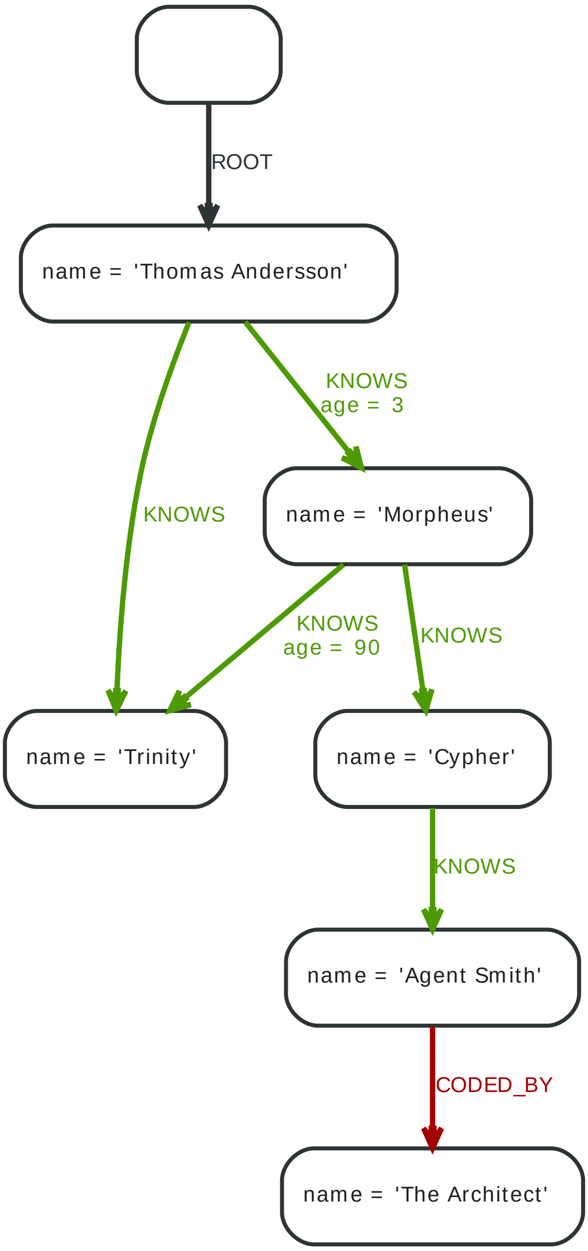
31.7. An example shell session

```
# Create a node
neo4j-sh (?)$ mknode --cd

# 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"

# send a cypher query
neo4j-sh (Jon,0)$ match n where id(n) = 0 return n;
+---------------------+
| n                   |
+---------------------+
| Node[0]{name:"Jon"} |
+---------------------+
1 row
384 ms


# 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 relationship 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)
```

31.8. 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:

Figure 31.1. Shell Matrix Example



Neo4j is configured for autoindexing, in this case with the following in the Neo4j configuration file:

```
node_auto_indexing=true
node_keys_indexable=name,age

relationship_auto_indexing=true
relationship_keys_indexable=ROOT,KNOWS,CODED_BY
```

The following is a sample shell session creating the Matrix graph and querying it.

```
# Create a reference node
neo4j-sh (?)$ mknode --cd

# 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 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 relationship
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

# Morpheus' friends, looking up Morpheus by name in the Neo4j autoindex
neo4j-sh (?)$ start morpheus = node:node_auto_index(name='Morpheus') match morpheus-[:KNOWS]-zionist return zionist.name;
+--------------------+
| zionist.name       |
+--------------------+
| "Trinity"          |
| "Cypher"           |
| "Thomas Andersson" |
+--------------------+
3 rows
132 ms


# Morpheus' friends, looking up Morpheus by name in the Neo4j autoindex
neo4j-sh (?)$ cypher 2.0 start morpheus = node:node_auto_index(name='Morpheus') match morpheus-[:KNOWS]-zionist return
zionist.name;
+--------------------+
| zionist.name       |
+--------------------+
```

```
| "Trinity"          |
| "Cypher"           |
| "Thomas Andersson" |
+--------------------+
3 rows
660 ms
```

Part VII. 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 32. Community Support

You can learn a lot about Neo4j at different *events.* To get information on upcoming Neo4j events, have a look here:

- http://neo4j.com/events/
- http://neo4j.meetup.com/

Get help from the Neo4j open source community; here are some starting points.

- The neo4j tag at stackoverflow: http://stackoverflow.com/questions/tagged/neo4j
- Neo4j Community Discussions: https://groups.google.com/forum/#!forum/neo4j
- Twitter: https://twitter.com/neo4j

Report a *bug* or add a *feature request*:

- https://github.com/neo4j/neo4j/issues

Questions regarding the *documentation:* The Neo4j Manual is published online with a comment function, please use that to post any questions or comments regarding the documentation. See http://neo4j.com/docs/2.2.1/.

Chapter 33. Contributing to Neo4j

The Neo4j project is an Open Source effort to bring fast complex data storage and processing to life. Every form of help is highly appreciated by the community - and you are not alone, see Section 33.6, “Contributors” [569]!

One crucial aspect of contributing to the Neo4j project is the Section 33.1, “Contributor License Agreement” [549].

In short: make sure to sign the CLA and send in the email, 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.

If you want to contribute, there are some good areas to start with, especially for getting in contact with the community, Chapter 32, *Community Support* [547].

To document your efforts, we highly recommend to read Section 33.3, “Writing Neo4j Documentation” [552].

33.1. Contributor License Agreement

Summary

We require all source code that is hosted on the Neo4j infrastructure to be contributed through the Neo4j Contributor License Agreement[1] (CLA). The purpose of the Neo4j Contributor License Agreement is to protect the integrity of the code base, which in turn protects the community around that code base: the founding entity Neo Technology, the Neo4j developer community and the Neo4j users. This kind of contributor agreement is common amongst free software and open source projects (it is in fact very similar to the widely signed Oracle Contributor Agreement[2]).

Please see the below or send a mail to admins [at] neofourjay.org if you have any other questions about the intent of the CLA. If you have a legal question, please ask a lawyer.

Common questions

Am I losing the rights to my own code?

No, the Neo4j CLA[3] only asks you to *share* your rights, not relinquish them. Unlike some contribution agreements that require you to transfer copyrights to another organization, the CLA does not take away your rights to your contributed intellectual property. When you agree to the CLA, you grant us joint ownership in copyright, and a patent license for your contributions. You retain all rights, title, and interest in your contributions and may use them for any purpose you wish. Other than revoking our rights, you can still do whatever you want with your code.

What can you do with my contribution?

We may exercise all rights that a copyright holder has, as well as the rights you grant in the Neo4j CLA[4] to use any patents you have in your contributions. As the CLA provides for joint copyright ownership, you may exercise the same rights as we in your contributions.

What are the community benefits of this?

Well, it allows us to sponsor the Neo4j projects and provide an infrastructure for the community, while making sure that we can include this in software that we ship to our customers without any nasty surprises. Without this ability, we as a small company would be hard pressed to release all our code as free software.

Moreover, the CLA lets us protect community members (both developers and users) from hostile intellectual property litigation should the need arise. This is in line with how other free software stewards like the Free Software Foundation - FSF[5] defend projects (except with the FSF, there's no shared copyright but instead you completely sign it over to the FSF). The contributor agreement also includes a "free software covenant," or a promise that a contribution will remain available as free software.

At the end of the day, you still retain all rights to your contribution and we can stand confident that we can protect the Neo4j community and the Neo Technology customers.

Can we discuss some items in the CLA?

Absolutely! Please give us feedback! But let's keep the legalese off the mailing lists. Please mail your feedback directly to cla (@t) neotechnology dot cöm and we'll get back to you.

I still don't like this CLA.

That's fine. You can still host it anywhere else, of course. Please do! We're only talking here about the rules for the infrastructure that we provide.

[1] http://dist.neo4j.org/neo4j-cla.pdf
[2] http://www.oracle.com/technetwork/community/oca-486395.html
[3] http://dist.neo4j.org/neo4j-cla.pdf
[4] http://dist.neo4j.org/neo4j-cla.pdf
[5] http://www.fsf.org

How to sign

When you've read through the CLA, please send a mail to cla (@t) neotechnology dot cöm. Include the following information:

- Your full name.
- Your e-mail address.
- An attached copy of the Neo4j CLA[6].
- That you agree to its terms.

For example:

```
Hi. My name is John Doe (john@doe.com).
I agree to the terms in the attached Neo4j Contributor License Agreement.
```

[6] http://dist.neo4j.org/neo4j-cla.pdf

33.2. Areas for contribution

Neo4j is a project with a vast ecosystem and a lot of space for contributions. Where you can and want to pitch in depends of course on your time, skill set and interests. Below are some of the areas that might interest you:

Neo4j Core Projects

- The Neo4j open issues[7] for some starting points for contribution
- See the GitHub Neo4j area[8] for a list of projects

Maintaining Neo4j Documentation

Some parts of the documentation need extra care from the community to stay up to date. They typically refer to different kinds of community contributions. The easiest way to contribute fixes is to comment at the online HTML version[9].

Drivers and bindings to Neo4j

- REST: see Chapter 7, *Languages* [95] for a list of active projects

[7] https://github.com/neo4j/neo4j/issues
[8] https://github.com/neo4j/
[9] http://neo4j.com/docs/milestone/

33.3. Writing Neo4j Documentation

Note
Other than writing documentation, you can help out by providing comments — head over to the online HTML version[10] to do that!

For how to build the manual see: readme[11]

The documents use the asciidoc format, see:

- Aciidoc Reference[12]
- AsciiDoc cheatsheet[13]

The cheatsheet is really useful!

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 `.asciidoc` 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.

By setting up a unit test accordingly, documentation can be written directly in the JavaDoc comment.

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.

Note that different ways to add documentation works best for different cases:

- For detail level documentation, it works well to write the documentation as part of unit tests (in the JavaDoc comment). In this case, you typically do not want to link to the source code in the documentation.
- For tutorial level documentation, the result will be best by writing a `.asciidoc` file containing the text. Source snippets and output examples can then be included from there. In this case you typically want to link to the source code, and users should be able to run it without any special setup.

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.

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

[10] http://neo4j.com/docs/milestone/
[11] https://github.com/neo4j/neo4j/blob/master/manual/README.asciidoc
[12] http://www.methods.co.nz/asciidoc/
[13] http://powerman.name/doc/asciidoc

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.

Writing

Put one sentence on each line. This makes it easy to move content around, and also easy to spot (too) long sentences.

Gotchas

- A chapter can't be empty. (the build will fail on the docbook xml validity check)
- 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!

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: the section called “Overall Flow” [552]

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[14]

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.

Text Formatting

- _Italics_ is rendered as *Italics* and used for emphasis.
- *Bold* is rendered as **Bold** and used sparingly, for strong emphasis only.
- +methodName()+ is rendered as `methodName()` and is used for literals as well (note: the content between the `+` signs *will* be parsed).
- `command` is rendered as `command` (typically used for command-line) (note: the content between the ` signs *will not* be parsed).
- Mono++space++d is rendered as Mono`space`d and is used for monospaced letters.
- 'my/path/' is rendered as *my/path/* (used for file names and paths).
- ``Double quoted'' (that is two grave accents to the left and two acute accents to the right) renders as “Double quoted”.
- `Single quoted' (that is a single grave accent to the left and a single acute accent to the right) renders as ‘Single quoted’.

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

[14] http://neo4j.org/

Here's how it's done:

```
NOTE: Note.
```

A multiline variation:

```
[TIP]
Tiptext.
Line 2.
```

Which is rendered as:

Tip
Tiptext. Line 2.

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:

Neo4j
the graph database

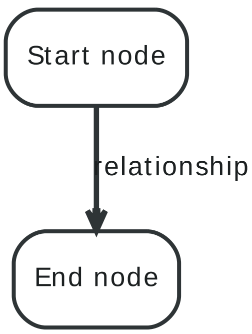
Static Graphviz/DOT

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"]
----
```

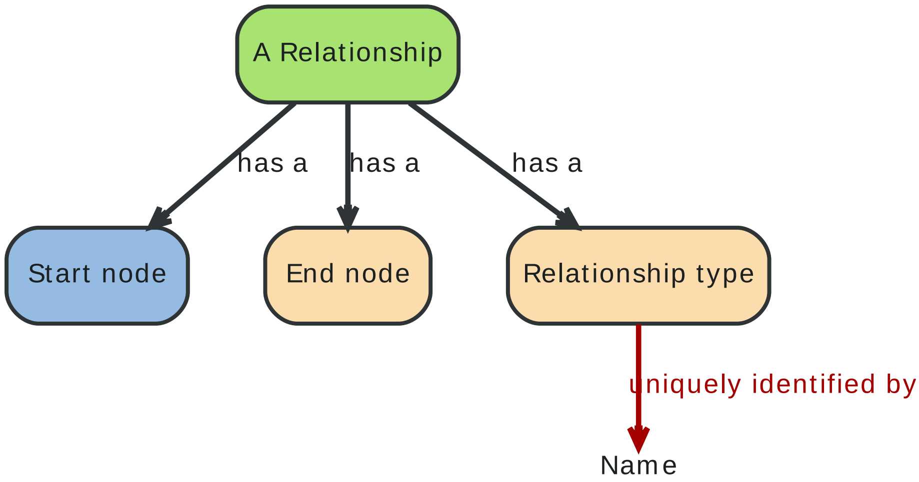
Which is rendered as:



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*.

Attributes

Common attributes you can use in documents:

- {neo4j-version} - rendered as "2.2.1"
- {neo4j-git-tag} - rendered as "2.2.1"
- {lucene-version} - rendered as "3_6_2"

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.

Sample Asciidoc attributes which can be used:

- {docdir} - root directory of the documents
- {nbsp} - non-breaking space

Comments

There's a separate build including comments. The comments show up with a yellow background. This build doesn't run by default, but after a normal build, you can use `make annotated` to build it. You can also use the resulting page to search for content, as the full manual is on a single page.

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!
////
```

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/JmxDocTest.java
classifier=test-sources
tag=getStartTime
----
```

This is how it renders:

```
private static Date getStartTimeFromManagementBean(
        GraphDatabaseService graphDbService )
{
    ObjectName objectName = JmxUtils.getObjectName( graphDbService, "Kernel" );
    Date date = JmxUtils.getAttribute( objectName, "KernelStartTime" );
    return date;
}
```

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

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.asciidoc` 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 (see below) 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", ""]
----
  N0 [
    label = "{Node\[0\]|name = \'you\'\l}"
  ]
  N1 [
    label = "{Node\[1\]|name = \'I\'\l}"
  ]
  N1 -> N0 [
    color = "#2e3436"
    fontcolor = "#2e3436"
    label = "know\n"
  ]
----

- A sample Cypher query:

[source,cypher]
----
START n = node(1)
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/neo4j/blob/{neo4j-git-tag}/community/embedded-examples/src/test/java/org/neo4j/examples/
DocumentationDocTest.java[DocumentationDocTest.java]

- The full source for this example as a source snippet, highlighted as Java code:

[snippet,java]
----
component=neo4j-examples
source=org/neo4j/examples/DocumentationDocTest.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.asciidoc[]
```

which renders the following chapter:

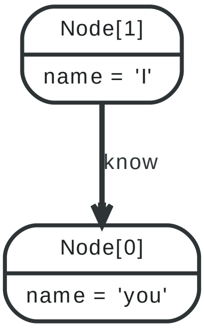
Hello world Sample Chapter

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 (see below) and programmatic adding with runtime data in the Java code.

- The annotated graph as GraphViz[15]-generated visualization:

Figure 33.1. Hello World Graph



- A sample Cypher query:

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

- A sample text output snippet:

```
Hello graphy world!
```

- a generated source link to the original GIThub source for this test:

DocumentationDocTest.java[16]

- The full source for this example as a source snippet, highlighted as Java code:

```
// START SNIPPET: _sampleDocumentation
package org.neo4j.examples;

import org.junit.Test;

import org.neo4j.kernel.impl.annotations.Documented;
import org.neo4j.test.GraphDescription.Graph;

import static org.neo4j.visualization.asciidoc.AsciidocHelper.createGraphVizWithNodeId;
import static org.neo4j.visualization.asciidoc.AsciidocHelper.createOutputSnippet;

public class DocumentationDocTest extends ImpermanentGraphJavaDocTestBase
{
```

[15] http://www.graphviz.org/
[16] https://github.com/neo4j/neo4j/blob/2.2.1/community/embedded-examples/src/test/java/org/neo4j/examples/DocumentationDocTest.java

```
    /**
     * 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 (see below) 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().addTestSourceSnippets( this.getClass(), "sampleDocumentation" );
        gen.get()
                .addGithubTestSourceLink( "github", this.getClass(),
                        "community/embedded-examples" );

        gen.get().addSnippet( "output",
                createOutputSnippet( "Hello graphy world!" ) );

        gen.get().addSnippet(
                "graph",
                createGraphVizWithNodeId( "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" ).getId()
                                     + ") return n" ) );
    }
}
// END SNIPPET: _sampleDocumentation
```

This is the end of this chapter.

Integrated Live Console

An interactive console can be added and will show up in the online HTML version. An optional title can be added, which will be used for the text of the button.

This is how to do it, using Cypher to define the data, with an empty line to separate it from the query:

```
.Interactive Example
[console]
----
CREATE (n {name: 'Neo4j'})

MATCH (n)
return n
----
```

And this is the result:

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

33.4. Translating the Neo4j Manual

To translate the Neo4j Manual, there's a special project setup to use. See the French translation project for an example: https://github.com/neo4j/manual-french

The project contains:

- *conf/* — configuration for the project.
- *docs/* — translated files for content provided by Neo4j modules.
- *po/* — translation files and po4a configuration files.
- *src/* — translated files for content provided by the original manual.
- *Makefile* — a makefile with project-specific configuration.
- *pom.xml* — Maven build configuration.

Prerequisites

- Apache Maven
- GNU Make
- Python
- Perl
- Perl module: `Unicode::GCString`

To check if you have the `Unicode::GCString` module installed, you can issue the following command:

```
perl -MUnicode::GCString -e ''
```

If there's no error, the module has been successfully installed on your system.

To install the module, you can use cpanminus[17]. For a convenient way to install it, see http://cpanmin.us. With cpanminus installed, execute this command:

```
cpanm Unicode::GCString
```

You will probably want to use a *.po* file editor as well, see the section called "Translation tools" [564].

Build flow and file layout

The build is essentially a two-step process. The first step generates or copies translated documents, while the second step is an ordinary AsciiDoc build using the output from the first step as sources.

Other than the *src/* and *docs/* diirectories of the project, the build generates files with the same layout in two more places:

1. *target/original/(src|docs)/* — the contents of the original manual. Note that's it easier to look for content here than to dig into the original manual itself.
2. *target/(src|docs)/* — the translated source to use for the AsciiDoc build.

The translated documents in *target/(src|docs)/* are generated in three steps:

1. It starts out as a copy of the original manual.
2. Next, any static translated files fromt the *src/* and *docs/* directories of the project are copied.
3. Finally, the translation files in the *po/* directory are used to generate translated documents.

Files produced by later steps will overwrite existing files from earlier steps.

17 http://search.cpan.org/dist/App-cpanminus/lib/App/cpanminus.pm

Adding a chapter to a translation file

The translation is split over multiple translation files, one per “part” of the manual. It’s all about making the translation easier to manage and the tools to perform well. The basic rule of thumb is that if some content is moved, it should likely still end up in the same translation file. In that case, the tools will even detect this and the translation will be moved automatically.

To add a document to a translation file, do like this:

```
make add DOCUMENT="src/introduction/the-neo4j-graphdb.asciidoc" PART="introduction"
```

If the translation file does not already exist, it will be created. The document will be added to the translation build configuration file as well. (The configuration is in the corresponding *.conf* file in the *po/* directory.)

If there exists a translated copy of the document at the location the `DOCUMENT` parameter points to, the script will attempt to populate the translation file with translated paragraphs from that document. Note that the structure of the document has to be a *perfect match*, or it will fail. However, the error messages are helpful, so just fix and try again until it works! Translation file and configuration are only changed when the first part succeeds.

> **Note**
> Only documents that need to be translated should be added. For example Cypher queries and query results should not be translated. In general, documents residing in a directory named *includes* should not be translated.

Also note that AsciiDoc `include::` lines are normally not part of the translation at all, but handled automatically. In case they need to be handled differently in a document, this has to be configured in the corresponding *.conf* file. For example a normal document entry in such a file can look like this:

```
[type: asciidoc] target/original/src/operations/index.asciidoc fr:target/src/operations/index.asciidoc
```

To configure a single document not to handle `include::` lines automatically, add the following at the end of the line:

```
opt: "-o definitions=target/tools/main/resources/conf/translate-includes"
```

Workflow

First, use Maven to set up the environment and download the original manual and documentation tools:

```
mvn clean package
```

To refresh the original manual and the tools, use the maven command again. For the sake of keeping in sync with the original manual, a daily run of this command is recommended.

Once things are set up, use `make` during work.

- `make` — same as `make preview`.
- `make add` — add a document to a translation file.
- `make preview` — refresh and build preview of the manual.
- `make refresh` — refresh translation files from the original and generated translated documents.

The preview of the translated manual is found in the *target/html/* directory.

The actual work on translation is done by editing translation files. Suggested tools for that are found below.

Translation tools

There are different editors for *.po* files containing the translations Below is a list of editors.

- Gtranslator[18]
- Lokalize[19]
- Virtaal[20]
- Poedit[21]

[18] http://projects.gnome.org/gtranslator/
[19] http://userbase.kde.org/Lokalize
[20] http://translate.sourceforge.net/wiki/virtaal/index
[21] http://www.poedit.net/

33.5. Contributing Code to Neo4j

Intro

The Neo4j community is a free software and open source community centered around software and components for the Neo4j Graph Database. It is sponsored by Neo Technology[22], which provides infrastructure (different kinds of hosting, documentation, etc) as well as people to manage it. The Neo4j community is an open community, in so far as it welcomes any member that accepts the basic criterias of contribution and adheres to the community's Code of Conduct.

Contribution can be in many forms (documentation, discussions, bug reports). This document outlines the rules of governance for a contributor of code.

Governance fundamentals

In a nutshell, you need to be aware of the following fundamentals if you wish to contribute code:

- All software published by the Neo4j project must have been contributed under the Neo4j Code Contributor License Agreement.
- Neo4j is a free software and open source community. As a contributor, you are free to place your work under any license that has been approved by either the Free Software Foundation[23] or the Open Source Initiative[24]. You still retain copyright, so in addition to that license you can of course release your work under any other license (for example a fully proprietary license), just not on the Neo4j infrastructure.
- The Neo4j software is split into components. A Git repository holds either a single or multiple components.
- The source code should follow the Neo4j Code Style and "fit in" with the Neo4j infrastructure as much as is reasonable for the specific component.

Contributor roles

Every individual that contributes code does so in the context of a role (a single individual can have multiple roles). The role defines their responsibilities and privileges:

- A *patch submitter* is a person who wishes to contribute a patch to an existing component. See Workflow below.
- A *committer* can contribute code directly to one or more components.
- A *component maintainer* is in charge of a specific component. They can:
  - commit code in their component's repository,
  - manage tickets for the repository,
  - grant push rights to the repository.
- A *Neo4j admin* manages the Neo4j infrastructure. They:
  - define new components and assign component maintainership,
  - drive, mentor and coach Neo4j component development.

Contribution workflow

Code contributions to Neo4j are normally done via Github Pull Requests, following the workflow shown below. Please check the pull request checklist before sending off a pull request as well.

1. Fork the appropriate Github repository.
2. Create a new branch for your specific feature or fix.

22 http://neotechnology.com
23 http://www.fsf.org/
24 http://opensource.org

3. Write unit tests for your code.
4. Write code.
5. Write appropriate Javadocs and Manual entries.
6. Commit changes.
7. Send pull request.

Pull request checklist

1. Sign the CLA.
2. Ensure that you have not added any merge commits.
3. Squash all your changes into a single commit.
4. Rebase against the latest master.
5. Run all relevant tests.
6. Send the request!

Unit Tests

You have a much higher chance of getting your changes accepted if you supply us with small, readable unit tests covering the code you've written. Also, make sure your code doesn't break any existing tests. *Note that there may be downstream components that need to be tested as well,* depending on what you change.

To run tests, use Maven rather than your IDE to ensure others can replicate your test run. The command for running Neo4j tests in any given component is `mvn clean test`.

The general structure of a unit test looks like this:

```
@Test
public void myTest()
{
    // Given
    [ Setup code here, setting the stage and parameters that are relevant ]

    // When
    [ The code that is being tested, preferrably just one line, like calling a method ]

    // Then
    [ Assertions on the result ]
}
```

Code Style

The Neo4j Code style is maintained on GitHub in styles for the different IDEs[25].

Commit messages

Please take some care in providing good commit messages. Use your common sense. In particular:

- Use *english*. This includes proper punctuation and correct spelling. Commit messages are supposed to convey some information at a glance — they're not a chat room.
- Remember that a commit is a *changeset*, which describes a cohesive set of changes across potentially many files. Try to group every commit as a logical change. Explain what it changes. If you have to redo work, you might want to clean up your commit log before doing a pull request.
- If you fix a bug or an issue that's related to a ticket, then refer to the ticket in the message. For example, *`‘Added this and then changed that. This fixes #14.”* Just mentioning #xxx in the commit will

[25] https://github.com/neo4j/neo4j.github.com/tree/master/code-style

connect it to the GitHub issue with that number, see GitHub issues[26]. Any of these synonyms will also work:
  - fixes #xxx
  - fixed #xxx
  - fix #xxx
  - closes #xxx
  - close #xxx
  - closed #xxx.
- Remember to convey *intent*. Don't be too brief but don't provide too much detail, either. That's what `git diff` is for.

Signing the CLA

One crucial aspect of contributing is the Contributor License Agreement. In short: make sure to sign the CLA, or the Neo4j project won't be able to accept your contribution.

Don't merge, use rebase instead!

Because we would like each contribution to be contained in a single commit, merge commits are not allowed inside a pull request. Merges are messy, and should only be done when necessary, eg. when merging a branch into master to remember where the code came from.

If you want to update your development branch to incorporate the latest changes from master, use git rebase. For details on how to use rebase, see Git manual on rebase: the Git Manual[27].

Single commit

If you have multiple commits, you should squash them into a single one for the pull request, unless there is some extraordinary reason not to. Keeping your changes in a single commit makes the commit history easier to read, it also makes it easy to revert and move features around.

One way to do this is to, while standing on your local branch with your changes, create a new branch and then interactively rebase your commits into a single one.

Interactive rebasing with Git

```
# On branch mychanges
git checkout -b mychanges-clean

# Assuming you have 4 commits, rebase the last four commits interactively:
git rebase -i HEAD~4

# In the dialog git gives you, keep your first commit, and squash all others into it.
# Then reword the commit description to accurately depict what your commit does.
# If applicable, include any issue numbers like so: #760
```

For more details, see the git manual: http://git-scm.com/book/en/Git-Tools-Rewriting-History#Changing-Multiple-Commit-Messages

If you are asked to modify parts of your code, work in your original branch (the one with multiple commits), and follow the above process to create a fixed single commit.

[26] https://github.com/blog/831-issues-2-0-the-next-generation
[27] http://git-scm.com/book/en/Git-Branching-Rebasing

33.6. Contributors

As an Open Source Project, the Neo4j User community extends its warmest thanks to all the contributors who have signed the Section 33.1, "Contributor License Agreement" [549] to date and are contributing to this collective effort.

name	GIThub ID
Johan Svensson	johan-neo[28]
Emil Eifrem	emileifrem[29]
Peter Neubauer	peterneubauer[30]
Mattias Persson	tinwelint[31]
Tobias Lindaaker	thobe[32]
Anders Nawroth	nawroth[33]
Andrés Taylor	systay[34]
Jacob Hansson	jakewins[35]
Jim Webber	jimwebber[36]
Josh Adell	jadell[37]
Andreas Kollegger	akollegger[38]
Chris Gioran	digitalstain[39]
Thomas Baum	tbaum[40]
Alistair Jones	apcj[41]
Michael Hunger	jexp[42]
Jesper Nilsson	jespernilsson[43]
Tom Sulston	tomsulston[44]
David Montag	dmontag[45]
Marlon Richert	marlonrichert[46]
Hugo Josefson	hugojosefson[47]
Vivek Prahlad	vivekprahlad[48]

[28] https://github.com/johan-neo
[29] https://github.com/emileifrem
[30] https://github.com/peterneubauer
[31] https://github.com/tinwelint
[32] https://github.com/thobe
[33] https://github.com/nawroth
[34] https://github.com/systay
[35] https://github.com/jakewins
[36] https://github.com/jimwebber
[37] https://github.com/jadell
[38] https://github.com/akollegger
[39] https://github.com/digitalstain
[40] https://github.com/tbaum
[41] https://github.com/apcj
[42] https://github.com/jexp
[43] https://github.com/jespernilsson
[44] https://github.com/tomsulston
[45] https://github.com/dmontag
[46] https://github.com/marlonrichert
[47] https://github.com/hugojosefson
[48] https://github.com/vivekprahlad

name	GIThub ID
Adriano Almeida	adrianoalmeida7[49]
Benjamin Gehrels	BGehrels[50]
Christopher Schmidt	FaKod[51]
Pascal Rehfeldt	prehfeldt[52]
Björn Söderqvist	cybear[53]
Abdul Azeez Shaik	abdulazeezsk[54]
James Thornton	espeed[55]
Radhakrishna Kalyan	nrkkalyan[56]
Michel van den Berg	promontis[57]
Brandon McCauslin	bm3780[58]
Hendy Irawan	ceefour[59]
Luanne Misquitta	luanne[60]
Jim Radford	radford[61]
Axel Morgner	amorgner[62]
Taylor Buley	buley[63]
Alex Smirnov	alexsmirnov[64]
Johannes Mockenhaupt	jotomo[65]
Pablo Pareja Tobes	pablopareja[66]
Björn Granvik	bjorngranvik[67]
Julian Simpson	simpsonjulian[68]
Pablo Pareja Tobes	pablopareja[69]
Rickard Öberg	rickardoberg[70]
Stefan Armbruster	sarmbruster[71]
Stephan Hagemann	shageman[72]

[49] https://github.com/adrianoalmeida7
[50] https://github.com/BGehrels
[51] https://github.com/FaKod
[52] https://github.com/prehfeldt
[53] https://github.com/cybear
[54] https://github.com/abdulazeezsk
[55] https://github.com/espeed
[56] https://github.com/nrkkalyan
[57] https://github.com/promontis
[58] https://github.com/bm3780
[59] https://github.com/ceefour
[60] https://github.com/luanne
[61] https://github.com/radford
[62] https://github.com/amorgner
[63] https://github.com/buley
[64] https://github.com/alexsmirnov
[65] https://github.com/jotomo
[66] https://github.com/pablopareja
[67] https://github.com/bjorngranvik
[68] https://github.com/simpsonjulian
[69] https://github.com/pablopareja
[70] https://github.com/rickardoberg
[71] https://github.com/sarmbruster
[72] https://github.com/shageman

name	GIThub ID
Linan Wang	wangii[73]
Ian Robinson	iansrobinson[74]
Marko Rodriguez	okram[75]
Saikat Kanjilal	skanjila[76]
Craig Taverner	craigtaverner[77]
David Winslow	dwins[78]
Patrick Fitzgerald	paddydub[79]
Stefan Berder	hrbonz[80]
Michael Kanner	SepiaGroup[81]
Lin Zhemin	miaoski[82]
Christophe Willemsen	kwattro[83]
Tony Liu	kooyeed[84]
Michael Klishin	michaelklishin[85]
Wes Freeman	wfreeman[86]
Chris Leishman	cleishm[87]
Brian Levine	blevine[88]
Ben Day	benday280412[89]
Davide Savazzi	svzdvd[90]
Nigel Small	nigelsmall[91]
Lasse Westh-Nielsen	lassewesth[92]
Wujek Srujek	wujek-srujek[93]
Alexander Yastrebov	AlexanderYastrebov[94]
Mike Bryant	mikesname[95]
Klaus Grossmann	iKlaus[96]

[73] https://github.com/wangii
[74] https://github.com/iansrobinson
[75] https://github.com/okram
[76] https://github.com/skanjila
[77] https://github.com/craigtaverner
[78] https://github.com/dwins
[79] https://github.com/paddydub
[80] https://github.com/hrbonz
[81] https://github.com/SepiaGroup
[82] https://github.com/miaoski
[83] https://github.com/kwattro
[84] https://github.com/kooyeed
[85] https://github.com/michaelklishin
[86] https://github.com/wfreeman
[87] https://github.com/cleishm
[88] https://github.com/blevine
[89] https://github.com/benday280412
[90] https://github.com/svzdvd
[91] https://github.com/nigelsmall
[92] https://github.com/lassewesth
[93] https://github.com/wujek-srujek
[94] https://github.com/AlexanderYastrebov
[95] https://github.com/mikesname
[96] https://github.com/iKlaus

name	GIThub ID
Pablo Lalloni	plalloni[97]
Stefan Plantikow	boggle[98]
Trenton Strong	trentonstrong[99]
Maciej Mazur	mamciek[100]
German Borbolla	germanborbolla[101]
Laurent Raufaste	lra[102]
Thomas Häfele	Perfect-Pixel[103]
Sevki Hasirci	Sevki[104]
Max De Marzi	maxdemarzi[105]
Pieter-Jan Van Aeken	PieterJanVanAeken[106]
Shane Gibbs	sgibbs-kellermed[107]
Yin Wang	yinwang0[108]
Volker Lanting	VolkerL[109]
Mark Needham	mneedham[110]
Chris Vest	chrisvest[111]
Ben Butler-Cole	benbc[112]
Tatham Oddie	tathamoddie[113]
Chris Skardon	cskardon[114]
Davide Grohmann	davidegrohmann[115]
Jakub Wieczorek	jakub-[116]
Magnus Vejlstrup	magnusvejlstrup[117]
Nikul Ukani	nikulukani[118]
Sebastian Wallin	wallin[119]
Dan Allen	mojavelinux[120]

[97] https://github.com/plalloni
[98] https://github.com/boggle
[99] https://github.com/trentonstrong
[100] https://github.com/mamciek
[101] https://github.com/germanborbolla
[102] https://github.com/lra
[103] https://github.com/Perfect-Pixel
[104] https://github.com/Sevki
[105] https://github.com/maxdemarzi
[106] https://github.com/PieterJanVanAeken
[107] https://github.com/sgibbs-kellermed
[108] https://github.com/yinwang0
[109] https://github.com/VolkerL
[110] https://github.com/mneedham
[111] https://github.com/chrisvest
[112] https://github.com/benbc
[113] https://github.com/tathamoddie
[114] https://github.com/cskardon
[115] https://github.com/davidegrohmann
[116] https://github.com/jakub-
[117] https://github.com/magnusvejlstrup
[118] https://github.com/nikulukani
[119] https://github.com/wallin
[120] https://github.com/mojavelinux

name	GIThub ID
Javad Karabi	karabijavad[121]
Georg Summer	gsummer[122]

[121] https://github.com/karabijavad
[122] https://github.com/gsummer

Part VIII. Advanced Usage

This part contains information on advanced usage of Neo4j. Among the topics covered are embedding Neo4j in your own software and writing plugins for the Neo4j Server.

You might want to keep the Neo4j JavaDocs[123] handy while reading!

[123] http://neo4j.com/docs/2.2.1/javadocs/

Chapter 34. Extending the Neo4j Server

The Neo4j Server can be extended by either plugins or unmanaged extensions.

34.1. 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[1] 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>2.2.1</version>
</dependency>
```

To create a plugin, your code must inherit from the ServerPlugin[2] class. Your plugin should also:

- ensure that it can produce an (Iterable of) `Node`, `Relationship` or `Path`, any Java primitive or `String` or an instance of a `org.neo4j.server.rest.repr.Representation`
- 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.

Note
If your plugin class has any constructors defined it must also have a no-arguments constructor defined.

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 )
```

[1] http://search.maven.org/#search|gav|1|g%3A%22org.neo4j%22%20AND%20a%3A%22server-api%22
[2] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/server/plugins/ServerPlugin.html

```
    public Iterable<Node> getAllNodes( @Source GraphDatabaseService graphDb )
    {
        ArrayList<Node> nodes = new ArrayList<>();
        try (Transaction tx = graphDb.beginTx())
        {
            for ( Node node : GlobalGraphOperations.at( graphDb ).getAllNodes() )
            {
                nodes.add( node );
            }
            tx.success();
        }
        return nodes;
    }

    @Description( "Get all relationships from the Neo4j graph database" )
    @PluginTarget( GraphDatabaseService.class )
    public Iterable<Relationship> getAllRelationships( @Source GraphDatabaseService graphDb )
    {
        List<Relationship> rels = new ArrayList<>();
        try (Transaction tx = graphDb.beginTx())
        {
            for ( Relationship rel : GlobalGraphOperations.at( graphDb ).getAllRelationships() )
            {
                rels.add( rel );
            }
            tx.success();
        }
        return rels;
    }
}
```

The full source code is found here: GetAll.java[3]

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 )
    {
        PathExpander<?> expander;
        List<Path> paths = new ArrayList<>();
        if ( types == null )
        {
            expander = PathExpanders.allTypesAndDirections();
        }
        else
        {
            PathExpanderBuilder expanderBuilder = PathExpanderBuilder.empty();
            for ( int i = 0; i < types.length; i++ )
            {
                expanderBuilder = expanderBuilder.add( DynamicRelationshipType.withName( types[i] ) );
            }
```

[3] https://github.com/neo4j/neo4j/blob/2.2.1/community/server-examples/src/main/java/org/neo4j/examples/server/plugins/GetAll.java

```
            expander = expanderBuilder.build();
        }
        try (Transaction tx = source.getGraphDatabase().beginTx())
        {
            PathFinder<Path> shortestPath = GraphAlgoFactory.shortestPath( expander,
                    depth == null ? 4 : depth.intValue() );
            for ( Path path : shortestPath.findAllPaths( source, target ) )
            {
                paths.add( path );
            }
            tx.success();
        }
        return paths;
    }
}
```

The full source code is found here: ShortestPath.java[4]

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).

Tip
Make sure the directories listings are retained in the jarfile by either building with default Maven, or with `jar -cvf myext.jar *`, making sure to jar directories instead of specifying single files.

The *.jar* file must include the file *META-INF/services/org.neo4j.server.plugins.ServerPlugin* with the fully qualified name of the implementation class. This is an example with multiple entries, each on a separate line:

```
org.neo4j.examples.server.plugins.DepthTwo
org.neo4j.examples.server.plugins.GetAll
org.neo4j.examples.server.plugins.ShortestPath
```

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" : {
```

[4] https://github.com/neo4j/neo4j/blob/2.2.1/community/server-examples/src/main/java/org/neo4j/examples/server/plugins/ShortestPath.java

```
  "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. For example 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/ShortestPath/node/123/shortestPath \
  -H "Content-Type: application/json" \
  -d '{"target":"http://localhost:7474/db/data/node/456", "depth":"5"}'
```

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 database operation will have to manage their own transactions, i.e. transactions aren't managed automatically. Note that the results of traversals or execution of graph algorithms should be exhausted inside the transaction before returning the result.

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).

34.2. Unmanaged Extensions

Sometimes you'll want finer grained control over your application's interactions with Neo4j than cypher provides. For these situations you can use the unmanaged extension API.

Caution

This is a sharp tool, allowing users to deploy arbitrary JAX-RS[5] classes to the server so be careful when using this. In particular it's easy to consume lots of heap space on the server and degrade performance. If in doubt please ask for help via one of the community channels (see Chapter 32, *Community Support* [547]).

Introduction to unmanaged extensions

The first step when writing an unmanaged extension is to create a project which includes dependencies to the JAX-RS and Neo4j core jars. In Maven this would be achieved by adding the following lines to the pom file:

```
<dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>javax.ws.rs-api</artifactId>
    <version>2.0</version>
    <scope>provided</scope>
</dependency>
```

```
<dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j</artifactId>
    <version>2.2.1</version>
    <scope>provided</scope>
</dependency>
```

Now we're ready to write our extension.

In our code we'll interact with the database using `GraphDatabaseService` which we can get access to by using the `@Context` annotation. The following examples serves as a template which you can base your extension on:

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( Charset.forName("UTF-8") ) ).build();
    }
}
```

[5] http://en.wikipedia.org/wiki/JAX-RS

The full source code is found here: HelloWorldResource.java[6]

Having built your code, the resulting jar file (and any custom dependencies) should be placed in the `$NEO4J_SERVER_HOME/plugins` directory. We also need to tell Neo4j where to look for the extension by adding some configuration in the *conf/neo4j-server.properties* file:

```
#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
```

Our hello method will now respond to `GET` requests at the URI: `http://{neo4j_server}:{neo4j_port}/examples/unmanaged/helloworld/{nodeId}`. e.g.

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

which results in

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

Streaming JSON responses

When writing unmanaged extensions we have greater control over the amount of memory that our Neo4j queries use. If we keep too much state around it can lead to more frequent full Garbage Collection and subsequent unresponsiveness by the Neo4j server.

A common way that state can creep in is the creation of JSON objects to represent the result of a query which we then send back to our application. Neo4j's Transactional Cypher HTTP endpoint (see Section 21.1, "Transactional Cypher HTTP endpoint" [286]) streams responses back to the client and we should follow in its footsteps.

For example, the following unmanaged extension streams an array of a person's colleagues:

Unmanaged extension streaming example

```
@Path("/colleagues")
public class ColleaguesResource
{
    private GraphDatabaseService graphDb;
    private final ObjectMapper objectMapper;

    private static final DynamicRelationshipType ACTED_IN = DynamicRelationshipType.withName( "ACTED_IN" );
    private static final Label PERSON = DynamicLabel.label( "Person" );

    public ColleaguesResource( @Context GraphDatabaseService graphDb )
    {
        this.graphDb = graphDb;
        this.objectMapper = new ObjectMapper();
    }

    @GET
    @Path("/{personName}")
    public Response findColleagues( final @PathParam("personName") String personName )
    {
        StreamingOutput stream = new StreamingOutput()
        {
            @Override
            public void write( OutputStream os ) throws IOException, WebApplicationException
            {
                JsonGenerator jg = objectMapper.getJsonFactory().createJsonGenerator( os, JsonEncoding.UTF8 );
                jg.writeStartObject();
                jg.writeFieldName( "colleagues" );
                jg.writeStartArray();
```

[6] https://github.com/neo4j/neo4j/blob/2.2.1/community/server-examples/src/main/java/org/neo4j/examples/server/unmanaged/HelloWorldResource.java

```
                try ( Transaction tx = graphDb.beginTx();
                      ResourceIterator<Node> persons = graphDb.findNodes( PERSON, "name", personName ) )
                {
                    while ( persons.hasNext() )
                    {
                        Node person = persons.next();
                        for ( Relationship actedIn : person.getRelationships( ACTED_IN, OUTGOING ) )
                        {
                            Node endNode = actedIn.getEndNode();
                            for ( Relationship colleagueActedIn : endNode.getRelationships( ACTED_IN, INCOMING ) )
                            {
                                Node colleague = colleagueActedIn.getStartNode();
                                if ( !colleague.equals( person ) )
                                {
                                    jg.writeString( colleague.getProperty( "name" ).toString() );
                                }
                            }
                        }
                    }
                    tx.success();
                }

                jg.writeEndArray();
                jg.writeEndObject();
                jg.flush();
                jg.close();
            }
        };

        return Response.ok().entity( stream ).type( MediaType.APPLICATION_JSON ).build();
    }
}
```

The full source code is found here: ColleaguesResource.java[7]

As well as depending on JAX-RS API this example also uses Jackson — a Java JSON library. You'll need to add the following dependency to your Maven POM file (or equivalent):

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.9.7</version>
</dependency>
```

Our `findColleagues` method will now respond to `GET` requests at the URI: `http://{neo4j_server}:{neo4j_port}/examples/unmanaged/colleagues/{personName}`. For example:

```
curl http://localhost:7474/examples/unmanaged/colleagues/Keanu%20Reeves
```

which results in

```
{"colleagues":["Hugo Weaving","Carrie-Anne Moss","Laurence Fishburne"]}
```

Using Cypher in an unmanaged extension

You can execute Cypher queries by using the `GraphDatabaseService` that is injected into the extension.

Note

In Neo4j versions prior to 2.2 you had to retrieve an `ExecutionEngine` to execute queries. This has been deprecated, and we recommend you to update any existing code to use `GraphDatabaseService` instead.

[7] https://github.com/neo4j/neo4j/blob/2.2.1/community/server-examples/src/main/java/org/neo4j/examples/server/unmanaged/ColleaguesResource.java

For example, the following unmanaged extension retrieves a person's colleagues using Cypher:

Unmanaged extension Cypher execution example

```
@Path("/colleagues-cypher-execution")
public class ColleaguesCypherExecutionResource
{
    private final ObjectMapper objectMapper;
    private GraphDatabaseService graphDb;

    public ColleaguesCypherExecutionResource( @Context GraphDatabaseService graphDb )
    {
        this.graphDb = graphDb;
        this.objectMapper = new ObjectMapper();
    }

    @GET
    @Path("/{personName}")
    public Response findColleagues( final @PathParam("personName") String personName )
    {
        final Map<String, Object> params = MapUtil.map( "personName", personName );

        StreamingOutput stream = new StreamingOutput()
        {
            @Override
            public void write( OutputStream os ) throws IOException, WebApplicationException
            {
                JsonGenerator jg = objectMapper.getJsonFactory().createJsonGenerator( os, JsonEncoding.UTF8 );
                jg.writeStartObject();
                jg.writeFieldName( "colleagues" );
                jg.writeStartArray();

                try ( Transaction tx = graphDb.beginTx();
                      Result result = graphDb.execute( colleaguesQuery(), params ) )
                {
                    while ( result.hasNext() )
                    {
                        Map<String,Object> row = result.next();
                        jg.writeString( ((Node) row.get( "colleague" )).getProperty( "name" ).toString() );
                    }
                    tx.success();
                }

                jg.writeEndArray();
                jg.writeEndObject();
                jg.flush();
                jg.close();
            }
        };

        return Response.ok().entity( stream ).type( MediaType.APPLICATION_JSON ).build();
    }

    private String colleaguesQuery()
    {
        return "MATCH (p:Person {name: {personName} })-[:ACTED_IN]->()<-[:ACTED_IN]-(colleague) RETURN colleague";
    }
}
```

The full source code is found here: ColleaguesCypherExecutionResource.java[8]

[8] https://github.com/neo4j/neo4j/blob/2.2.1/community/server-examples/src/main/java/org/neo4j/examples/server/unmanaged/ColleaguesCypherExecutionResource.java

Our `findColleagues` method will now respond to `GET` requests at the URI: `http://{neo4j_server}:{neo4j_port}/examples/unmanaged/colleagues-cypher-execution/{personName}`. e.g.

```
curl http://localhost:7474/examples/unmanaged/colleagues-cypher-execution/Keanu%20Reeves
```

which results in

```
{"colleagues":["Hugo Weaving","Carrie-Anne Moss","Laurence Fishburne"]}
```

34.3. Testing your extension

Neo4j provides tools to help you write integration tests for your extensions. You can access this toolkit by adding the following test dependency to your project:

```
<dependency>
   <groupId>org.neo4j.test</groupId>
   <artifactId>neo4j-harness</artifactId>
   <version>2.2.1</version>
   <scope>test</scope>
</dependency>
```

The test toolkit provides a mechanism to start a Neo4j instance with custom configuration and with extensions of your choice. It also provides mechanisms to specify data fixtures to include when starting Neo4j.

Usage example

```
@Path("")
public static class MyUnmanagedExtension
{
    @GET
    public Response myEndpoint()
    {
        return Response.ok().build();
    }
}

@Test
public void testMyExtension() throws Exception
{
    // Given
    try ( ServerControls server = TestServerBuilders.newInProcessBuilder()
            .withExtension( "/myExtension", MyUnmanagedExtension.class )
            .newServer() )
    {
        // When
        HTTP.Response response = HTTP.GET( server.httpURI().resolve( "myExtension" ).toString() );

        // Then
        assertEquals( 200, response.status() );
    }
}
```

The full source code of the example is found here: ExtensionTestingDocTest.java[9]

If you are using the JUnit test framework, there is a JUnit rule available as well. Note the use of `server.httpURI().resolve( "myExtension" )` to ensure that the correct base URI is used.

JUnit example

```
@Rule
public Neo4jRule neo4j = new Neo4jRule()
        .withFixture( "CREATE (admin:Admin)" );

@Test
public void shouldWorkWithServer() throws Exception
{
    // Given
    URI serverURI = neo4j.httpURI();
```

[9] https://github.com/neo4j/neo4j/blob/2.2.1/community/neo4j-harness/src/test/java/org/neo4j/harness/doc/ExtensionTestingDocTest.java

```
    // When I access the server
    HTTP.Response response = HTTP.GET( serverURI.toString() );

    // Then it should reply
    assertEquals(200, response.status());
}
```

34.4. Installing Plugins and Extensions in Neo4j Desktop

Neo4j Desktop can also be extended with server plugins and extensions. Neo4j Desktop will add all jars in *%ProgramFiles%\Neo4j Community\plugins* to the classpath, but please note that nested directories for plugins are currently not supported.

Otherwise server plugins and extensions are subject to the same rules as usual. Please note when configuring server extensions that *neo4j-server.properties* for Neo4j Desktop lives in *%APPDATA%\Neo4j Community*.

Chapter 35. Using Neo4j embedded in Java applications

It’s easy to use Neo4j embedded in Java applications. In this chapter you will find everything needed — from setting up the environment to doing something useful with your data.

35.1. 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.

Add Neo4j to the build path

Get the Neo4j libraries from one of these sources:

- Extract a Neo4j download[1] zip/tarball, and use the *jar* files found in the *lib/* directory.
- Use the *jar* files available from Maven Central Repository[2]

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[3].
IntelliJ IDEA	See Libraries, Global Libraries, and the Configure Library dialog[4]
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[5].

Editions

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, Groovy Grape, Grails, Scala SBT!

Neo4j editions

Edition	Dependency	Description	License
Community	org.neo4j:neo4j[6]	a high performance, fully ACID transactional graph database	GPLv3
Enterprise	org.neo4j:neo4j-enterprise[7]	adding advanced monitoring, online backup and High Availability clustering	AGPLv3

[1] http://neo4j.com/download/
[2] http://search.maven.org/#search|ga|1|g%3A%22org.neo4j%22
[3] http://help.eclipse.org/indigo/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/preferences/java/buildpath/ref-preferences-user-libraries.htm
[4] http://www.jetbrains.com/idea/webhelp/configuring-project-and-global-libraries.html
[5] http://netbeans.org/kb/docs/java/project-setup.html#projects-classpath
[6] http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j%22%20AND%20a%3A%22neo4j%22
[7] http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-enterprise%22

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

For more information regarding licensing, see the Licensing Guide[8].

Javadocs can be downloaded packaged in jar files from Maven Central or read at javadocs[9].

Add Neo4j as a dependency

You can either go with the top-level artifact from the table above or include the individual components directly. The examples included here use the top-level artifact approach.

Maven

Add the dependency to your project along the lines of the snippet below. This is usually done in the `pom.xml` file found in the root directory of the project.

Maven dependency

```
<project>
...
 <dependencies>
  <dependency>
   <groupId>org.neo4j</groupId>
   <artifactId>neo4j</artifactId>
   <version>2.2.1</version>
  </dependency>
  ...
 </dependencies>
...
</project>
```

Where the `artifactId` is found in the editions table.

Eclipse and Maven

For development in Eclipse[10], it is recommended to install the m2e plugin[11] 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:

---

[8] http://www.neo4j.org/learn/licensing
[9] http://neo4j.com/docs/2.2.1/javadocs/
[10] http://www.eclipse.org
[11] http://www.eclipse.org/m2e/

```
..
<dependencies>
  ..
  <dependency org="org.neo4j" name="neo4j" rev="2.2.1"/>
  ..
</dependencies>
..
```

Where the `name` *is found in the editions table above*

Gradle

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

```
def neo4jVersion = "2.2.1"
apply plugin: 'java'
repositories {
   mavenCentral()
}
dependencies {
   compile "org.neo4j:neo4j:${neo4jVersion}"
}
```

Where the coordinates (`org.neo4j:neo4j` *in the example) are found in the editions table above.*

Starting and stopping

To create a new database or ópen an existing one you instantiate a `GraphDatabaseService`[12].

```
graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
registerShutdownHook( graphDb );
```

Note

The `GraphDatabaseService` 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 application).
    Runtime.getRuntime().addShutdownHook( new Thread()
    {
        @Override
        public void run()
        {
            graphDb.shutdown();
        }
    } );
}
```

Starting an embedded database with configuration settings

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

```
GraphDatabaseService graphDb = new GraphDatabaseFactory()
```

[12] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/GraphDatabaseService.html

```
    .newEmbeddedDatabaseBuilder( storeDir )
    .loadPropertiesFromFile( pathToConfig + "neo4j.properties" )
    .newGraphDatabase();
```

Configuration settings can also be applied programmatically, like so:

```
GraphDatabaseService graphDb = new GraphDatabaseFactory()
    .newEmbeddedDatabaseBuilder( storeDir )
    .setConfig( GraphDatabaseSettings.pagecache_memory, "512M" )
    .setConfig( GraphDatabaseSettings.string_block_size, "60" )
    .setConfig( GraphDatabaseSettings.array_block_size, "300" )
    .newGraphDatabase();
```

For configuration settings, see Chapter 24, *Configuration & Performance* [435].

Starting an embedded read-only instance

If you want a *read-only view* of the database, create an instance this way:

```
graphDb = new GraphDatabaseFactory().newEmbeddedDatabaseBuilder(
        "target/read-only-db/location" )
                                    .setConfig( GraphDatabaseSettings.read_only, "true" )
                                    .newGraphDatabase();
```

Obviously the database has to already exist in this case.

Note
Concurrent access to the same database files by multiple (read-only or write) instances is not supported.

35.2. Hello World

Learn how to create and access nodes and relationships. For information on project setup, see Section 35.1, “Include Neo4j in your project” [590].

Remember, from Section 2.1, “The Neo4j Graph Database” [5], that a Neo4j graph consists 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.

Tip
The source code of this example is found here: EmbeddedNeo4j.java[13]

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
}
```

We also prepare some variables to use:

```
GraphDatabaseService graphDb;
Node firstNode;
Node secondNode;
Relationship relationship;
```

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.

```
graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
registerShutdownHook( graphDb );
```

Note that starting a database 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.

Wrap operations in a transaction

All operations 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:

[13] https://github.com/neo4j/neo4j/blob/2.2.1/community/embedded-examples/src/main/java/org/neo4j/examples/EmbeddedNeo4j.java

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

For more information on transactions, see Chapter 18, *Transaction Management* [272] and Java API for Transaction[14].

> **Note**
> For brevity, we do not spell out wrapping of operations in a transaction throughout the manual.

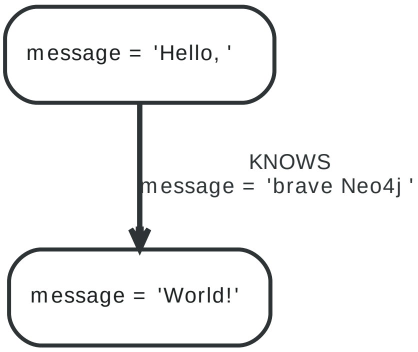
Create a small graph

Now, let's create a few nodes. The API is very intuitive. Feel free to have a look at the Neo4j Javadocs[15]. 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:

```
firstNode = graphDb.createNode();
firstNode.setProperty( "message", "Hello, " );
secondNode = graphDb.createNode();
secondNode.setProperty( "message", "World!" );

relationship = firstNode.createRelationshipTo( secondNode, RelTypes.KNOWS );
relationship.setProperty( "message", "brave Neo4j " );
```

We now have a graph that looks like this:

Figure 35.1. Hello World Graph



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!

Remove the data

In this case we'll remove the data before committing:

```
// let's remove the data
```

[14] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/Transaction.html
[15] http://neo4j.com/docs/2.2.1/javadocs/

```
firstNode.getSingleRelationship( RelTypes.KNOWS, Direction.OUTGOING ).delete();
firstNode.delete();
secondNode.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.

Shut down the database server

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

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

35.3. Property values

Both nodes and relationships can have properties.

Properties are named values where the name 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.

`NULL` **is not a valid property value.**
`NULL`s 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	
`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[16].

[16] http://docs.oracle.com/javase/specs/jls/se5.0/html/typesValues.html#4.2.3

35.4. User database with indexes

You have a user database, and want to retrieve users by name using indexes.

Tip
The source code used in this example is found here:
EmbeddedNeo4jWithNewIndexing.java[17]

To begin with, we start the database server:

```
GraphDatabaseService graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
```

Then we have to configure the database to index users by name. This only needs to be done once.

```
IndexDefinition indexDefinition;
try ( Transaction tx = graphDb.beginTx() )
{
    Schema schema = graphDb.schema();
    indexDefinition = schema.indexFor( DynamicLabel.label( "User" ) )
            .on( "username" )
            .create();
    tx.success();
}
```

Indexes are populated asynchronously when they are first created. It is possible to use the core API to wait for index population to complete:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Schema schema = graphDb.schema();
    schema.awaitIndexOnline( indexDefinition, 10, TimeUnit.SECONDS );
}
```

It's time to add the users:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = DynamicLabel.label( "User" );

    // Create some users
    for ( int id = 0; id < 100; id++ )
    {
        Node userNode = graphDb.createNode( label );
        userNode.setProperty( "username", "user" + id + "@neo4j.org" );
    }
    System.out.println( "Users created" );
    tx.success();
}
```

Note
Please read Section 35.6, "Managing resources when using long running transactions" [601] on how to properly close `ResourceIterators` returned from index lookups.

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

```
Label label = DynamicLabel.label( "User" );
```

[17] https://github.com/neo4j/neo4j/blob/2.2.1/community/embedded-examples/src/main/java/org/neo4j/examples/EmbeddedNeo4jWithNewIndexing.java

```
int idToFind = 45;
String nameToFind = "user" + idToFind + "@neo4j.org";
try ( Transaction tx = graphDb.beginTx() )
{
    try ( ResourceIterator<Node> users =
            graphDb.findNodes( label, "username", nameToFind ) )
    {
        ArrayList<Node> userNodes = new ArrayList<>();
        while ( users.hasNext() )
        {
            userNodes.add( users.next() );
        }

        for ( Node node : userNodes )
        {
            System.out.println( "The username of user " + idToFind + " is " + node.getProperty( "username" ) );
        }
    }
}
```

When updating the name of a user, the index is updated as well:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = DynamicLabel.label( "User" );
    int idToFind = 45;
    String nameToFind = "user" + idToFind + "@neo4j.org";

    for ( Node node : loop( graphDb.findNodes( label, "username", nameToFind ) ) )
    {
        node.setProperty( "username", "user" + ( idToFind + 1 ) + "@neo4j.org" );
    }
    tx.success();
}
```

When deleting a user, it is automatically removed from the index:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = DynamicLabel.label( "User" );
    int idToFind = 46;
    String nameToFind = "user" + idToFind + "@neo4j.org";

    for ( Node node : loop( graphDb.findNodes( label, "username", nameToFind ) ) )
    {
        node.delete();
    }
    tx.success();
}
```

In case we change our data model, we can drop the index as well:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = DynamicLabel.label( "User" );
    for ( IndexDefinition indexDefinition : graphDb.schema()
            .getIndexes( label ) )
    {
        // There is only one index
        indexDefinition.drop();
    }

    tx.success();
}
```

35.5. User database with legacy index

Unless you have specific reasons to use the legacy indexing, see Section 35.4, "User database with indexes" [598] instead.

Note

Please read Section 35.6, "Managing resources when using long running transactions" [601] on how to properly close `ResourceIterators` returned from index lookups.

You have a user database, and want to retrieve users by name using the legacy indexing system.

Tip

The source code used in this example is found here: EmbeddedNeo4jWithIndexing.java[18]

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 GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
registerShutdownHook();
```

It's time to add the users:

```
try ( Transaction tx = graphDb.beginTx() )
{
    nodeIndex = graphDb.index().forNodes( "nodes" );
    // Create some users and index their names with the IndexService
    for ( int id = 0; id < 100; id++ )
    {
        createAndIndexUser( idToUserName( id ) );
    }
```

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

```
int idToFind = 45;
String userName = idToUserName( idToFind );
Node foundUser = nodeIndex.get( USERNAME_KEY, userName ).getSingle();

System.out.println( "The username of user " + idToFind + " is "
    + foundUser.getProperty( USERNAME_KEY ) );
```

[18] https://github.com/neo4j/neo4j/blob/2.2.1/community/embedded-examples/src/main/java/org/neo4j/examples/EmbeddedNeo4jWithIndexing.java

35.6. Managing resources when using long running transactions

It is necessary to always open a transaction when accessing the database. Inside a long running transaction it is good practice to ensure that any `ResourceIterator`[19]s obtained inside the transaction are closed as early as possible. This is either achieved by just exhausting the iterator or by explicitly calling its close method.

What follows is an example of how to work with a `ResourceIterator`. As we don't exhaust the iterator, we will close it explicitly using the `close()` method.

```
Label label = DynamicLabel.label( "User" );
int idToFind = 45;
String nameToFind = "user" + idToFind + "@neo4j.org";
try ( Transaction tx = graphDb.beginTx();
      ResourceIterator<Node> users = graphDb.findNodes( label, "username", nameToFind ) )
{
    Node firstUserNode;
    if ( users.hasNext() )
    {
        firstUserNode = users.next();
    }
    users.close();
}
```

[19] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/ResourceIterator.html

35.7. Basic unit testing

The basic pattern of unit testing with Neo4j is illustrated by the following example.

To access the Neo4j testing facilities you should have the `neo4j-kernel` *tests.jar* together with the `neo4j-io` *tests.jar* on the classpath during tests. You can download them from Maven Central: org.neo4j:neo4j-kernel[20] and org.neo4j:neo4j-io[21].

Using Maven as a dependency manager you would typically add this dependency together with JUnit and Hamcrest like so:

Maven dependency

```
<project>
...
 <dependencies>
  <dependency>
   <groupId>org.neo4j</groupId>
   <artifactId>neo4j-kernel</artifactId>
   <version>2.2.1</version>
   <type>test-jar</type>
   <scope>test</scope>
  </dependency>
  <dependency>
   <groupId>org.neo4j</groupId>
   <artifactId>neo4j-io</artifactId>
   <version>2.2.1</version>
   <type>test-jar</type>
   <scope>test</scope>
  </dependency>
  <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.12</version>
   <scope>test</scope>
  </dependency>
  <dependency>
   <groupId>org.hamcrest</groupId>
   <artifactId>hamcrest-all</artifactId>
   <version>1.3</version>
   <scope>test</scope>
  </dependency>
  ...
 </dependencies>
...
</project>
```

Observe that the `<type>test-jar</type>` is crucial. Without it you would get the common `neo4j-kernel` jar, not the one containing the testing facilities.

With that in place, we're ready to code our tests.

> **Tip**
> For the full source code of this example see: Neo4jBasicDocTest.java[22]

Before each test, create a fresh database:

[20] http://search.maven.org/#search|ga|1|g%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-kernel%22
[21] http://search.maven.org/#search|ga|1|g%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-io%22
[22] https://github.com/neo4j/neo4j/blob/2.2.1/community/embedded-examples/src/test/java/org/neo4j/examples/Neo4jBasicDocTest.java

```
@Before
public void prepareTestDatabase()
{
    graphDb = new TestGraphDatabaseFactory().newImpermanentDatabase();
}
```

After the test has executed, the database should be shut down:

```
@After
public void destroyTestDatabase()
{
    graphDb.shutdown();
}
```

During a test, create nodes and check to see that they are there, while enclosing write operations in a transaction.

```
Node n = null;
try ( Transaction tx = graphDb.beginTx() )
{
    n = graphDb.createNode();
    n.setProperty( "name", "Nancy" );
    tx.success();
}

// The node should have a valid id
assertThat( n.getId(), is( greaterThan( -1L ) ) );

// Retrieve a node by using the id of the created node. The id's and
// property should match.
try ( Transaction tx = graphDb.beginTx() )
{
    Node foundNode = graphDb.getNodeById( n.getId() );
    assertThat( foundNode.getId(), is( n.getId() ) );
    assertThat( (String) foundNode.getProperty( "name" ), is( "Nancy" ) );
}
```

If you want to set configuration parameters at database creation, it's done like this:

```
GraphDatabaseService db = new TestGraphDatabaseFactory()
    .newImpermanentDatabaseBuilder()
    .setConfig( GraphDatabaseSettings.pagecache_memory, "512M" )
    .setConfig( GraphDatabaseSettings.string_block_size, "60" )
    .setConfig( GraphDatabaseSettings.array_block_size, "300" )
    .newGraphDatabase();
```

35.8. Traversal

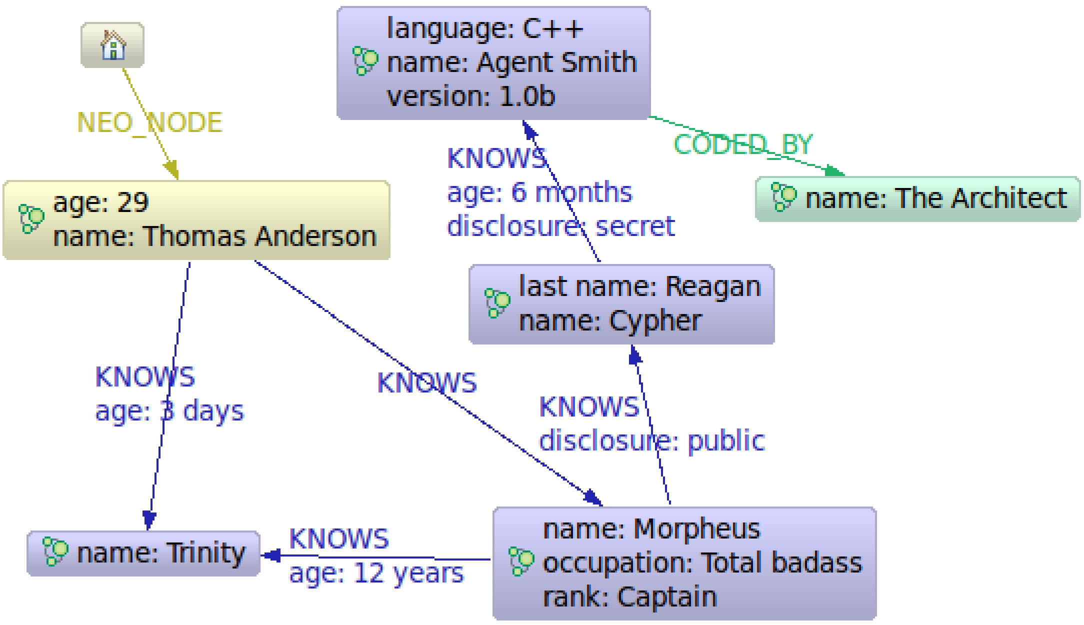
For reading about traversals, see Chapter 36, *The Traversal Framework* [624].

For more examples of traversals, see Chapter 5, *Basic Data Modeling Examples* [45].

The Matrix

This is the first graph we want to traverse into:

Figure 35.2. Matrix node space view



Tip

The source code of this example is found here: NewMatrix.java[23]

Friends and friends of friends

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

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

```
int numberOfFriends = 0;
String output = neoNode.getProperty( "name" ) + "'s friends:\n";
Traverser friendsTraverser = getFriends( neoNode );
for ( Path friendPath : friendsTraverser )
{
    output += "At depth " + friendPath.length() + " => "
              + friendPath.endNode()
```

[23] https://github.com/neo4j/neo4j/blob/2.2.1/community/embedded-examples/src/main/java/org/neo4j/examples/NewMatrix.java

```
                      .getProperty( "name" ) + "\n";
    numberOfFriends++;
}
output += "Number of friends found: " + numberOfFriends + "\n";
```

Which will give us the following output:

```
Thomas Anderson's friends:
At depth 1 => Morpheus
At depth 1 => Trinity
At depth 2 => Cypher
At depth 3 => Agent Smith
Number of friends found: 4
```

Who coded the Matrix?

```
private Traverser findHackers( final Node startNode )
{
    TraversalDescription td = graphDb.traversalDescription()
            .breadthFirst()
            .relationships( RelTypes.CODED_BY, Direction.OUTGOING )
            .relationships( RelTypes.KNOWS, Direction.OUTGOING )
            .evaluator(
                    Evaluators.includeWhereLastRelationshipTypeIs( RelTypes.CODED_BY ) );
    return td.traverse( startNode );
}
```

Print out the result:

```
String output = "Hackers:\n";
int numberOfHackers = 0;
Traverser traverser = findHackers( getNeoNode() );
for ( Path hackerPath : traverser )
{
    output += "At depth " + hackerPath.length() + " => "
              + hackerPath.endNode()
                      .getProperty( "name" ) + "\n";
    numberOfHackers++;
}
output += "Number of hackers found: " + numberOfHackers + "\n";
```

Now we know who coded the Matrix:

```
Hackers:
At depth 4 => The Architect
Number of hackers found: 1
```

Walking an ordered path

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

Tip

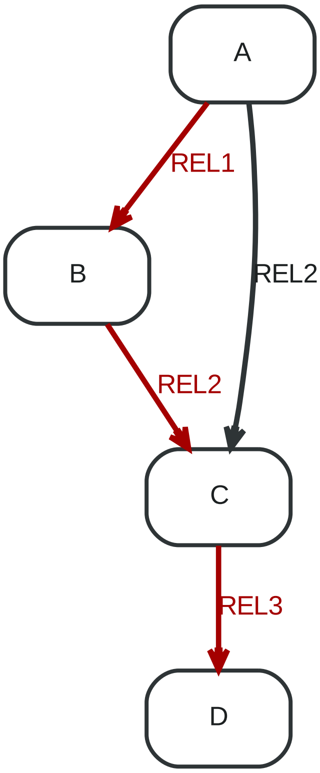
The source code of this example is found here: OrderedPath.java[24]

Create a toy graph

```
Node A = db.createNode();
Node B = db.createNode();
Node C = db.createNode();
```

[24] https://github.com/neo4j/neo4j/blob/2.2.1/community/embedded-examples/src/main/java/org/neo4j/examples/orderedpath/OrderedPath.java

```
Node D = db.createNode();

A.createRelationshipTo( C, REL2 );
C.createRelationshipTo( D, REL3 );
A.createRelationshipTo( B, REL1 );
B.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:

Define how to walk the path

```
final ArrayList<RelationshipType> orderedPathContext = new ArrayList<RelationshipType>();
orderedPathContext.add( REL1 );
orderedPathContext.add( withName( "REL2" ) );
orderedPathContext.add( withName( "REL3" ) );
TraversalDescription td = db.traversalDescription()
        .evaluator( new Evaluator()
        {
            @Override
            public Evaluation evaluate( final Path path )
            {
                if ( path.length() == 0 )
                {
                    return Evaluation.EXCLUDE_AND_CONTINUE;
                }
                RelationshipType expectedType = orderedPathContext.get( path.length() - 1 );
                boolean isExpectedType = path.lastRelationship()
                        .isType( expectedType );
                boolean included = path.length() == orderedPathContext.size() && isExpectedType;
                boolean continued = path.length() < orderedPathContext.size() && isExpectedType;
                return Evaluation.of( included, continued );
            }
        } )
        .uniqueness( Uniqueness.NODE_PATH );
```

Note that we set the uniqueness to `Uniqueness.NODE_PATH`[25] as we want to be able to revisit the same node dureing the traversal, but not the same path.

Perform the traversal and print the result

```
Traverser traverser = td.traverse( A );
PathPrinter pathPrinter = new PathPrinter( "name" );
for ( Path path : traverser )
{
    output += Paths.pathToString( path, pathPrinter );
}
```

Which will output:

```
(A)--[REL1]-->(B)--[REL2]-->(C)--[REL3]-->(D)
```

In this case we use a custom class to format the path output. This is how it’s done:

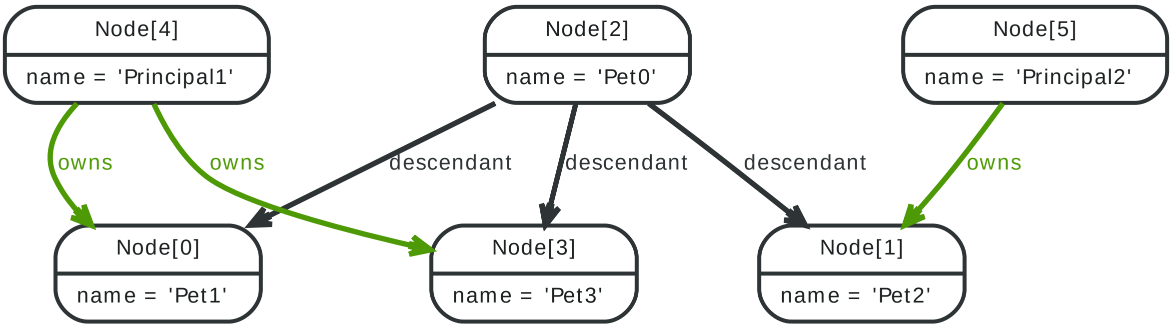
```
static class PathPrinter implements Paths.PathDescriptor<Path>
{
    private final String nodePropertyKey;

    public PathPrinter( String nodePropertyKey )
    {
        this.nodePropertyKey = nodePropertyKey;
    }

    @Override
    public String nodeRepresentation( Path path, Node node )
    {
        return "(" + node.getProperty( nodePropertyKey, "" ) + ")";
    }

    @Override
    public String relationshipRepresentation( Path path, Node from, Relationship relationship )
    {
        String prefix = "--", suffix = "--";
        if ( from.equals( relationship.getEndNode() ) )
        {
            prefix = "<--";
        }
        else
        {
            suffix = "-->";
        }
        return prefix + "[" + relationship.getType().name() + "]" + suffix;
    }
}
```

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.

[25] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/Uniqueness.html#NODE_PATH

Figure 35.3. 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 = db.traversalDescription()
        .uniqueness( Uniqueness.NODE_PATH )
        .evaluator( new Evaluator()
{
    @Override
    public Evaluation evaluate( Path path )
    {
        boolean endNodeIsTarget = path.endNode().equals( target );
        return Evaluation.of( endNodeIsTarget, !endNodeIsTarget );
    }
} );

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

This will return the following paths:

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

In the default `path.toString()` implementation, `(1)--[knows,2]-->(4)` denotes a node with ID=1 having a relationship with ID 2 or type `knows` to a node with ID-4.

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:

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

Social network

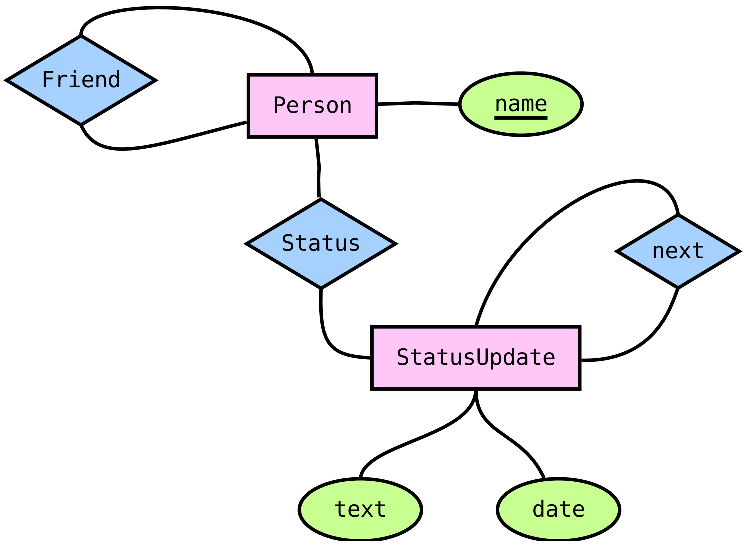
Note
The following example uses the new enhanced traversal API.

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.

Tip
The source code of the example is found here: socnet[26]

Simple social model

Figure 35.4. 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.

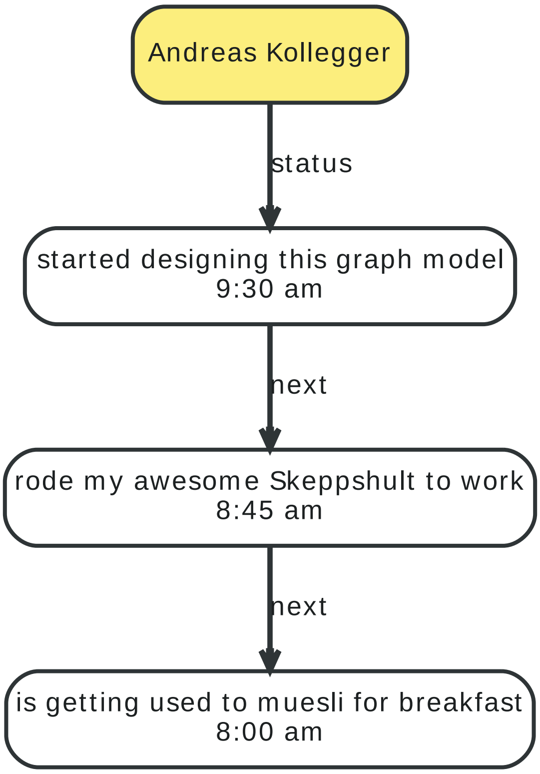
- `Person`
  - `friend`: relates two distinct `Person` instances (no self-reference)
  - `status`: connects to the most recent `StatusUpdate`
- `StatusUpdate`
  - `next`: points to the next `StatusUpdate` in the chain, which was posted before the current one

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:

[26] https://github.com/neo4j/neo4j/tree/2.2.1/community/embedded-examples/src/main/java/org/neo4j/examples/socnet



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

```
TraversalDescription traversal = graphDb().traversalDescription()
        .depthFirst()
        .relationships( NEXT );
```

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[27].

The code looks like:

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

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

[27] http://www.slideshare.net/systay/pattern-activity-stream

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

return returnVal;
```

35.9. 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).

Tip

The source code of the examples is found here: Person.java[28]

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() + "]";
}
```

[28] https://github.com/neo4j/neo4j/blob/2.2.1/community/embedded-examples/src/main/java/org/neo4j/examples/socnet/Person.java

35.10. Graph Algorithm examples

Tip

The source code used in the example is found here: PathFindingDocTest.java[29]

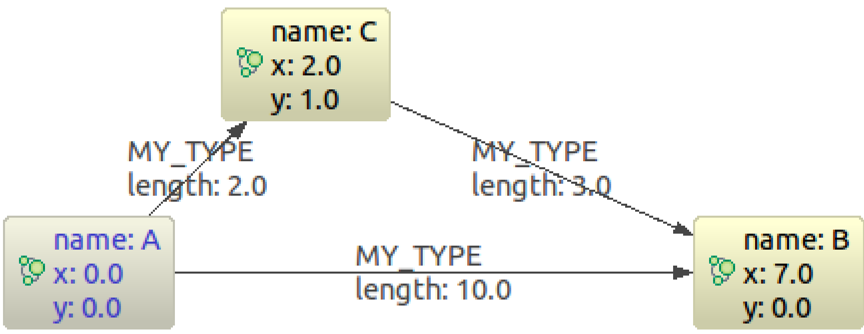
Calculating the shortest path (least number of relationships) between two nodes:

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

Using Dijkstra's algorithm[30] 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(
    PathExpanders.forTypeAndDirection( ExampleTypes.MY_TYPE, Direction.BOTH ), "cost" );

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

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

Using A*[31] 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 );
```

[29] https://github.com/neo4j/neo4j/blob/2.2.1/community/embedded-examples/src/test/java/org/neo4j/examples/PathFindingDocTest.java
[30] http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
[31] http://en.wikipedia.org/wiki/A*_search_algorithm

```
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>()
{
    @Override
    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(
        PathExpanders.allTypesAndDirections(),
        CommonEvaluators.doubleCostEvaluator( "length" ), estimateEvaluator );
WeightedPath path = astar.findSinglePath( nodeA, nodeB );
```

35.11. Reading a management attribute

The JmxUtils[32] class includes methods to access 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.

Tip
The source code of the example is found here: JmxDocTest.java[33]

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

```
private static Date getStartTimeFromManagementBean(
        GraphDatabaseService graphDbService )
{
    ObjectName objectName = JmxUtils.getObjectName( graphDbService, "Kernel" );
    Date date = JmxUtils.getAttribute( objectName, "KernelStartTime" );
    return date;
}
```

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

- For all editions, see the org.neo4j.jmx[34] package.
- For the Enterprise edition, see the org.neo4j.management[35] package as well.

[32] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/jmx/JmxUtils.html
[33] https://github.com/neo4j/neo4j/blob/2.2.1/community/embedded-examples/src/test/java/org/neo4j/examples/JmxDocTest.java
[34] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/jmx/package-summary.html
[35] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/management/package-summary.html

35.12. How to create unique nodes

This section is about how to ensure uniqueness of a property when creating nodes. For an overview of the topic, see Section 18.6, "Creating unique nodes" [280].

Get or create unique node using Cypher and unique constraints

Create a unique constraint

```
try ( Transaction tx = graphdb.beginTx() )
{
    graphdb.schema()
            .constraintFor( DynamicLabel.label( "User" ) )
            .assertPropertyIsUnique( "name" )
            .create();
    tx.success();
}
```

Use MERGE to create a unique node

```
Node result = null;
ResourceIterator<Node> resultIterator = null;
try ( Transaction tx = graphDb.beginTx() )
{
    String queryString = "MERGE (n:User {name: {name}}) RETURN n";
    Map<String, Object> parameters = new HashMap<>();
    parameters.put( "name", username );
    resultIterator = graphDb.execute( queryString, parameters ).columnAs( "n" );
    result = resultIterator.next();
    tx.success();
    return result;
}
```

Get or create unique node using a legacy index

Important
While this is a working solution, please consider using the preferred solution at the section called "Get or create unique node using Cypher and unique constraints" [616] instead.

By using `put-if-absent`[36] functionality, entity uniqueness can be guaranteed using an index.

Here the index acts as the lock and will only lock the smallest part needed to guarantee uniqueness across threads and transactions. To get the more high-level `get-or-create` functionality make use of `UniqueFactory`[37] as seen in the example below.

Create a factory for unique nodes at application start

```
try ( Transaction tx = graphDb.beginTx() )
{
    UniqueFactory.UniqueNodeFactory result = new UniqueFactory.UniqueNodeFactory( graphDb, "users" )
    {
        @Override
        protected void initialize( Node created, Map<String, Object> properties )
        {
            created.addLabel( DynamicLabel.label( "User" ) );
            created.setProperty( "name", properties.get( "name" ) );
        }
    };
    tx.success();
```

[36] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/Index.html#putIfAbsent%28T,%20java.lang.String,%20java.lang.Object%29
[37] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/UniqueFactory.html

```
    return result;
}
```

Use the unique node factory to get or create a node

```
try ( Transaction tx = graphDb.beginTx() )
{
    Node node = factory.getOrCreate( "name", username );
    tx.success();
    return node;
}
```

Pessimistic locking for node creation

> **Important**
> While this is a working solution, please consider using the preferred solution at the section called “Get or create unique node using Cypher and unique constraints” [616] instead.

One might be tempted to use Java synchronization for pessimistic locking, but this is dangerous. By mixing locks in Neo4j and in the Java runtime, it is easy to produce deadlocks that are not detectable by Neo4j. As long as all locking is done by Neo4j, all deadlocks will be detected and avoided. Also, a solution using manual synchronization doesn’t ensure uniqueness in an HA environment.

This example uses a single “lock node” for locking. We create it only as a place to put locks, nothing else.

Create a lock node at application start

```
try ( Transaction tx = graphDb.beginTx() )
{
    final Node lockNode = graphDb.createNode();
    tx.success();
    return lockNode;
}
```

Use the lock node to ensure nodes are not created concurrently

```
try ( Transaction tx = graphDb.beginTx() )
{
    Index<Node> usersIndex = graphDb.index().forNodes( "users" );
    Node userNode = usersIndex.get( "name", username ).getSingle();
    if ( userNode != null )
    {
        return userNode;
    }

    tx.acquireWriteLock( lockNode );
    userNode = usersIndex.get( "name", username ).getSingle();
    if ( userNode == null )
    {
        userNode = graphDb.createNode( DynamicLabel.label( "User" ) );
        usersIndex.add( userNode, "name", username );
        userNode.setProperty( "name", username );
    }
    tx.success();
    return userNode;
}
```

Note that finishing the transaction will release the lock on the lock node.

35.13. Terminating a running transaction

Sometimes you may want to terminate (abort) a long running transaction from another thread.

Tip
The source code used in this example is found here: TerminateTransactions.java[38]

To begin with, we start the database server:

```
GraphDatabaseService graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
```

Now we start creating an infinite binary tree of nodes in the database, as an example of a long running transaction.

```
RelationshipType relType = DynamicRelationshipType.withName( "CHILD" );
Queue<Node> nodes = new LinkedList<>();
int depth = 1;

try ( Transaction tx = graphDb.beginTx() )
{
    Node rootNode = graphDb.createNode();
    nodes.add( rootNode );

    for (; true; depth++) {
        int nodesToExpand = nodes.size();
        for (int i = 0; i < nodesToExpand; ++i) {
            Node parent = nodes.remove();

            Node left = graphDb.createNode();
            Node right = graphDb.createNode();

            parent.createRelationshipTo( left, relType );
            parent.createRelationshipTo( right, relType );

            nodes.add( left );
            nodes.add( right );
        }
    }
}
catch ( TransactionTerminatedException ignored )
{
    return String.format( "Created tree up to depth %s in 1 sec", depth );
}
```

After waiting for some time, we decide to terminate the transaction. This is done from a separate thread.

```
tx.terminate();
```

Running this will execute the long running transaction for about one second and prints the maximum depth of the tree that was created before the transaction was terminated. No changes are actually made to the data — because the transaction has been terminated, the end result is as if no operations were performed.

Example output

```
Created tree up to depth 14 in 1 sec
```

[38] https://github.com/neo4j/neo4j/blob/2.2.1/community/embedded-examples/src/main/java/org/neo4j/examples/TerminateTransactions.java

Finally, let's shut down the database again.

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

35.14. Execute Cypher Queries from Java

Tip
The full source code of the example: JavaQuery.java[39]

In Java, you can use the Cypher query language as per the example below. First, let's add some data.

```
GraphDatabaseService db = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );

try ( Transaction tx = db.beginTx())
{
    Node myNode = db.createNode();
    myNode.setProperty( "name", "my node" );
    tx.success();
}
```

Execute a query:

```
try ( Transaction ignored = db.beginTx();
      Result result = db.execute( "match (n {name: 'my node'}) return n, n.name" ) )
{
    while ( result.hasNext() )
    {
        Map<String,Object> row = result.next();
        for ( Entry<String,Object> column : row.entrySet() )
        {
            rows += column.getKey() + ": " + column.getValue() + "; ";
        }
        rows += "\n";
    }
}
```

In the above example, we also show how to iterate over the rows of the `Result`[40].

The code will generate:

```
n.name: my node; n: Node[0];
```

Caution
When using an `Result`, you should consume the entire result (iterate over all rows using `next()`, iterating over the iterator from `columnAs()` or calling for example `resultAsString()`). Failing to do so will not properly clean up resources used by the `Result` object, leading to unwanted behavior, such as leaking transactions. In case you don't want to iterate over all of the results, make sure to invoke `close()` as soon as you are done, to release the resources tied to the result.

Tip
Using a try-with-resources statement[41] will make sure that the result is closed at the end of the statement. This is the recommended way to handle results.

You can also get a list of the columns in the result like this:

[39] https://github.com/neo4j/neo4j/blob/2.2.1/community/cypher/docs/cypher-docs/src/test/java/org/neo4j/cypher/example/JavaQuery.java
[40] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/Result.html
[41] http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

```
List<String> columns = result.columns();
```

This gives us:

```
[n, n.name]
```

To fetch the result items from a single column, do like below. In this case we'll have to read the property from the node and not from the result.

```
Iterator<Node> n_column = result.columnAs( "n" );
for ( Node node : IteratorUtil.asIterable( n_column ) )
{
    nodeResult = node + ": " + node.getProperty( "name" );
}
```

In this case there's only one node in the result:

```
Node[0]: my node
```

Only use this if the result only contains a single column, or you are only interested in a single column of the result.

Note

`resultAsString()`, `writeAsStringTo()`, `columnAs()` cannot be called more than once on the same `Result` object, as they consume the result. In the same way, part of the result gets consumed for every call to `next()`. You should instead use only one and if you need the facilities of the other methods on the same query result instead create a new `Result`.

For more information on the Java interface to Cypher, see the Java API[42].

For more information and examples for Cypher, see Part III, "Cypher Query Language" [101] and Chapter 5, *Basic Data Modeling Examples* [45].

[42] http://neo4j.com/docs/2.2.1/javadocs/index.html

35.15. Query Parameters

For more information on parameters see Section 8.5, “Parameters” [112].

Below follows example of how to use parameters when executing Cypher queries from Java.

Node id

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "id", 0 );
String query = "MATCH n WHERE id(n) = {id} RETURN n.name";
Result result = db.execute( query, params );
```

Node object

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "node", andreasNode );
String query = "MATCH n WHERE n = {node} RETURN n.name";
Result result = db.execute( query, params );
```

Multiple node ids

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "ids", Arrays.asList( 0, 1, 2 ) );
String query = "MATCH n WHERE id(n) in {ids} RETURN n.name";
Result result = db.execute( query, params );
```

String literal

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "name", "Johan" );
String query = "MATCH (n) WHERE n.name = {name} RETURN n";
Result result = db.execute( query, params );
```

Index value

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "value", "Michaela" );
String query = "START n=node:people(name = {value}) RETURN n";
Result result = db.execute( query, params );
```

Index query

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "query", "name:Andreas" );
String query = "START n=node:people({query}) RETURN n";
Result result = db.execute( query, params );
```

Numeric parameters for `SKIP` and `LIMIT`

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "s", 1 );
params.put( "l", 1 );
String query = "MATCH (n) RETURN n.name SKIP {s} LIMIT {l}";
Result result = db.execute( query, params );
```

Regular expression

```
Map<String, Object> params = new HashMap<String, Object>();
params.put( "regex", ".*h.*" );
String query = "MATCH (n) WHERE n.name =~ {regex} RETURN n.name";
Result result = db.execute( query, params );
```

Create node with properties

```
Map<String, Object> props = new HashMap<String, Object>();
props.put( "name", "Andres" );
props.put( "position", "Developer" );

Map<String, Object> params = new HashMap<String, Object>();
params.put( "props", props );
String query = "CREATE ({props})";
db.execute( query, params );
```

Create multiple nodes with properties

```
Map<String, Object> n1 = new HashMap<String, Object>();
n1.put( "name", "Andres" );
n1.put( "position", "Developer" );
n1.put( "awesome", true );

Map<String, Object> n2 = new HashMap<String, Object>();
n2.put( "name", "Michael" );
n2.put( "position", "Developer" );
n2.put( "children", 3 );

Map<String, Object> params = new HashMap<String, Object>();
List<Map<String, Object>> maps = Arrays.asList( n1, n2 );
params.put( "props", maps );
String query = "CREATE (n:Person {props}) RETURN n";
db.execute( query, params );
```

Setting all properties on node

```
Map<String, Object> n1 = new HashMap<>();
n1.put( "name", "Andres" );
n1.put( "position", "Developer" );

Map<String, Object> params = new HashMap<>();
params.put( "props", n1 );

String query = "MATCH (n) WHERE n.name='Michaela' SET n = {props}";
db.execute( query, params );
```

Chapter 36. The Traversal Framework

The Neo4j Traversal API[1] is a callback based, lazily executed way of specifying desired movements through a graph in Java. Some traversal examples are collected under Section 35.8, “Traversal” [604].

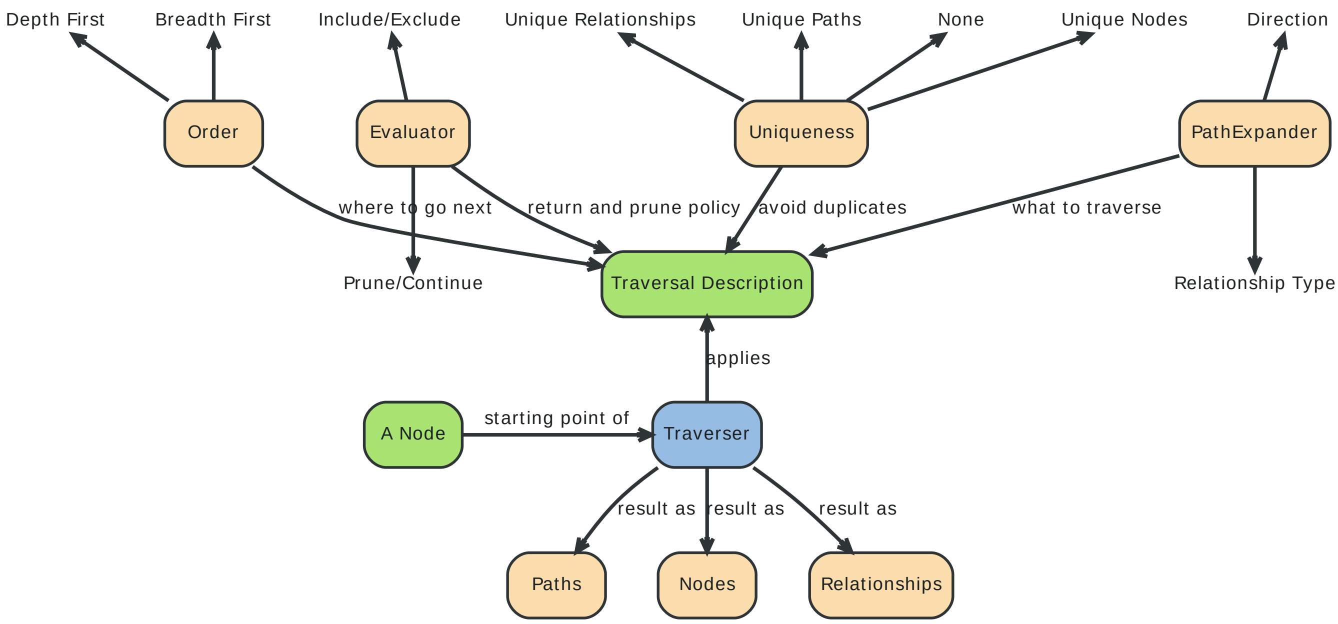
You can also use The Cypher Query Language as a powerful declarative way to query the graph.

[1] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/package-summary.html

36.1. Main concepts

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

- *Pathexpanders* — define what to traverse, typically in terms of relationship 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.
- *Starting nodes* where the traversal will begin.



See Section 36.2, “Traversal Framework Java API” [626] for more details.

36.2. 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 `PathExpander` (replacing `RelationshipExpander` and `Expander`) interface is central to traversals, but users of the API rarely need to implement it. There are also a set of interfaces for advanced use, when explicit control over the traversal order is required: `BranchSelector`, `BranchOrderingPolicy` and `TraversalBranch`.

TraversalDescription

The `TraversalDescription`[2] 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*, regardless 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[3] and another one excluding direction[4], where the latter traverses relationships in both directions[5].

Evaluator

`Evaluator`[6]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.

Traverser

The `Traverser`[7] object is the result of invoking `traverse()`[8] 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.

Uniqueness

Sets the rules for how positions can be revisited during a traversal as stated in `Uniqueness`[9]. Default if not set is `NODE_GLOBAL`[10].

---

[2] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html
[3] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#relationships
[4] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#relationships
[5] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/Direction.html#BOTH
[6] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/Evaluator.html
[7] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/Traverser.html
[8] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#traverse(org.neo4j.graphdb.Node)
[9] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/Uniqueness.html
[10] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/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[11]/ breadth-first[12] `BranchSelector|ordering` policies. The same result can be achieved by calling the `order`[13] method with ordering policies from `BranchOrderingPolicies`[14], or to write your own `BranchSelector/BranchOrderingPolicy` and pass in.

Order — How to move through branches?

A more generic version of depthFirst/breadthFirst methods in that it allows an arbitrary `BranchOrderingPolicy`[15] to be injected into the description.

BranchSelector

A `BranchSelector/BranchOrderingPolicy` 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:

- `BranchOrderingPolicies.PREORDER_DEPTH_FIRST`: Traversing depth first, visiting each node before visiting its child nodes.
- `BranchOrderingPolicies.POSTORDER_DEPTH_FIRST`: Traversing depth first, visiting each node after visiting its child nodes.
- `BranchOrderingPolicies.PREORDER_BREADTH_FIRST`: Traversing breadth first, visiting each node before visiting its child nodes.
- `BranchOrderingPolicies.POSTORDER_BREADTH_FIRST`: 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.

[11] http://en.wikipedia.org/wiki/Depth-first_search
[12] http://en.wikipedia.org/wiki/Breadth-first_search
[13] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#order
[14] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/BranchOrderingPolicies.html
[15] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/BranchOrderingPolicy.html

`BranchSelector`s 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 `BranchSelector`s to decide in what order branches are returned (where a branch's position is represented as a `Path`[16] from the start node to the current node). Common policies are `depth-first`[17] and `breadth-first`[18] and that's why there are convenience methods for those. For example, calling `TraversalDescription#depthFirst()`[19] is equivalent to:

```
description.order( BranchOrderingPolicies.PREORDER_DEPTH_FIRST );
```

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`[20]es from the current one.

Path

A `Path`[21] 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.

PathExpander/RelationshipExpander

The traversal framework use `PathExpander`s (replacing `RelationshipExpander`) to discover the relationships that should be followed from a particular path to further branches in the traversal.

Expander

A more generic version of relationships where a `RelationshipExpander` is injected, defining all relationships to be traversed for any given node.

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`.

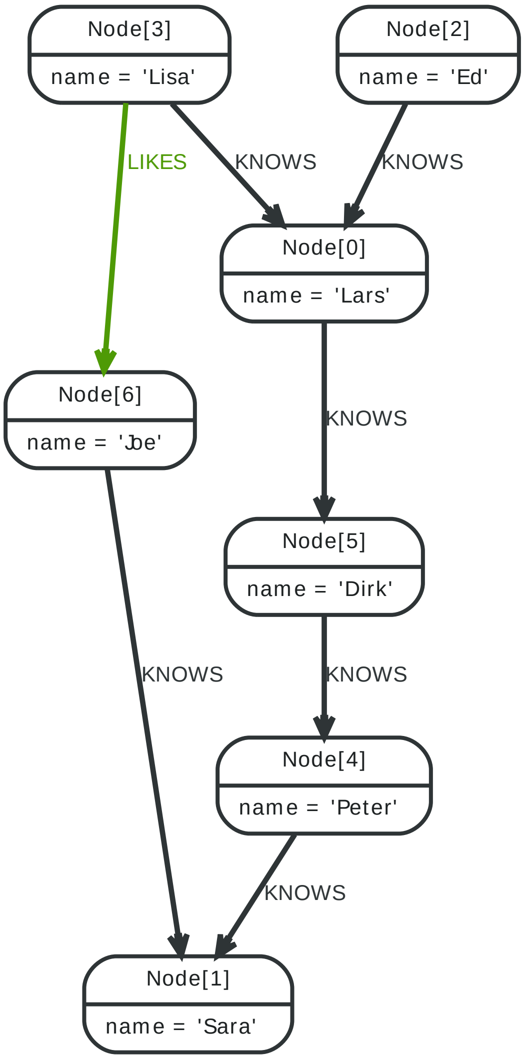
All the RelationshipExpanders provided by the Neo4j traversal framework also implement the Expander interface. For a user of the traversal API it is easier to implement the PathExpander/RelationshipExpander interface, since it only contains one method — the method for getting the relationships from a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

[16] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/Path.html
[17] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#depthFirst()
[18] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#breadthFirst()
[19] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#depthFirst()
[20] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/TraversalBranch.html
[21] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/Path.html

How to use the Traversal framework

A traversal description[22] is built using a fluent interface and such a description can then spawn traversers[23].

Figure 36.1. Traversal Example Graph



With the definition of the `RelationshipTypes` as

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

The graph can be traversed with for example the following traverser, starting at the “Joe” node:

```
for ( Path position : db.traversalDescription()
        .depthFirst()
        .relationships( Rels.KNOWS )
        .relationships( Rels.LIKES, Direction.INCOMING )
        .evaluator( Evaluators.toDepth( 5 ) )
        .traverse( node ) )
{
    output += position + "\n";
}
```

[22] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html
[23] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/Traverser.html

The traversal will output:

```
(6)
(6)<--[LIKES,1]--(3)
(6)<--[LIKES,1]--(3)--[KNOWS,6]-->(0)
(6)<--[LIKES,1]--(3)--[KNOWS,6]-->(0)--[KNOWS,4]-->(5)
(6)<--[LIKES,1]--(3)--[KNOWS,6]-->(0)--[KNOWS,4]-->(5)--[KNOWS,3]-->(4)
(6)<--[LIKES,1]--(3)--[KNOWS,6]-->(0)--[KNOWS,4]-->(5)--[KNOWS,3]-->(4)--[KNOWS,2]-->(1)
(6)<--[LIKES,1]--(3)--[KNOWS,6]-->(0)<--[KNOWS,5]--(2)
```

Since `TraversalDescription`[24]s are immutable it is also useful to create template descriptions which holds common settings shared by different traversals. For example, let's start with this traverser:

```
friendsTraversal = db.traversalDescription()
        .depthFirst()
        .relationships( Rels.KNOWS )
        .uniqueness( Uniqueness.RELATIONSHIP_GLOBAL );
```

This traverser would yield the following output (we will keep starting from the "Joe" node):

```
(6)
(6)--[KNOWS,0]-->(1)
(6)--[KNOWS,0]-->(1)<--[KNOWS,2]--(4)
(6)--[KNOWS,0]-->(1)<--[KNOWS,2]--(4)<--[KNOWS,3]--(5)
(6)--[KNOWS,0]-->(1)<--[KNOWS,2]--(4)<--[KNOWS,3]--(5)<--[KNOWS,4]--(0)
(6)--[KNOWS,0]-->(1)<--[KNOWS,2]--(4)<--[KNOWS,3]--(5)<--[KNOWS,4]--(0)<--[KNOWS,6]--(3)
(6)--[KNOWS,0]-->(1)<--[KNOWS,2]--(4)<--[KNOWS,3]--(5)<--[KNOWS,4]--(0)<--[KNOWS,5]--(2)
```

Now let's create a new traverser from it, restricting depth to three:

```
for ( Path path : friendsTraversal
        .evaluator( Evaluators.toDepth( 3 ) )
        .traverse( node ) )
{
    output += path + "\n";
}
```

This will give us the following result:

```
(6)
(6)--[KNOWS,0]-->(1)
(6)--[KNOWS,0]-->(1)<--[KNOWS,2]--(4)
(6)--[KNOWS,0]-->(1)<--[KNOWS,2]--(4)<--[KNOWS,3]--(5)
```

Or how about from depth two to four? That's done like this:

```
for ( Path path : friendsTraversal
        .evaluator( Evaluators.fromDepth( 2 ) )
        .evaluator( Evaluators.toDepth( 4 ) )
        .traverse( node ) )
{
    output += path + "\n";
}
```

This traversal gives us:

```
(6)--[KNOWS,0]-->(1)<--[KNOWS,2]--(4)
(6)--[KNOWS,0]-->(1)<--[KNOWS,2]--(4)<--[KNOWS,3]--(5)
(6)--[KNOWS,0]-->(1)<--[KNOWS,2]--(4)<--[KNOWS,3]--(5)<--[KNOWS,4]--(0)
```

[24] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html

For various useful evaluators, see the Evaluators[25] Java API or simply implement the Evaluator[26] interface yourself.

If you're not interested in the `Path`[27]s, but the `Node`[28]s you can transform the traverser into an iterable of nodes[29] like this:

```
for ( Node currentNode : friendsTraversal
        .traverse( node )
        .nodes() )
{
    output += currentNode.getProperty( "name" ) + "\n";
}
```

In this case we use it to retrieve the names:

```
Joe
Sara
Peter
Dirk
Lars
Lisa
Ed
```

Relationships[30] are fine as well, here's how to get them:

```
for ( Relationship relationship : friendsTraversal
        .traverse( node )
        .relationships() )
{
    output += relationship.getType().name() + "\n";
}
```

Here the relationship types are written, and we get:

```
KNOWS
KNOWS
KNOWS
KNOWS
KNOWS
KNOWS
```

Tip
The source code for the traversers in this example is available at: TraversalExample.java[31]

[25] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/Evaluators.html
[26] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/Evaluator.html
[27] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/Path.html
[28] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/Node.html
[29] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/Traverser.html#nodes()
[30] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/traversal/Traverser.html#relationships()
[31] https://github.com/neo4j/neo4j/blob/2.2.1/community/embedded-examples/src/main/java/org/neo4j/examples/TraversalExample.java

Chapter 37. Legacy Indexing

> **Note**
> This is not the same as indexes defined in the schema, the documentation below is for the legacy indexing in Neo4j.

This chapter focuses on how to use the Manual Indexes and Autoindexes. As of Neo4j 2.0, this is not the favored method of indexing data in Neo4j, instead we recommend defining indexes in the database schema.

However, support for legacy indexes remains, because certain features, such as uniqueness constraints, are not yet handled by the new indexes.

37.1. Introduction

Legacy Indexing operations are part of the Neo4j index API[1].

Each index is tied to a unique, user-specified name (for example "first_name" or "books") and can index either nodes[2] or relationships[3].

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.

For initial import of data using indexes, see Section 38.2, “Index Batch Insertion” [653].

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

[1] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/package-summary.html
[2] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/Node.html
[3] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/Relationship.html

37.2. 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 37.8, "Relationship indexes" [642] for more information specific to relationship indexes.

See Section 37.10, "Configuration and fulltext indexes" [644] 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" );
```

37.3. Delete

Indexes can be deleted. When deleting, the entire contents of the index will be removed as well as its associated configuration. An 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()[4] 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.

[4] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/Index.html#delete%28%29

37.4. 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();
reeves.setProperty( "name", "Keanu Reeves" );
actors.add( reeves, "name", reeves.getProperty( "name" ) );
Node bellucci = graphDb.createNode();
bellucci.setProperty( "name", "Monica Bellucci" );
actors.add( bellucci, "name", bellucci.getProperty( "name" ) );
// multiple values for a field, in this case for search only
// and not stored as a property.
actors.add( bellucci, "name", "La Bellucci" );
// Movies
Node theMatrix = graphDb.createNode();
theMatrix.setProperty( "title", "The Matrix" );
theMatrix.setProperty( "year", 1999 );
movies.add( theMatrix, "title", theMatrix.getProperty( "title" ) );
movies.add( theMatrix, "year", theMatrix.getProperty( "year" ) );
Node theMatrixReloaded = graphDb.createNode();
theMatrixReloaded.setProperty( "title", "The Matrix Reloaded" );
theMatrixReloaded.setProperty( "year", 2003 );
movies.add( theMatrixReloaded, "title", theMatrixReloaded.getProperty( "title" ) );
movies.add( theMatrixReloaded, "year", 2003 );
Node malena = graphDb.createNode();
malena.setProperty( "title", "Malèna" );
malena.setProperty( "year", 2000 );
movies.add( malena, "title", malena.getProperty( "title" ) );
movies.add( malena, "year", malena.getProperty( "year" ) );
```

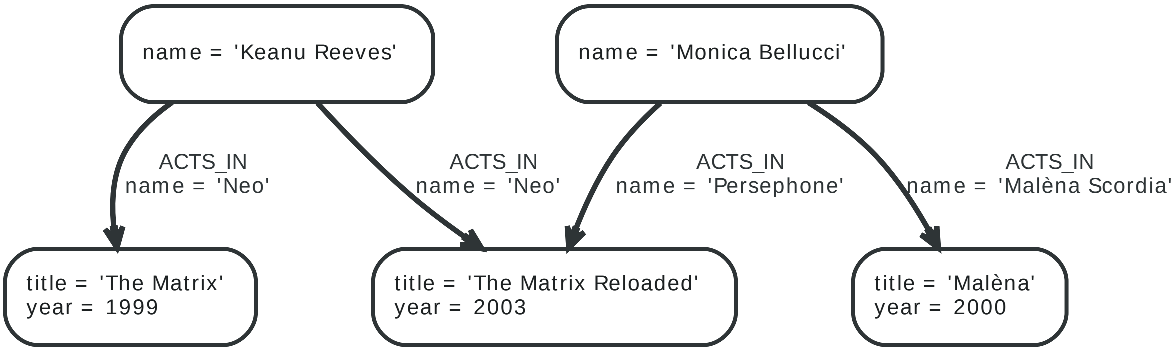
Note that there can be multiple values associated with the same entity and key.

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 );
role1.setProperty( "name", "Neo" );
roles.add( role1, "name", role1.getProperty( "name" ) );
Relationship role2 = reeves.createRelationshipTo( theMatrixReloaded, ACTS_IN );
role2.setProperty( "name", "Neo" );
roles.add( role2, "name", role2.getProperty( "name" ) );
Relationship role3 = bellucci.createRelationshipTo( theMatrixReloaded, ACTS_IN );
role3.setProperty( "name", "Persephone" );
roles.add( role3, "name", role3.getProperty( "name" ) );
Relationship role4 = bellucci.createRelationshipTo( malena, ACTS_IN );
role4.setProperty( "name", "Malèna Scordia" );
roles.add( role4, "name", role4.getProperty( "name" ) );
```

After these operations, our example graph looks like this:

Figure 37.1. Movie and Actor Graph



37.5. Remove

Removing[5] from an index is similar to adding, but can be done by supplying one of the following combinations of arguments:

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

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

[5] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/Index.html#remove%28T,%20java.lang.String,%20java.lang.Object%29

37.6. Update

Important
To update an index entry, the old one must be removed and a new one added. For details on removing index entries, see Section 37.5, “Remove” [638].

Remember that a node or relationship can be associated with any number of key-value pairs in an index. This 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 that demonstrates how it's done:

```
// create a node with a property
// so we have something to update later on
Node fishburn = graphDb.createNode();
fishburn.setProperty( "name", "Fishburn" );
// index it
actors.add( fishburn, "name", fishburn.getProperty( "name" ) );
// update the index entry
// when the property value changes
actors.remove( fishburn, "name", fishburn.getProperty( "name" ) );
fishburn.setProperty( "name", "Laurence Fishburn" );
actors.add( fishburn, "name", fishburn.getProperty( "name" ) );
```

37.7. Search

An index can be searched in two ways, get[6] and query[7]. 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[8] can be used directly with the default indexing backend.

Get

This is how to search for a single exact match:

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

IndexHits[9] is an `Iterable` with some additional useful methods. For example getSingle()[10] 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 case you don't iterate through all the hits, IndexHits.close()[11] must be called explicitly.

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.
}
```

[6] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/Index.html#get%28java.lang.String,%20java.lang.Object%29
[7] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/Index.html#query%28java.lang.String,%20java.lang.Object%29
[8] http://lucene.apache.org/core/3_6_2/queryparsersyntax.html
[9] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/IndexHits.html
[10] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/IndexHits.html#getSingle%28%29
[11] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/IndexHits.html#close%28%29

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 the section called "Querying with Lucene Query objects" [646] for how to do that.

37.8. 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[12] interface which extends Index<Relationship>[13].

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:
try ( Transaction tx = graphDb.beginTx() )
{
    IndexHits<Relationship> 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.

[12] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/RelationshipIndex.html
[13] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/Index.html

37.9. Scores

The `IndexHits` interface exposes scoring[14] 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 the section called "Sorting" [645] 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] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/IndexHits.html#currentScore%28%29

37.10. 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();
```

Here's an example of how to create an exact index which is case-insensitive:

```
Index<Node> index = graphDb.index().forNodes( "exact-case-insensitive",
        stringMap( "type", "exact", "to_lower_case", "true" ) );
Node node = graphDb.createNode();
index.add( node, "name", "Thomas Anderson" );
assertContains( index.query( "name", "\"Thomas Anderson\"" ), node );
assertContains( index.query( "name", "\"thoMas ANDerson\"" ), node );
```

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[15]. `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[16]	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.

[15] http://lucene.apache.org/core/3_6_2/api/core/org/apache/lucene/analysis/KeywordAnalyzer.html
[16] http://lucene.apache.org/core/3_6_2/api/core/org/apache/lucene/analysis/Analyzer.html

37.11. Extra features for Lucene indexes

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[17] 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 ) );
```

Sorting

Lucene performs sorting very well, and that is also exposed in the index backend, through the QueryContext[18] 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 )
{
```

[17] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/index/lucene/ValueContext.html
[18] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/index/lucene/QueryContext.html

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

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[19] 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.

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.

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[20] class:

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

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[21] 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.

[19] http://lucene.apache.org/core/3_6_2/api/core/org/apache/lucene/search/TermQuery.html
[20] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/index/lucene/QueryContext.html
[21] http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used

37.12. 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.

Configuration

By default Auto Indexing is off for both Nodes and Relationships. To configure this in the *conf/neo4j.properties* file, use the configuration keys `node_auto_indexing` and `relationship_auto_indexing`. For embedded mode, use the configuration options `GraphDatabaseSettings.node_auto_indexing` and `GraphDatabaseSettings.relationship_auto_indexing`. In both cases, set the value to `true`. This will enable automatic indexing on startup. Just note that we're not done yet, see below!

To actually auto index something, you have to set which properties should get indexed. You do this by listing the property keys to index on. In the configuration file, use the `node_keys_indexable` and `relationship_keys_indexable` configuration keys. When using embedded mode, use the `GraphDatabaseSettings.node_keys_indexable` and `GraphDatabaseSettings.relationship_keys_indexable` configuration keys. In all cases, the value should be a comma separated list of property keys to index on.

When coding in Java, it's done like this:

```
/*
 * 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.
 */
GraphDatabaseService graphDb = new GraphDatabaseFactory().
    newEmbeddedDatabaseBuilder( storeDirectory ).
    setConfig( GraphDatabaseSettings.node_keys_indexable, "nodeProp1,nodeProp2" ).
    setConfig( GraphDatabaseSettings.relationship_keys_indexable, "relProp1,relProp2" ).
    setConfig( GraphDatabaseSettings.node_auto_indexing, "true" ).
    setConfig( GraphDatabaseSettings.relationship_auto_indexing, "true" ).
    newGraphDatabase();

Node node1 = null, node2 = null;
Relationship rel = null;
try ( Transaction tx = graphDb.beginTx() )
{
    // 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();
}
```

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`[22] with exactly the same functionality. Continuing from the previous example, accessing the index is done like this:

```
try ( Transaction tx = graphDb.beginTx() )
{
    // 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() );
}
```

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
GraphDatabaseService graphDb = new GraphDatabaseFactory().
        newEmbeddedDatabase( storeDirectory );

// 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 );
```

[22] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/graphdb/index/Index.html

Note
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.

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 similarly happens when you remove the actual property, but only while the property is auto indexed. In particular, if you had configured your database to auto index a property, but later removed it from the configuration, deleting that property will not remove it from the auto index.

Chapter 38. Batch Insertion

Neo4j has a batch insertion facility intended for initial imports, which bypasses transactions and other checks in favor of performance. This is useful when you have a big dataset that needs to be loaded once.

Batch insertion is included in the neo4j-kernel[1] component, which is part of all Neo4j distributions and editions.

Be aware of the following points when using batch insertion:

- The intended use is for initial import of data but you can use it on an existing database if the existing database is shutdown first.
- Batch insertion is *not thread safe.*
- Batch insertion is *non-transactional.*
- Batch insertion will re-populate all existing indexes and indexes created during batch insertion on shutdown.
- Unless `shutdown` is successfully invoked at the end of the import, the database files *will* be corrupt.

Warning
Always perform batch insertion in a *single thread* (or use synchronization to make only one thread at a time access the batch inserter) and invoke `shutdown` when finished.

[1] http://search.maven.org/#search|ga|1|neo4j-kernel

38.1. Batch Inserter Examples

Initial import

To bulk load data using the batch inserter you'll need to write a Java application which makes use of the low level `BatchInserter`[2] interface.

Tip
You can't have multiple threads using the batch inserter concurrently without external synchronization.

You can get hold of an instance of `BatchInserter` by using `BatchInserters`[3]. Here's an example of the batch inserter in use:

```
BatchInserter inserter = null;
try
{
    inserter = BatchInserters.inserter(
            new File( "target/batchinserter-example" ).getAbsolutePath(),
            fileSystem );

    Label personLabel = DynamicLabel.label( "Person" );
    inserter.createDeferredSchemaIndex( personLabel ).on( "name" ).create();

    Map<String, Object> properties = new HashMap<>();

    properties.put( "name", "Mattias" );
    long mattiasNode = inserter.createNode( properties, personLabel );

    properties.put( "name", "Chris" );
    long chrisNode = inserter.createNode( properties, personLabel );

    RelationshipType knows = DynamicRelationshipType.withName( "KNOWS" );
    inserter.createRelationship( mattiasNode, chrisNode, knows, null );
}
finally
{
    if ( inserter != null )
    {
        inserter.shutdown();
    }
}
```

When creating a relationship you can set properties on the relationship by passing in a map containing properties rather than `null` as the last parameter to `createRelationship`.

It's important that the call to `shutdown` is inside a finally block to ensure that it gets called even if exceptions are thrown. If he batch inserter isn't cleanly shutdown then the consistency of the store is not guaranteed.

Tip
The source code for the examples on this page can be found here: BatchInsertDocTest.java[4]

[2] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/unsafe/batchinsert/BatchInserter.html
[3] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/unsafe/batchinsert/BatchInserters.html
[4] https://github.com/neo4j/neo4j/blob/2.2.1/community/kernel/src/test/java/examples/BatchInsertDocTest.java

Setting configuration options

You can pass custom configuration options to the `BatchInserter`. (See the section called “Batch insert example” [463] for information on the available options.) e.g.

```
Map<String, String> config = new HashMap<>();
config.put( "dbms.pagecache.memory", "512m" );
BatchInserter inserter = BatchInserters.inserter(
        new File("target/batchinserter-example-config").getAbsolutePath(), fileSystem, config );
// Insert data here ... and then shut down:
inserter.shutdown();
```

Alternatively you could store the configuration in a file:

batchinsert-config

```
dbms.pagecache.memory=8m
```

You can then refer to that file when initializing `BatchInserter`:

```
try ( InputStream input = fileSystem.openAsInputStream( new File( "target/docs/batchinsert-config" ).getAbsoluteFile() ) )
{
    Map<String, String> config = MapUtil.load( input );
    BatchInserter inserter = BatchInserters.inserter(
            "target/docs/batchinserter-example-config", fileSystem, config );
    // Insert data here ... and then shut down:
    inserter.shutdown();
}
```

Importing into an existing database

Although it’s a less common use case, the batch inserter can also be used to import data into an existing database. However, you will need to ensure that the existing database is *shut down* before you write to it.

Warning
Since the batch importer bypasses transactions there is a possibility of data inconsistency if the import process crashes midway. We would strongly suggest you take a backup of your existing database before using the batch inserter against it.

38.2. Index Batch Insertion

For general notes on batch insertion, see Chapter 38, *Batch Insertion* [650].

Indexing during batch insertion is done using BatchInserterIndex[5] which are provided via BatchInserterIndexProvider[6]. An example:

```
BatchInserter inserter = BatchInserters.inserter( "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 as well
indexProvider.shutdown();
inserter.shutdown();
```

The configuration parameters are the same as mentioned in Section 37.10, “Configuration and fulltext indexes” [644].

Best practices

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

- Try to avoid flushing[7] 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[8] 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.

[5] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/unsafe/batchinsert/BatchInserterIndex.html
[6] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/unsafe/batchinsert/BatchInserterIndexProvider.html
[7] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/unsafe/batchinsert/BatchInserterIndex.html#flush%28%29
[8] http://neo4j.com/docs/2.2.1/javadocs/org/neo4j/unsafe/batchinsert/BatchInserterIndex.html#setCacheCapacity%28java.lang.String,%20int%29

Terminology

The terminology used for Cypher and Neo4j is drawn from the worlds of database design and graph theory. This section provides cross-linked summaries of common terms.

In some cases, multiple terms (e.g., arc, edge, relationship) may be used for the same or similar concept. An asterisk (*) to the right of a term indicates that the term is commonly used for Neo4j and Cypher.

acyclic
: *for a graph or subgraph:* when there is no way to start at some node `n` and follow a sequence of adjacent relationships that eventually loops back to `n` again. The opposite of cyclic.

adjacent
: nodes sharing an incident (that is, directly-connected) relationship or relationships sharing an incident node.

aggregating expression
: expression that summarizes a set of values, like computing their sum or their maximum.

arc
: *graph theory:* a synonym for a directed relationship.

array
: container that holds a number of elements. The element types can be the types supported by the underlying graph storage layer, but all elements must be of the same type.

attribute
: Synonym for property.

clause
: component of a Cypher query or command; starts with an identifying keyword (for example `CREATE`). The following clauses currently exist in Cypher: `CREATE`, `CREATE UNIQUE`, `DELETE`, `FOREACH`, `LOAD CSV`, `MATCH`, `MERGE`, `OPTIONAL MATCH`, `REMOVE`, `RETURN`, `SET`, `START`, `UNION`, and `WITH`.

co-incident
: alternative term for adjacent relationships, which share a common node.

collection
: container that holds a number of values. The values can have mixed types.

command
: a statement that operates on the database without affecting the data graph or returning content from it.

commit
: successful completion of a transaction, ensuring durability of any changes made.

constraint
: *part of a database schema:* defines a contract that the database will never break (for example, uniqueness of a property on all nodes that have a specific label).

cyclic
: The opposite of acyclic.

Cypher
: a special-purpose programming language for describing queries and operations on a graph database, with accompanying natural language concepts.

DAG
: a directed, acyclic graph: there are no cyclic paths and all the relationships are directed.

data graph
: graph stored in the database. See also property graph.

data record
: a unit of storage containing an arbitrary unordered collection of properties.

degree
: *of a node:* is the number of relationships leaving or entering (if directed) the node; loops are counted twice.

directed relationship
: a relationship that has a direction; that is the relationship has a source node and a destination node. The opposite of an undirected relationship. All relationships in a Neo4j graph are directed.

edge
: *graph theory:* a synonym for undirected relationship.

execution plan
: parsed and compiled statement that is ready for Neo4j to execute. An execution plan consists of the physical operations that need to be performed in order to achieve the intent of the statement.

execution result
: all statements return an execution result. For queries, this can contain an iterator of result rows.

expression
: produces values; may be used in *projections*, as a *predicate*, or when setting *properties* on graph elements.

graph
: 1. data graph,
  2. property graph,
  3. *graph theory:* set of vertices and edges.

graph database
: a database that uses graph-based structures (for example, nodes, relationships, properties) to represent and store data.

graph element
: node or relationship that is part of a graph.

identifier
: identifiers are named bindings to values (for example, collections, scalars) in a statement. For example, in `MATCH n RETURN n`, `n` is an identifier.

incident
: adjacent relationship attached to a node or a node attached to a relationship.

incoming relationship
: *pertaining to a directed relationship:* from the point of view of a node `n`, this is any relationship `r` arriving at `n`, exemplified by `()-[:r]->(n)`. The opposite of outgoing.

index
: data structure that improves performance of a database by redundantly storing the same information in a way that is faster to read.

intermediate result
: set of identifiers and values (record) passed from one clause to another during query execution. This is internal to the execution of a given query.

label
: marks a node as a member of a named subset. A node may be assigned zero or more labels. Labels are written as `:label` in Cypher (the actual label is prefixed by a colon). Note: *graph theory:* This differs from mathematical graphs, where a label applies uniquely to a single vertex.

loop
: a relationship that connects a node to itself.

neighbor
: *of node:* another node, connected by a common relationship; *of relationship:* another relationship, connected to a common node.

node*	data record within a data graph; contains an arbitrary collection of properties. Nodes may have zero, one, or more labels and are optionally connected by relationships. Similar to vertex.
null	`NULL` is a special marker, used to indicate that a data item does not exist in the graph or that the value of an expression is unknown or inapplicable.
operator	there are three categories of operators in Cypher: 1. *Arithmetic,* such as `+`, `/`, `%` etc.; 2. *Logical,* such as `OR`, `AND`, `NOT` etc.; and 3. *Comparison,* such as `<`, `>`, `=` etc.
outgoing relationship	*pertaining to a directed relationship:* from the point of view of a node `n`, this is any relationship `r` leaving `n`, exemplified by `(n)-[:r]->()`. The opposite of incoming relationship.
parameter	named value provided when running a statement. Parameters allow Cypher to efficiently re-use execution plans without having to parse and recompile every statement when only a literal value changes.
path	collection of alternating nodes and relationships that corresponds to a walk in the data graph.
pattern graph	graph used to express the shape (that is, connectivity pattern) of the data being searched for in the data graph. This is what `MATCH` and `WHERE` describe in a Cypher query.
predicate	expression that returns `TRUE`, `FALSE` or `NULL`. When used in `WHERE`, `NULL` is treated as `FALSE`.
projection	an operation taking result rows as both input and output data. This may be a subset of the identifiers provided in the input, a calculation based on single or multiple identifiers in the input, or both. The relevant clauses are `WITH` and `RETURN`.
property graph	a graph having directed, typed relationships. Each node or relationship may have zero or more associated properties.
property*	named value stored in a node or relationship. Synonym for attribute.
query	statement that reads or writes data from the database
relationship type	marks a relationship as a member of a named subset. A relationship must be assigned one and only one type. For example, in the Cypher pattern `(start)-[:TYPE]->(to)`, `TYPE` is the relationship type.
relationship*	data record in a property graph that associates an ordered pair of nodes. Similar to arc and edge.
result row	each query returns an iterator of result rows, which represents the result of executing the query. Each result row is a set of key-value pairs (a record).
rollback	abort of the containing transaction, effectively undoing any changes defined inside the transaction.

schema	persistent database state that describes available indexes and enabled constraints for the data graph.
schema command	statement that updates the schema.
statement	text string containing a Cypher query or command.
transaction	A transaction comprises a unit of work performed against a database. It is treated in a coherent and reliable way, independent of other transactions. A transaction, by definition, must be atomic, consistent, isolated, and durable.
transitive closure	*of a graph:* is a graph which contains a relationship from node `x` to node `y` whenever there is a directed path from `x` to `y`; For example, if there is a relationship from `a` to `b`, and another from `b` to `c`, then the transitive closure includes a relationship from `a` to `c`.
type	types classify values. Each value in Cypher has a concrete type. Supported types are: • string, • boolean, • the number types (double, integer, long), • the map types (plain maps, nodes, and relationships), • and collections of any concrete type. The type hierarchy supports several other types (for example, any, scalar, derived map, collection). These are used to classify values and collections of values having different concrete types.
undirected relationship	a relationship that doesn't have a direction. The opposite of directed relationship.
vertex	*graph theory:* the fundamental unit used to form a mathematical graph (plural: vertices). See node.

Appendix A. Resources

- Neo4j Cypher Refcard[1].
- Neo4j Javadocs[2]. You can also download a *javadocs.jar* file from Maven Central, see org.neo4j.doc:neo4j-javadocs[3] or download it from neo4j-javadocs-2.2.1-javadoc.jar[4] directly.
- Neo4j GraphGist, an online tool for creating interactive documents with executable Cypher statements: http://gist.neo4j.org/.
- The main Neo4j site at http://neo4j.com/ is a good starting point to learn about Neo4j.
- See http://neo4j.com/developer/language-guides/ for how to use Neo4j from different programming languages.

Below are some starting points within this manual:

- Section 2.1, "The Neo4j Graph Database" [5]
- Part III, "Cypher Query Language" [101]
- Chapter 3, *Introduction to Cypher* [16]
- Chapter 21, *REST API* [285]
- Chapter 23, *Installation & Deployment* [424]
- Section 23.4, "Upgrading" [430]
- Chapter 27, *Security* [496]

[1] http://neo4j.com/docs/2.2.1/cypher-refcard/
[2] http://neo4j.com/docs/2.2.1/javadocs/
[3] http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j.doc%22%20AND%20a%3A%22neo4j-javadocs%22
[4] http://central.maven.org/maven2/org/neo4j/doc/neo4j-javadocs/2.2.1/neo4j-javadocs-2.2.1-javadoc.jar

Appendix B. Manpages

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

- neo4j
- neo4j-shell
- neo4j-import
- neo4j-backup
- neo4j-arbiter

Name

neo4j — Neo4j Server control

Synopsis

neo4j <command>

DESCRIPTION

Neo4j is a graph database, perfect for working with highly connected data. The `neo4j` command is used to control the Neo4j Server.

The preferred way to install Neo4j on Linux systems is by using prebuilt installation packages. For information regarding Windows, see below.

COMMANDS

console	Start the server as an application, running as a foreground process. 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.
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.

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 running Cypher queries. There's also commands to get information about the database. In addition, you can browse 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.

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.
-c *COMMAND*	Command line to execute. After executing it the shell exits.
-file *FILE*	File to read and execute. After executing it the shell exits. If - is supplied as filename data is read from stdin instead.
-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.properties
neo4j-shell -path /path/to/db -readonly
```

Name

neo4j-import — Neo4j Import Tool

Synopsis

neo4j-import [options]

DESCRIPTION

neo4j-import is used to create a new Neo4j database from data in CSV files. See the chapter "Import Tool" in the Neo4j Manual for details on the CSV file format — a special kind of header is required.

OPTIONS

--into <store-dir>
Database directory to import into. Must not contain existing database.

--nodes [:Label1:Label2] "<file1>,<file2>,…"
Node CSV header and data. Multiple files will be logically seen as one big file from the perspective of the importer. The first line must contain the header. Multiple data sources like these can be specified in one import, where each data source has its own header. Note that file groups must be enclosed in quotation marks.

--relationships [:RELATIONSHIP_TYPE] "<file1>,<file2>,…"
Relationship CSV header and data. Multiple files will be logically seen as one big file from the perspective of the importer. The first line must contain the header. Multiple data sources like these can be specified in one import, where each data source has its own header. Note that file groups must be enclosed in quotation marks.

--delimiter <delimiter-character>
Delimiter character, or *TAB*, between values in CSV data. The default option is `,`.

--array-delimiter <array-delimiter-character>
Delimiter character, or *TAB*, between array elements within a value in CSV data. The default option is `;`.

--quote <quotation-character>
Character to treat as quotation character for values in CSV data. The default option is `"`. Quotes inside quotes escaped like `"""Go away"", he said."` and `"\"Go away\", he said."` are supported. If you have set "`'`" to be used as the quotation character, you could write the previous example like this instead: `'"Go away", he said.'`

--id-type <id-type>
One out of [STRING, INTEGER, ACTUAL] and specifies how ids in node/relationship input files are treated. STRING: arbitrary strings for identifying nodes. INTEGER: arbitrary integer values for identifying nodes. ACTUAL: (advanced) actual node ids. The default option is `STRING`.Default value: STRING

--processors <max processor count>
(advanced) Max number of processors used by the importer. Defaults to the number of available processors reported by the JVM. There is a certain amount of minimum threads needed so for that reason there is no lower bound for this value. For optimal performance this value shouldn't be greater than the number of available processors.

--stacktrace
Enable printing of error stack traces.

--bad <file name>
Relationships that refer to nodes that cannot be found can, instead of making the import fail, be logged to a file specified by this option. Can be relative (to store directory) or absolute

--bad-tolerance <max number of bad entries>
Number of bad entries before the import is considered failed. This tolerance threshold is about relationships refering to missing nodes. Format errors in input data are still treated as errors. Default value: 1000

--input-encoding <character set> Character set that input data is encoded in. Provided value must be one out of the available character sets in the JVM, as provided by Charset#availableCharsets(). If no input encoding is provided, the default character set of the JVM will be used.

Usage - Windows

The `Neo4jImport.bat` script is used in the same way.

EXAMPLES

Below is a basic example, where we import movies, actors and roles from three files.

movies.csv

```
movieId:ID,title,year:int,:LABEL
tt0133093,"The Matrix",1999,Movie
tt0234215,"The Matrix Reloaded",2003,Movie;Sequel
tt0242653,"The Matrix Revolutions",2003,Movie;Sequel
```

actors.csv

```
personId:ID,name,:LABEL
keanu,"Keanu Reeves",Actor
laurence,"Laurence Fishburne",Actor
carrieanne,"Carrie-Anne Moss",Actor
```

roles.csv

```
:START_ID,role,:END_ID,:TYPE
keanu,"Neo",tt0133093,ACTED_IN
keanu,"Neo",tt0234215,ACTED_IN
keanu,"Neo",tt0242653,ACTED_IN
laurence,"Morpheus",tt0133093,ACTED_IN
laurence,"Morpheus",tt0234215,ACTED_IN
laurence,"Morpheus",tt0242653,ACTED_IN
carrieanne,"Trinity",tt0133093,ACTED_IN
carrieanne,"Trinity",tt0234215,ACTED_IN
carrieanne,"Trinity",tt0242653,ACTED_IN
```

The command will look like this:

```
neo4j-import --into path_to_target_directory --nodes movies.csv --nodes actors.csv --relationships roles.csv
```

See the Neo4j Manual for further examples.

Name

neo4j-backup — Neo4j Backup Tool

Synopsis

neo4j-backup -host <host> [-port <port>] -to target_directory

DESCRIPTION

A tool to perform live backups over the network from a running Neo4j graph database onto a local filesystem. Backups can be either full or incremental. The first backup must be a full backup, after that incremental backups can be performed.

The source(s) are given as host:port pairs, the target is a filesystem location.

BACKUP TYPE

-full copies the entire database to a directory.

-incremental copies the changes that have taken place since the last full or incremental backup to an existing backup store.

The backup tool will automatically detect whether it needs to do a full or an incremental backup.

SOURCE ADDRESS

```
Backup sources are given in the following format:

*-host <host> [-port <port>]*

*host*::
  In single mode, the host of a source database; in HA mode, the cluster address of a cluster member.

*port*::
  In single mode, the port of a source database backup service; in HA mode, the port of a cluster instance. If not given, the
 default value `6362` will be used for single mode, `5001` for HA.

[[neo4j-backup-manpage-usage-important]]
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.

Usage - Windows

The `Neo4jBackup.bat` script is used in the same way.

EXAMPLES

```
# Performing a backup the first time: create a blank directory and run the backup tool
mkdir /mnt/backup/neo4j-backup
neo4j-backup -host 192.168.1.34 -to /mnt/backup/neo4j-backup

# Subsequent backups using the same _target_-directory will be incremental and therefore quick
neo4j-backup -host freja -to /mnt/backup/neo4j-backup

# Performing a backup where the service is registered on a custom port
neo4j-backup -host freja -port 9999 -to /mnt/backup/neo4j-backup

# Performing a backup from HA cluster, specifying a cluster member
./neo4j-backup -host oden -to /mnt/backup/neo4j-backup
```

```
# Performing a backup from HA cluster, specifying a cluster member registered on custom port
./neo4j-backup -host oden -port 9191 -to /mnt/backup/neo4j-backup
```

RESTORE FROM BACKUP

The Neo4j backups are fully functional databases. To use a backup, replace your database directory with the backup.

Name

neo4j-arbiter — Neo4j Arbiter for High-Availability clusters

Synopsis

neo4j-arbiter <command>

DESCRIPTION

Neo4j Arbiter is a service that can help break ties in Neo4j clusters that have an even number of cluster members.

COMMANDS

console	Start the server as an application, running as a foreground process. 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/arbiter.cfg	Arbiter server configuration.
conf/arbiter-wrapper.cfg	Configuration for service wrapper.