



The Neo4j Manual

The Neo4j Manual v2.3.1

The Neo4j Team neo4j.com¹

¹ <http://neo4j.com/>

The Neo4j Manual v2.3.1

by The Neo4j Team neo4j.com¹

Publication date 2015-11-10 14:12:30
Copyright © 2015 Neo Technology

Starting points

- [What is the Neo4j graph database?](#)
- [Cypher Query Language](#)
- [REST API](#)
- [Installation](#)
- [Upgrading](#)
- [Security](#)
- [Resources](#)

License: Creative Commons 3.0

This book is presented in open source and licensed through Creative Commons 3.0. You are free to copy, distribute, transmit, and/or adapt the work. This license is based upon the following conditions:

Attribution You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Any of the above conditions can be waived if you get permission from the copyright holder.

In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights
- The author's moral rights
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights



Note

For any reuse or distribution, you must make clear to the others the license terms of this work. The best way to do this is with a direct link to this page: <http://creativecommons.org/licenses/by-sa/3.0/>²

¹ <http://neo4j.com/>

² <http://creativecommons.org/licenses/by-sa/3.0/>

Preface	v
I. Introduction	1
1. Neo4j Highlights	3
2. Graph Database Concepts	4
II. Tutorials	14
3. Introduction to Cypher	16
4. Use Cypher in an application	46
5. Basic Data Modeling Examples	47
6. Advanced Data Modeling Examples	62
7. Languages	96
III. Cypher Query Language	102
8. Introduction	105
9. Syntax	118
10. General Clauses	136
11. Reading Clauses	156
12. Writing Clauses	188
13. Functions	216
14. Schema	245
15. Query Tuning	255
16. Execution Plans	261
IV. Reference	279
17. Capabilities	281
18. Transaction Management	287
19. Data Import	297
20. Graph Algorithms	298
21. REST API	299
22. Deprecations	435
V. Operations	436
23. Installation & Deployment	438
24. Configuration & Performance	450
25. High Availability	474
26. Backup	497
27. Security	502
28. Monitoring	508
VI. Tools	529
29. Import tool	531
30. Web Interface	543
31. Neo4j Shell	544
VII. Advanced Usage	560
32. Extending the Neo4j Server	562
33. Using Neo4j embedded in Java applications	575
34. The Traversal Framework	611
35. Legacy Indexing	619
36. Batch Insertion	634
Terminology	638
A. Resources	642
B. Manpages	643
neo4j	644
neo4j-shell	645
neo4j-import	646
neo4j-backup	648
neo4j-arbiter	649

Preface

This is the reference manual for Neo4j version 2.3.1, 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” [102] — details on the Cypher query language.
- Part IV, “Reference” [279] — detailed information on Neo4j.
- Part V, “Operations” [436] — how to install and maintain Neo4j.
- Part VI, “Tools” [529] — guides on tools.
- Part VII, “Advanced Usage” [560] — using Neo4j in more advanced ways.
- [Terminology](#) [638] — terminology about graph databases.
- [Appendix A, Resources](#) [642] — find additional documentation resources.
- [Appendix B, Manpages](#) [643] — 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](#) [642].

Who should read this?

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

Where to get help?

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 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.

If you want to contribute to the Neo4j open source project, see <http://neo4j.com/developer/contribute/>.

Part I. Introduction

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

1. Neo4j Highlights	3
2. Graph Database Concepts	4
2.1. The Neo4j Graph Database	5
2.2. Comparing Database Models	11

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 \[287\]](#).

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 \[474\]](#) 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

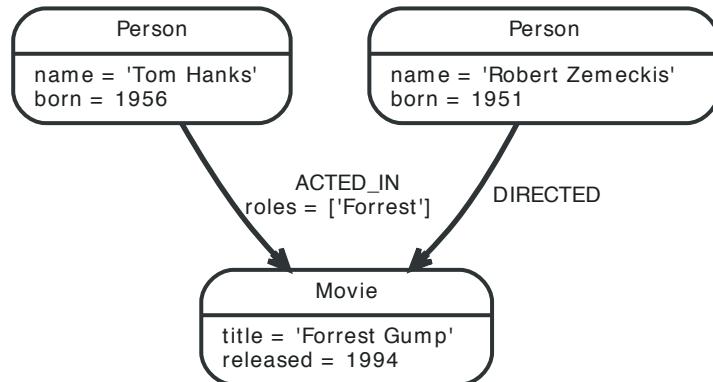
This chapter contains an introduction to the graph data model and also compares it to other data models used when persisting data.

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 \[638\]](#).

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*:

title = 'Forrest Gump'

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:

name = 'Tom Hanks'
born = 1956

title = 'Forrest Gump'
released = 1994

name = 'Robert Zemeckis'
born = 1951

Relationships

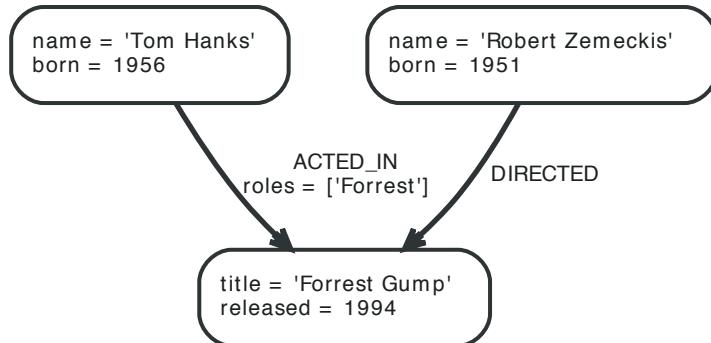
Relationships organize the nodes by connecting them. A relationship connects two nodes — a start node and an end node. Just like nodes, relationships can have properties.

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.

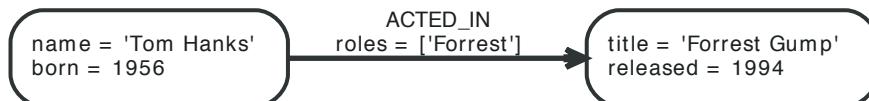
Relationships organize nodes into arbitrary structures, allowing a graph to resemble a list, a tree, a map, or a compound entity — any of which can be combined into yet more complex, richly interconnected structures.

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.

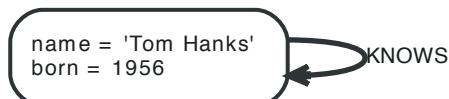


Relationships are equally well traversed in either direction.

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.

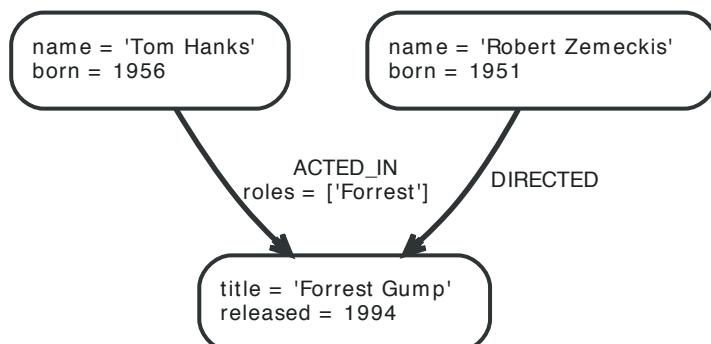
Note that a node can have relationships to itself as well:



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.

NULLs can instead be modeled by the absence of a key.

For further details on supported property values, see [Section 33.3, “Property values” \[584\]](#).

Labels

Labels assign roles or types to nodes.

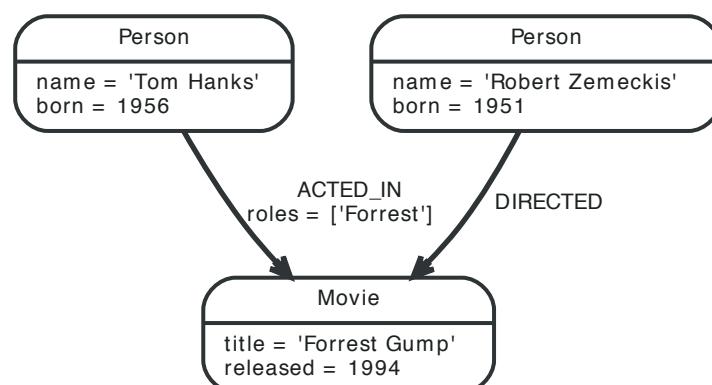
A label is a named graph construct that is used to group nodes into sets; all nodes labeled with the same label belongs to the same set. Many database queries can work with these sets instead of the whole graph, making queries easier to write and more efficient to execute. A node may be labeled with 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” \[9\]](#)).

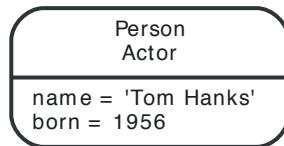
An example would be a label named `User` that you label all your nodes representing users with. With that in place, you can ask Neo4j to perform operations only on your user nodes, such as finding all users with a given name.

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.

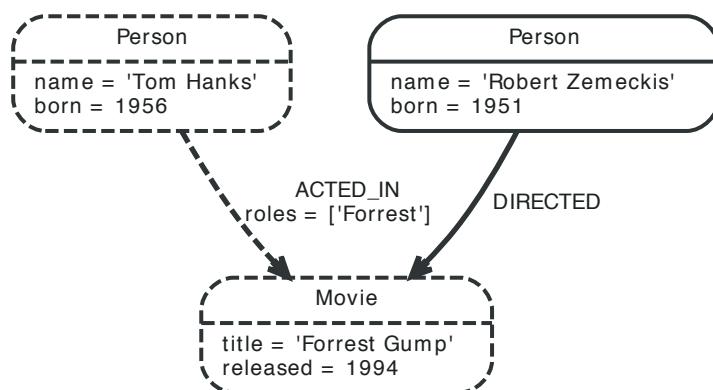
A traversal is how you query a graph, navigating from starting nodes to related nodes, finding answers to questions like "what music do my friends like that I don't yet own," or "if this power supply goes down, what web services are affected?"

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" [102] 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.

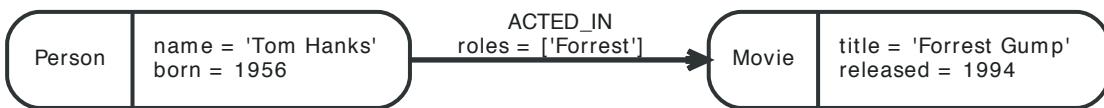
If we want to find out which movies Tom Hanks acted in according to our tiny example database the traversal would start from the Tom Hanks node, follow any ACTED_IN relationships connected to the node, and end up with Forrest Gump as the result (see the dashed lines):



Paths

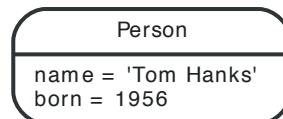
A path is one or more nodes with connecting relationships, typically retrieved as a query or traversal result.

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 \[474\]](#)). If you apply them on a slave you will receive a `Neo.ClientError.Transaction.InvalidType` error code (see [Section 21.2, “Neo4j Status Codes” \[309\]](#)).

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 35, Legacy Indexing \[619\]](#)).

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” \[246\]](#)
- REST API: [Section 21.15, “Indexing” \[369\]](#)
- Listing Indexes via Shell: [the section called “Listing Indexes and Constraints” \[553\]](#)
- Java Core API: [Section 33.4, “User database with indexes” \[585\]](#)

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” \[249\]](#)
- REST API: [Section 21.16, “Constraints” \[371\]](#)
- Listing Constraints via Shell: [the section called “Listing Indexes and Constraints” \[553\]](#)

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

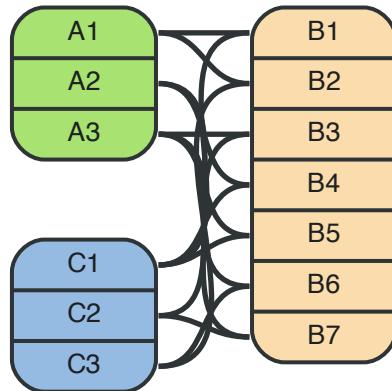
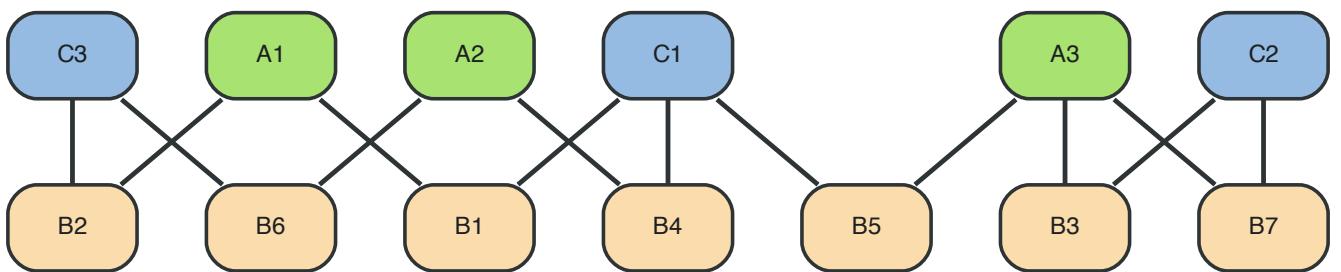


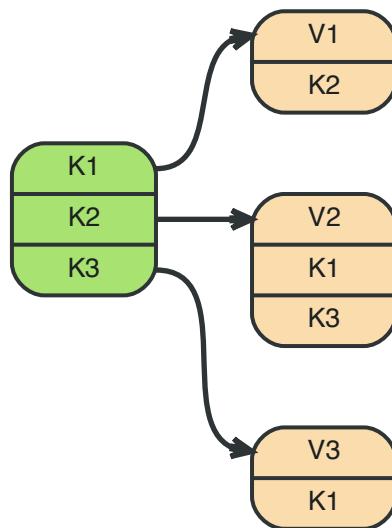
Figure 2.2. Graph Database as RDBMS



A Graph Database elaborates a Key-Value Store

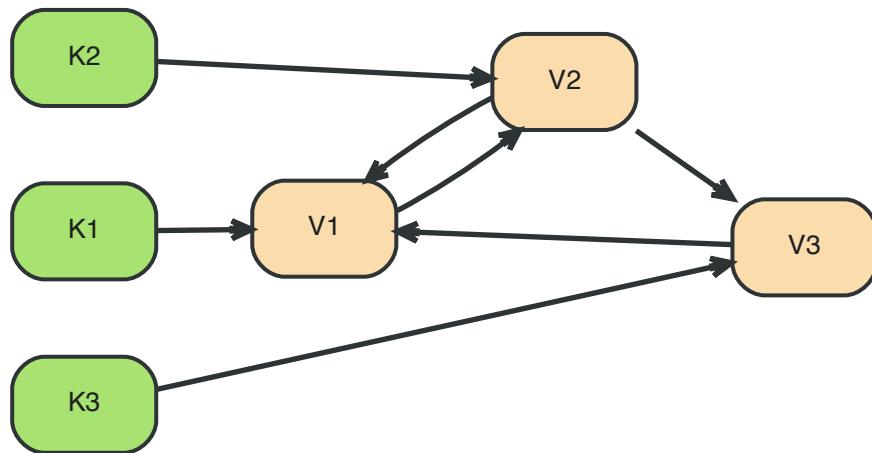
A Key-Value model is great for lookups of simple values or lists. When the values are themselves interconnected, you've got a graph. Neo4j lets you elaborate the simple data structures into more complex, interconnected data.

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



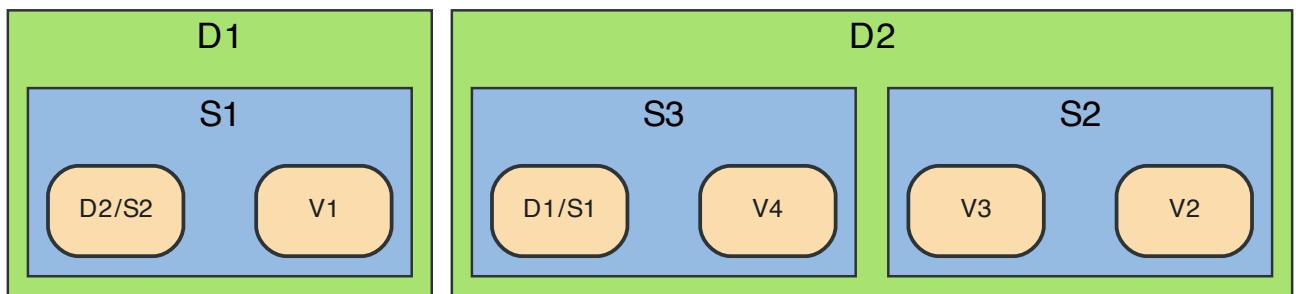
A Graph Database relates Column-Family

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

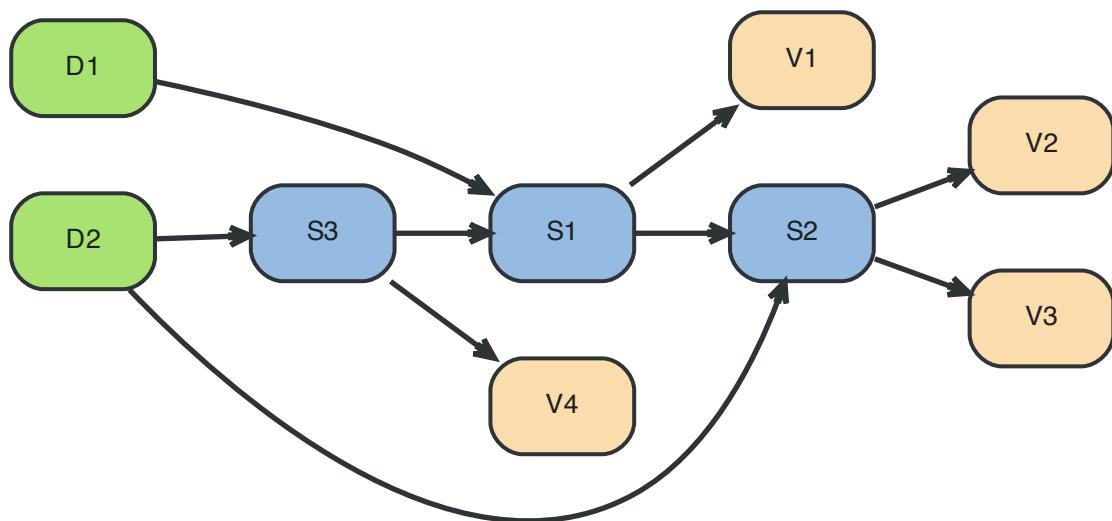
The container hierarchy of a document database accommodates nice, schema-free data that can easily be represented as a tree. Which is of course a graph. Refer to other documents (or document elements) within that tree and you have a more expressive representation of the same data. When in Neo4j, those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

3. Introduction to Cypher	16
3.1. Background and Motivation	17
3.2. Graphs, Patterns, and Cypher	18
3.3. Patterns in Practice	21
3.4. Getting the Results You Want	26
3.5. How to Compose Large Statements	30
3.6. Labels, Constraints and Indexes	32
3.7. Loading Data	34
3.8. Utilizing Data Structures	37
3.9. Cypher vs. SQL	40
4. Use Cypher in an application	46
5. Basic Data Modeling Examples	47
5.1. Movie Database	48
5.2. Social Movie Database	50
5.3. Finding Paths	52
5.4. Linked Lists	56
5.5. TV Shows	58
6. Advanced Data Modeling Examples	62
6.1. ACL structures in graphs	63
6.2. Hyperedges	67
6.3. Basic friend finding based on social neighborhood	69
6.4. Co-favorited places	70
6.5. Find people based on similar favorites	72
6.6. Find people based on mutual friends and groups	73
6.7. Find friends based on similar tagging	74
6.8. Multirelational (social) graphs	75
6.9. Implementing newsfeeds in a graph	76
6.10. Boosting recommendation results	79
6.11. Calculating the clustering coefficient of a network	80
6.12. Pretty graphs	81
6.13. A multilevel indexing structure (path tree)	85
6.14. Complex similarity computations	89
6.15. The Graphity activity stream model	90
6.16. User roles in graphs	92
7. Languages	96
7.1. How to use the REST API from Java	97

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/2.3.1/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 [SQL²](#) (used in [relational databases³](#)), Cypher is a textual, declarative query language. It uses a form of [ASCII art⁴](#) 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 [declarative⁵](#) language, Cypher does *not* tell Neo4j how to find nodes, traverse relationships, etc. (This level of control is available from Neo4j's [Java⁶](#) APIs⁷, see [Section 32.2, "Unmanaged Extensions" \[567\]](#))

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_\(programming_language\)](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 allows 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:

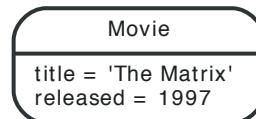
```
CREATE (:Movie { title:"The Matrix",released:1997 })
```

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}

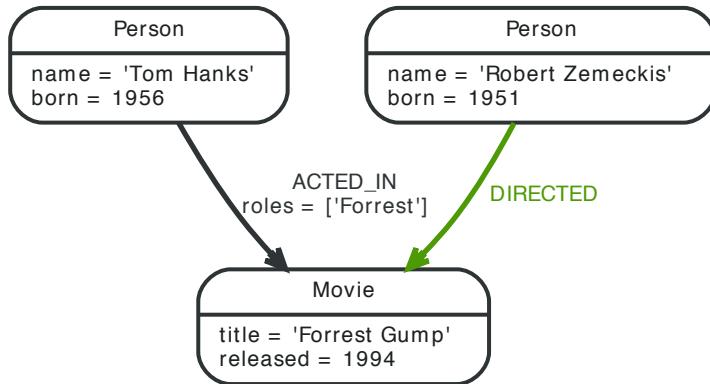
1 row
Nodes created: 1
Properties set: 2
Labels added: 1

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:



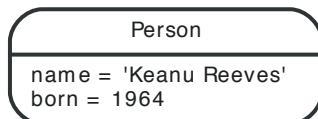
This should show both *The Matrix* and *Forrest Gump*.

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 identifier.property.

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

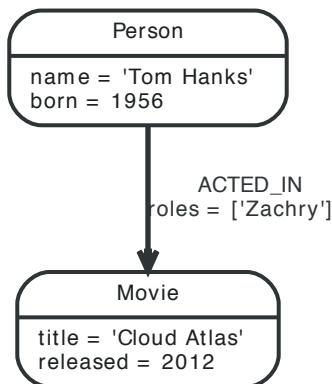
Now we know enough to connect new nodes to existing ones and can combine MATCH and CREATE to attach structures to the graph.

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

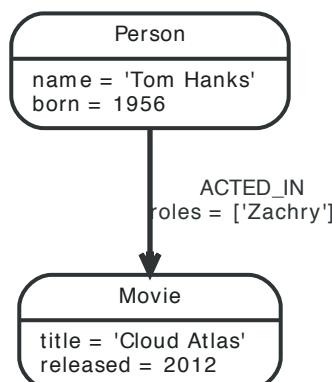
A `MERGE` clause without any previously assigned identifiers in it either matches the full pattern or creates the full pattern. It never produces a partial mix of matching and creating within a pattern. To achieve a partial match/create, make sure to use already defined identifiers for the parts that 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
```

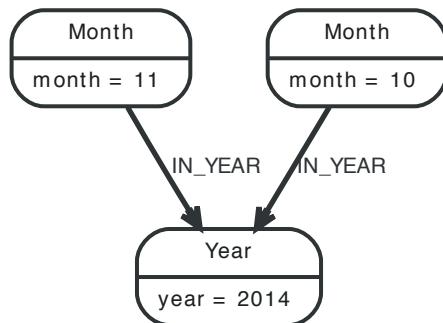


In case the direction of a relationship is arbitrary, you can leave off the arrowhead. `MERGE` will then check for the relationship in either direction, and create a new directed relationship if no matching relationship was found.

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:



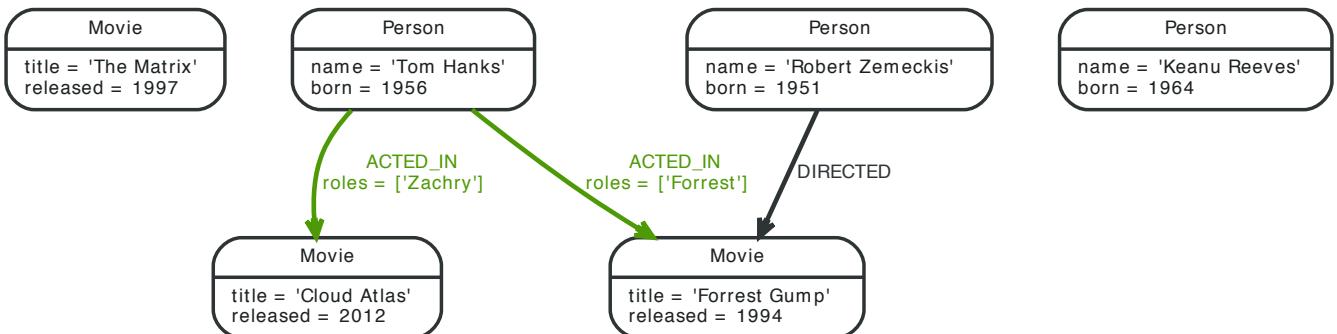
Here there is no global search for the two Month nodes; they are only searched for in the context of the 2014 Year node.

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 lose 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 its 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

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)
```

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 (actor:Person)-[r:ACTED_IN]->(movie:Movie)
RETURN actor.name AS name, type(r) AS acted_in, movie.title AS title
UNION
MATCH (director:Person)-[r:DIRECTED]->(movie:Movie)
RETURN director.name AS name, type(r) AS acted_in, movie.title AS title
```

name	acted_in	title
"Tom Hanks"	"ACTED_IN"	"Cloud Atlas"
"Tom Hanks"	"ACTED_IN"	"Forrest Gump"
"Robert Zemeckis"	"DIRECTED"	"Forrest Gump"

3 rows

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 *must* alias all columns as they would otherwise not be accessible. Only columns that you declare in your WITH clause is available in subsequent query parts.

See below for an example where we collect the movies someone appeared in, and then filter out those which appear in only one movie.

```
MATCH (person:Person)-[:ACTED_IN]->(m:Movie)
WITH person, count(*) AS appearances, collect(m.title) AS movies
WHERE appearances > 1
RETURN person.name, appearances, movies
```

person.name	appearances	movies
"Tom Hanks"	2	["Cloud Atlas", "Forrest Gump"]

1 row



Tip

If you want to filter by an aggregated value in SQL or similar 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. 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 Constraints

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 constraints and indexes to them. Let's start out adding a constraint — in this case we decided that all Movie node titles 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.

Using 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 index seek 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* [245]
- Section 14.2, “Constraints” [249]
- Section 14.1, “Indexes” [246]
- Section 10.8, “Using” [152]
- Section 12.3, “Set” [202]
- Section 12.5, “Remove” [207]

3.7. 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.

For more details, see [Section 11.6, “Load CSV” \[184\]](#).

movies.csv

id,title,country,year

Introduction to Cypher

```
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/2.3.1/csv/intro/movies.csv" AS line  
CREATE (m:Movie { id:line.id, title:line.title, released:toInt(line.year)});
```

persons.csv

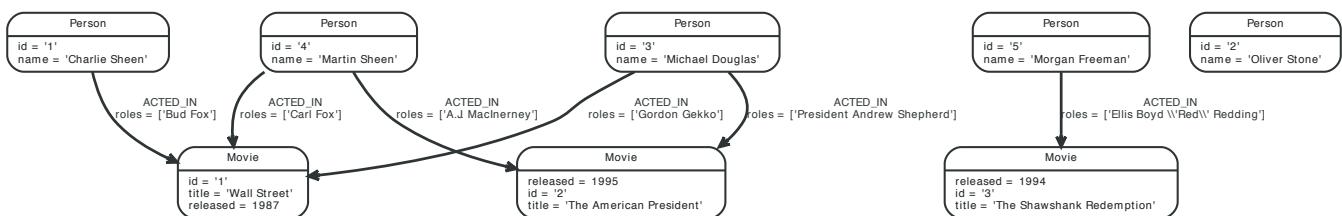
```
id,name  
1,Charlie Sheen  
2,Oliver Stone  
3,Michael Douglas  
4,Martin Sheen  
5,Morgan Freeman
```

```
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.3.1/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/2.3.1/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);
```



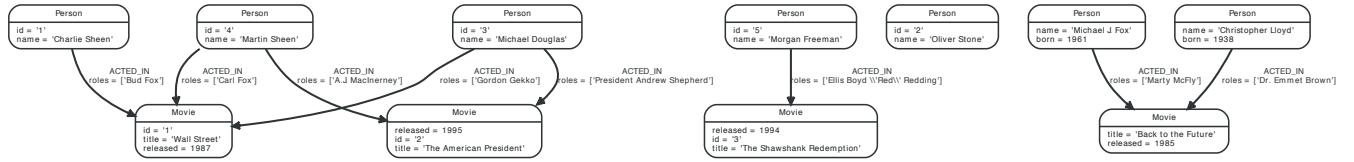
If your file contains denormalized data, you can either run the same file with multiple passes and simple operations as shown above or you might have to use MERGE to create entities uniquely.

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/2.3.1/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.

3.8. 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 are a number of functions that work with lists. They range from simple ones like `size(list)` that returns the size of a list to `reduce`, which runs an expression against the elements and accumulates the results.

Let's first load a bit of data into the graph. If you want more details on how the data is loaded, see [the section called "Importing CSV" \[34\]](#).

```
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.3.1/csv/intro/movies.csv" AS line
CREATE (m:Movie { id:line.id, title:line.title, released:toInt(line.year)});
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.3.1/csv/intro/persons.csv" AS line
MERGE (a:Person { id:line.id })
ON CREATE SET a.name=line.name;
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.3.1/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);
LOAD CSV WITH HEADERS FROM "http://neo4j.com/docs/2.3.1/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)
```

Now, let's try out data structures.

To begin with, collect the names of the actors per movie, and return two of them:

```
MATCH (movie:Movie)<--[:ACTED_IN]-(actor:Person)
RETURN movie.title AS movie, collect(actor.name)[0..2] AS two_of_cast
```

movie	two_of_cast
"The American President"	["Michael Douglas", "Martin Sheen"]
"Back to the Future"	["Christopher Lloyd", "Michael J. Fox"]
"Wall Street"	["Michael Douglas", "Martin Sheen"]
"The Shawshank Redemption"	["Morgan Freeman"]

4 rows

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 ANY (n IN nodes(path) WHERE n.name = 'Michael Douglas')
RETURN extract(n IN nodes(path)| coalesce(n.name, n.title))
```

extract(n IN nodes(path))	coalesce(n.name, n.title))
["Martin Sheen", "Wall S	et", "Michael Douglas"]
["Charlie Sheen", "Wall	eet", "Michael Douglas"]

6 rows

extract(n IN nodes(path	coalesce(n.name, n.title))
["Michael Douglas", "Wal	treet", "Martin Sheen"]
["Michael Douglas", "Wal	treet", "Charlie Sheen"]
["Martin Sheen", "The Am	can President", "Michael Douglas"]
["Michael Douglas", "The	erican President", "Martin Sheen"]
6 rows	

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, filter(actor IN cast WHERE actor.name STARTS WITH "M")
```

movie	filter(actor IN cast WHERE actor.name STARTS WITH "M")
"The American President"	[{name -> "Michael Douglas", roles -> ["President Andrew Shepherd"]}, {name -> "Martin Sheen", roles -> ["A. J. MacInerney"]}]
"Back to the Future"	[{name -> "Michael J. Fox", roles -> ["Marty McFly"]}]
"Wall Street"	[{name -> "Michael Douglas", roles -> ["Gordon Gekko"]}, {name -> "Martin Sheen", roles -> ["Carl Fox"]}]
"The Shawshank Redemption"	[{name -> "Morgan Freeman", roles -> ["Ellis Boyd 'Red' Redding"]}]
4 rows	

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 3 co-actors and then follow their movies and again list the cast for each of those movies:

```
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)<-[ :ACTED_IN]-(colleague:Person)
WHERE actor.name < colleague.name
```

Introduction to Cypher

```
WITH actor, colleague, count(*) AS frequency, collect(movie) AS movies
ORDER BY frequency DESC LIMIT 3 UNWIND movies AS m
MATCH (m)<-[ACTED_IN]-(a)
RETURN m.title AS movie, collect(a.name) AS cast
```

movie	cast
"The American President"	["Michael Douglas", "Martin Sheen"]
"Back to the Future"	["Christopher Lloyd", "Michael J. Fox"]
"Wall Street"	["Michael Douglas", "Martin Sheen", "Charlie Sheen", "Michael Douglas", "Martin Sheen", "Charlie Sheen"]

3 rows

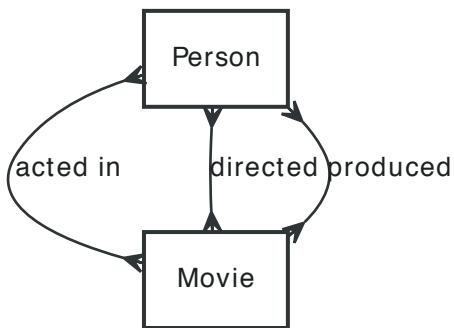
3.9. Cypher vs. SQL

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" \[168\]](#) 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"
"Al Pacino"	"The Devil's Advocate"

8 rows

person.name	movie.title
"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” \[300\]](#) 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 \[62\]](#).

The examples use Cypher queries a lot, read [Part III, “Cypher Query Language” \[102\]](#) for more information.

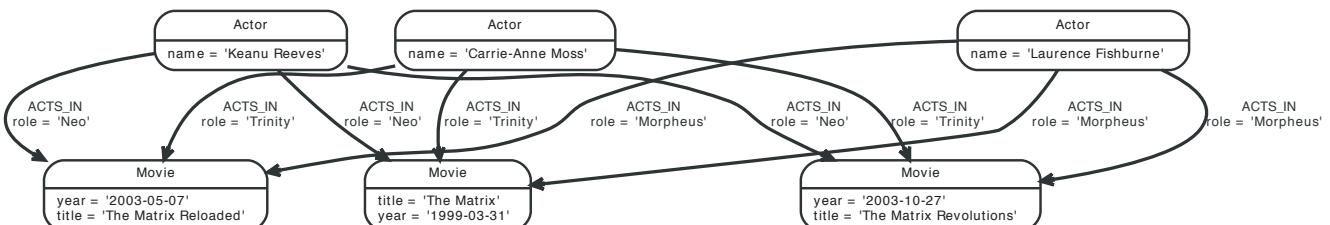
5.1. Movie Database

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]{year:"2003-10-27", title:"The Matrix Revolutions"}
Node[3]{name:"Keanu Reeves"}	:ACTS_IN[1]{role:"Neo"}	Node[1]{year:"2003-05-07", title:"The Matrix Reloaded"}
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]{year:"2003-10-27", title:"The Matrix Revolutions"}
Node[4]{name:"Laurence Fishburne"}	:ACTS_IN[4]{role:"Morpheus"}	Node[1]{year:"2003-05-07", title:"The Matrix Reloaded"}
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]{year:"2003-10-27", title:"The Matrix Revolutions"}
Node[5]{name:"Carrie-Anne Moss"}	:ACTS_IN[7]{role:"Trinity"}	Node[1]{year:"2003-05-07", title:"The Matrix Reloaded"}
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"	"Friend 5"
"FRIEND"	"Friend 4"
"FRIEND"	"Friend 3"
"FRIEND"	"Friend 2"
"FRIEND"	"Friend 1"
"FRIEND"	"Friend 10"
"FRIEND"	"Friend 8"
"FRIEND"	"Friend 9"
"FRIEND"	"Friend 6"
"FRIEND"	"Friend 7"
"FRIEND"	"A Friend"

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

9 rows

names and titles	length(p)
["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.

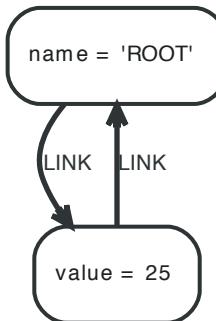
To initialize an empty linked list, we simply create a node, and make it link to itself. Unlike the actual list elements, it doesn't have a value property.

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



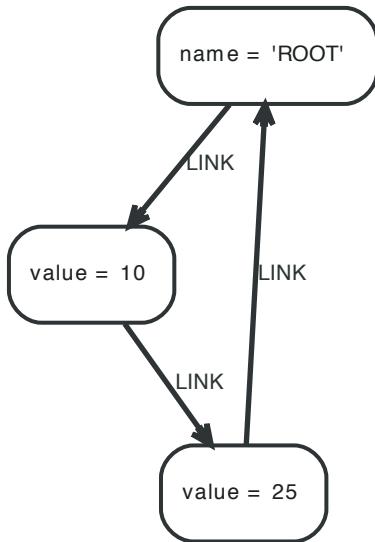
Adding values is done by finding the relationship where the new value should be placed in, and replacing it with a new node, and two relationships to it. We also have to handle the fact that the before and after nodes could be the same as the root node. The case where before, after and the root node 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:

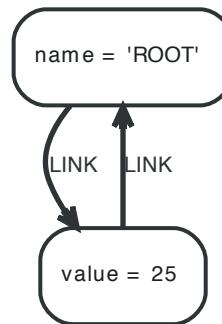
```
MATCH (root)-[:LINK*0..]->(before),(after)-[:LINK*0..]->(root),(before)-[old:LINK]->(after)
WHERE root.name = 'ROOT' AND (before.value < 10 OR before = root) AND (10 < after.value OR after =
root)
CREATE UNIQUE (before)-[:LINK]->({ value:10 })-[:LINK]->(after)
DELETE old
```



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
  
```

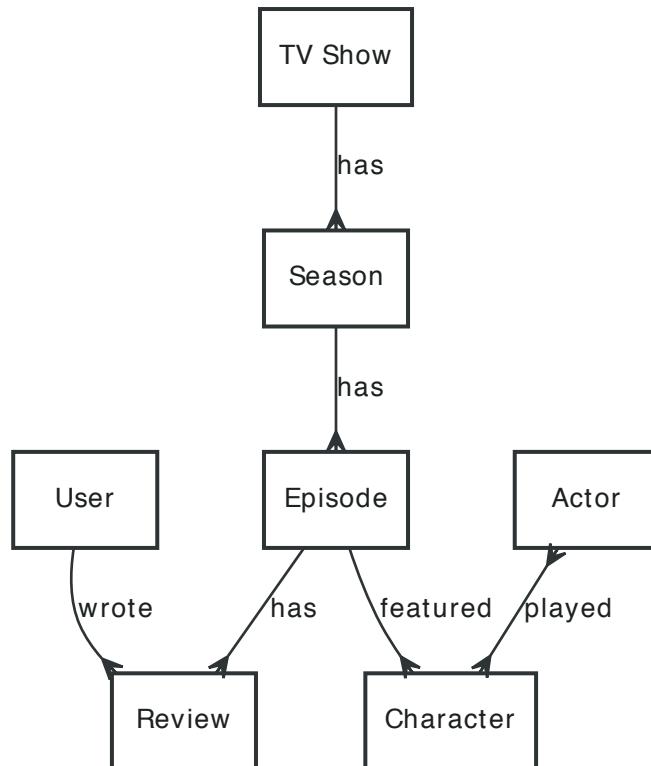


5.5. TV Shows

This example shows 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.
WRITED_REVIEW	Connects users with reviews they contributed.

Sample Data

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

```

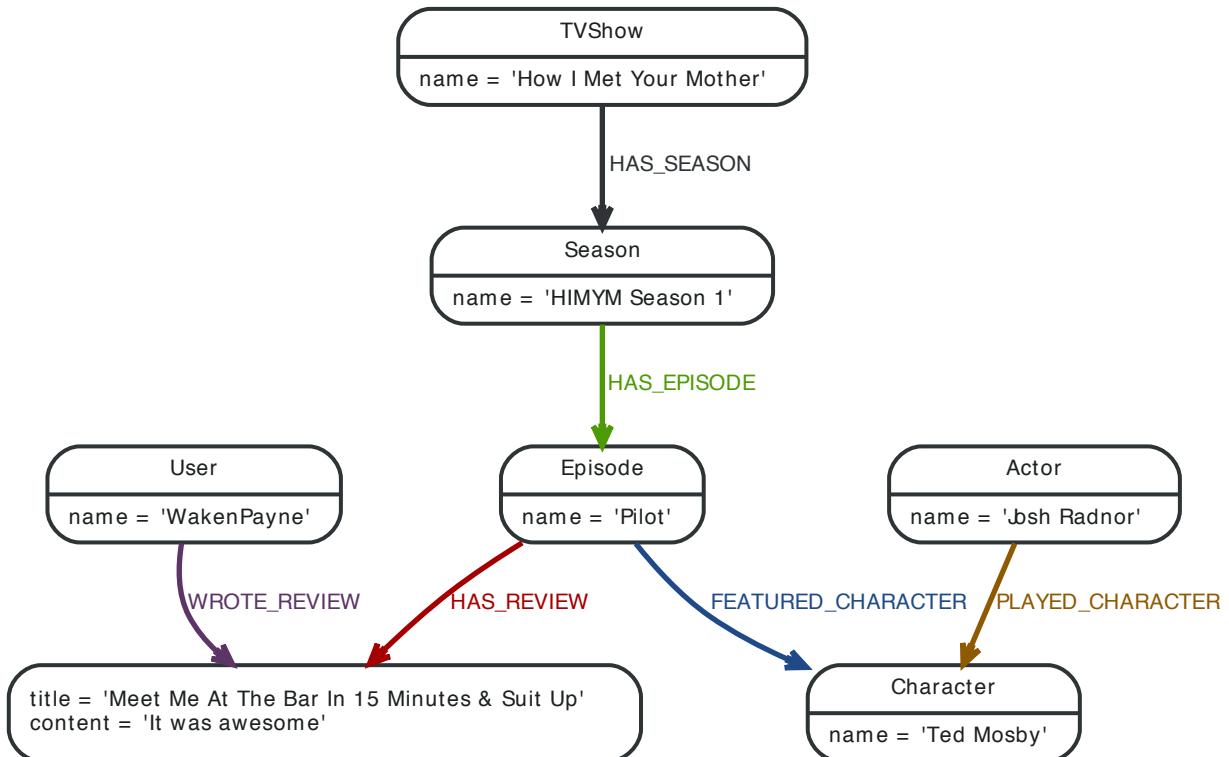
CREATE (himym:TVShow { name: "How I Met Your Mother" })
CREATE (himym_s1:Season { name: "HIMYM Season 1" })
CREATE (himym_s1_e1:Episode { name: "Pilot" })
CREATE (ted:Character { name: "Ted Mosby" })
CREATE (joshRadnor:Actor { name: "Josh Radnor" })
  
```

```

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[15]{title:"What a great pilot for a show :)", content:"The humour is great."}
"HIMYM Season 1"	"Pilot"	Node[5]{title:"Meet Me At The Bar In 15 Minutes & Suit Up", content:"It was awesome"}
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
"Lily Aldrin"
"Barney Stinson"
"Robin Scherbatsky"
"Marshall Eriksen"
"Ted Mosby"
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

"Alyson Hannigan"

"Neil Patrick Harris"

"Cobie Smulders"

"Jason Segel"

"Josh Radnor"

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

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

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” \[102\]](#) for more information.

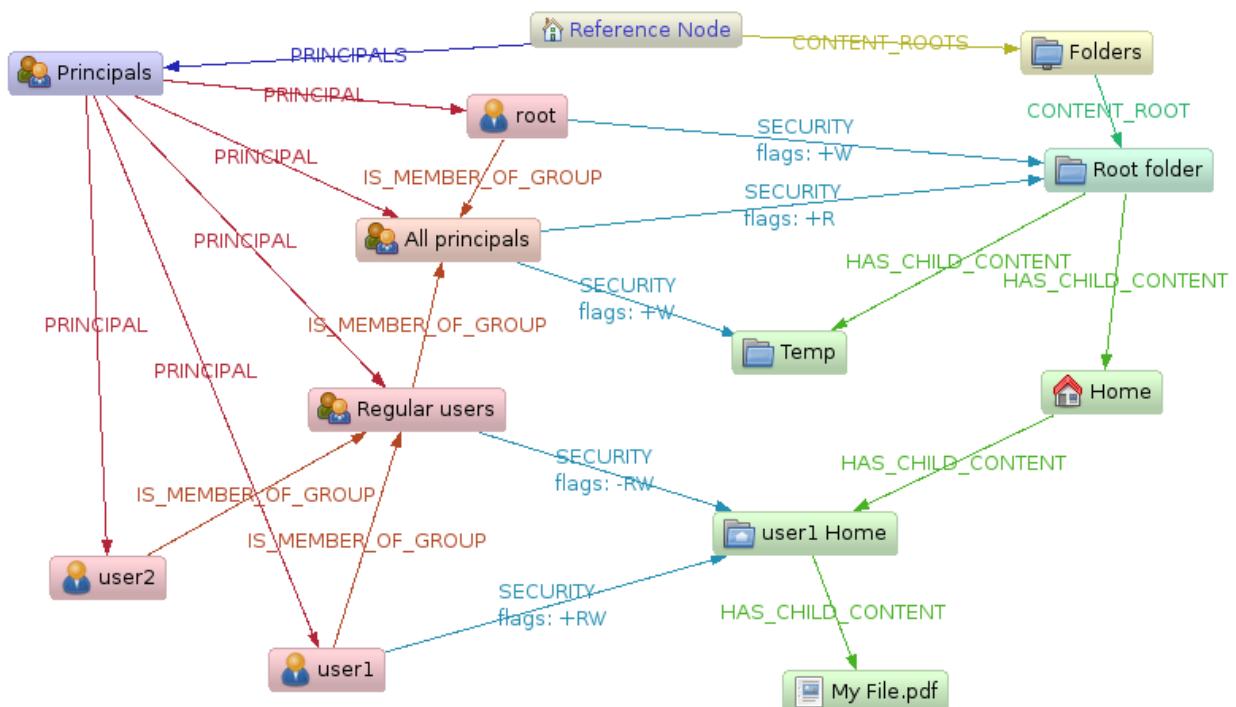
6.1. ACL structures in graphs

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 principals 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

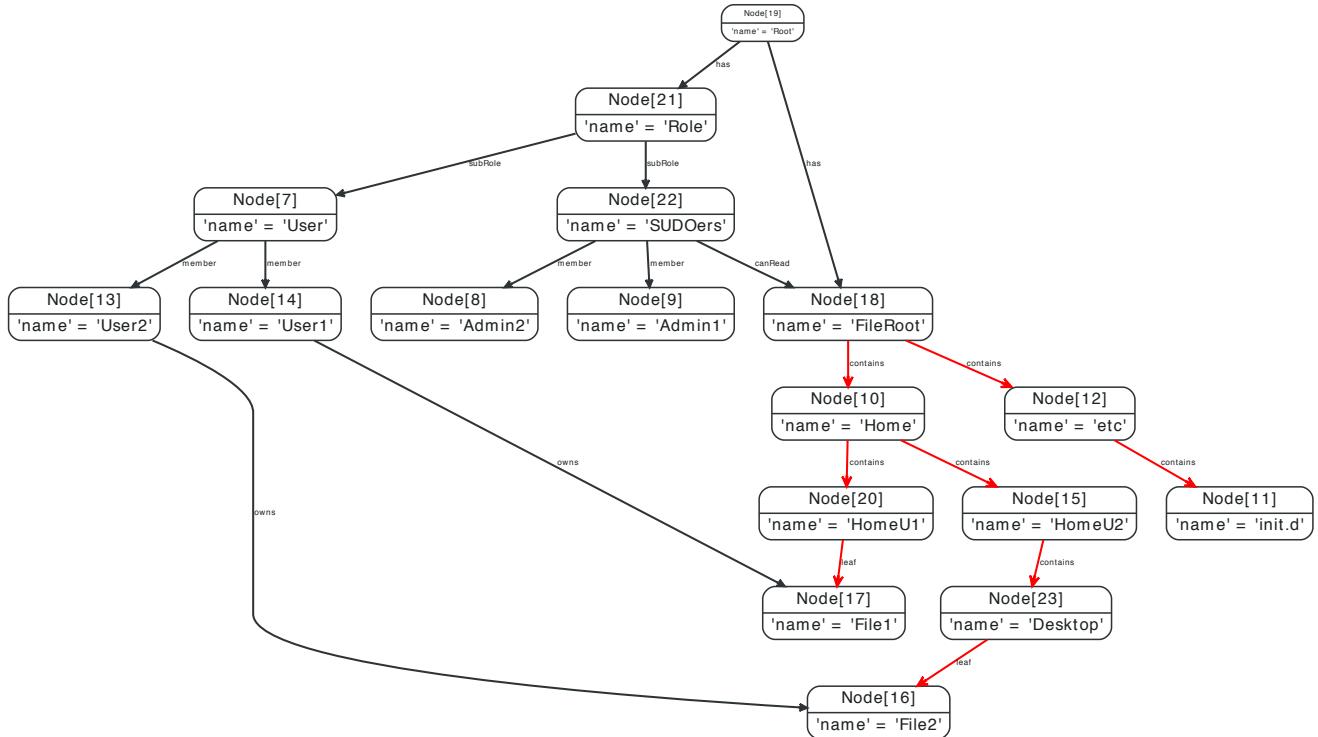
Now, to get the resulting access rights for e.g. "user 1" on the "My File.pdf" in a Top-Down approach on the model in the graph above would go like:

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 (n {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 (n {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
"File2"	"Desktop"	<null>	<null>
"File2"	"HomeU2"	<null>	<null>
"File2"	"Home"	<null>	<null>
"File2"	"FileRoot"	"SUDOers"	"Admin2"
"File2"	"FileRoot"	"SUDOers"	"Admin1"
"File1"	"HomeU1"	<null>	<null>
"File1"	"Home"	<null>	<null>
"File1"	"FileRoot"	"SUDOers"	"Admin2"
"File1"	"FileRoot"	"SUDOers"	"Admin1"

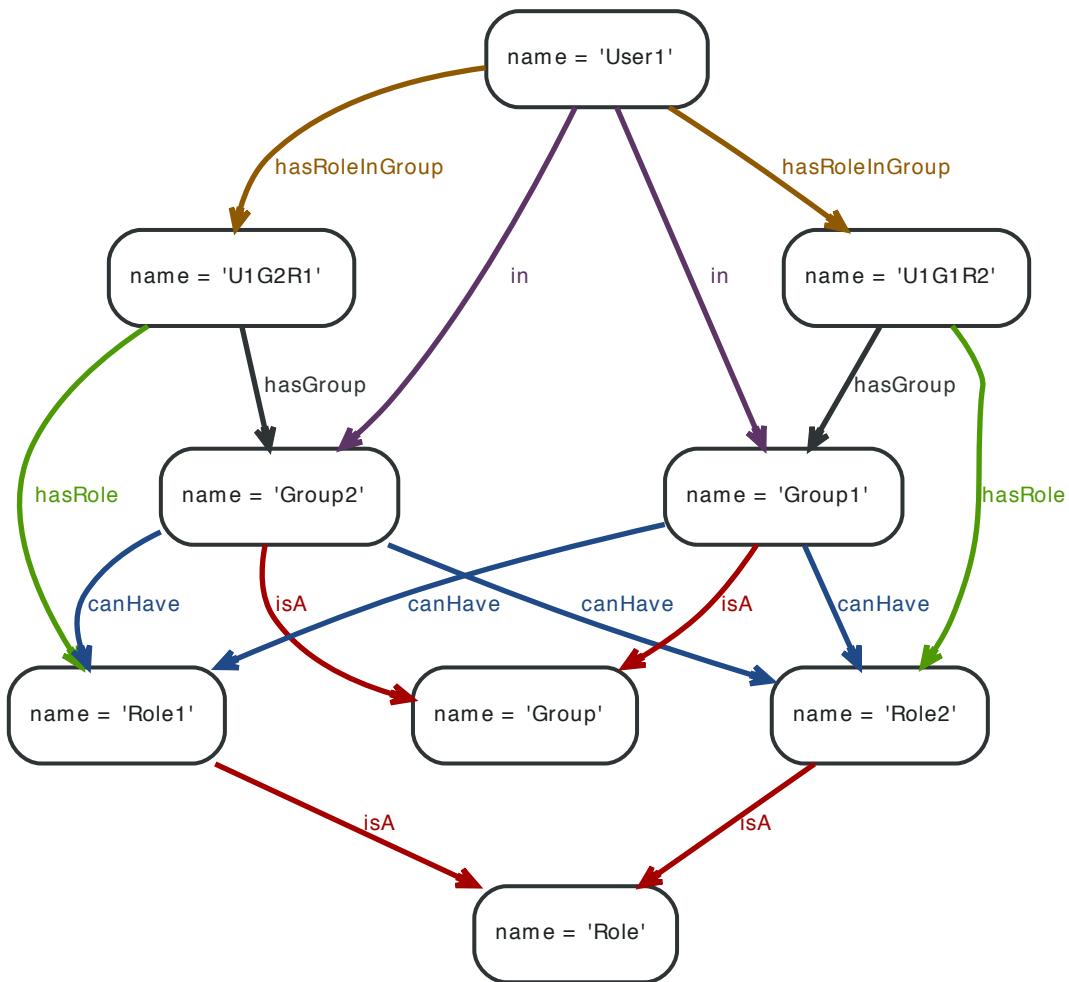
9 rows

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 (u { name: 'User1' })-[:hasRoleInGroup]->(hyperEdge)-[:hasGroup]->(g { 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 (u {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

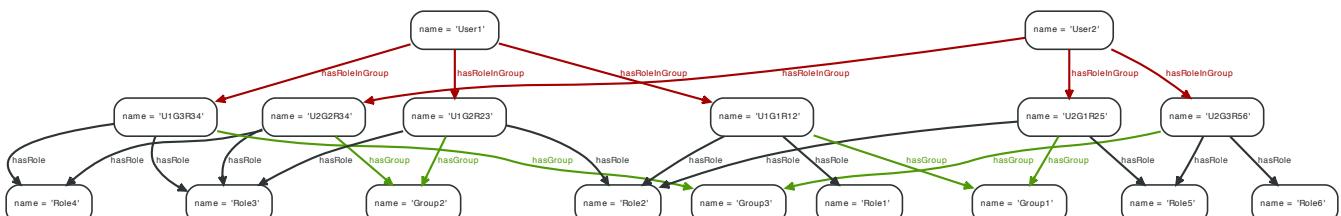
Find common groups based on shared roles

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

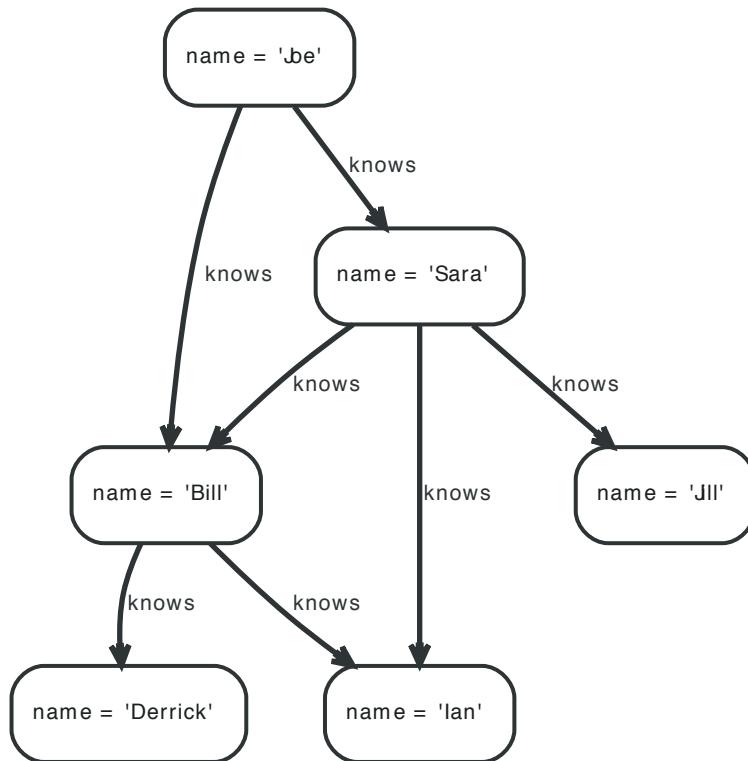
group.name	count(role)
"Group1"	1
"Group2"	1

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.

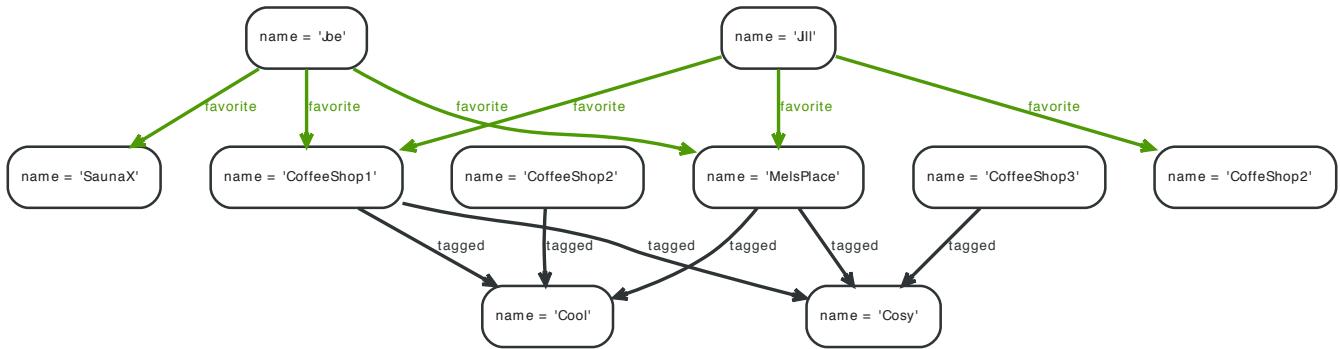
Result

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"]

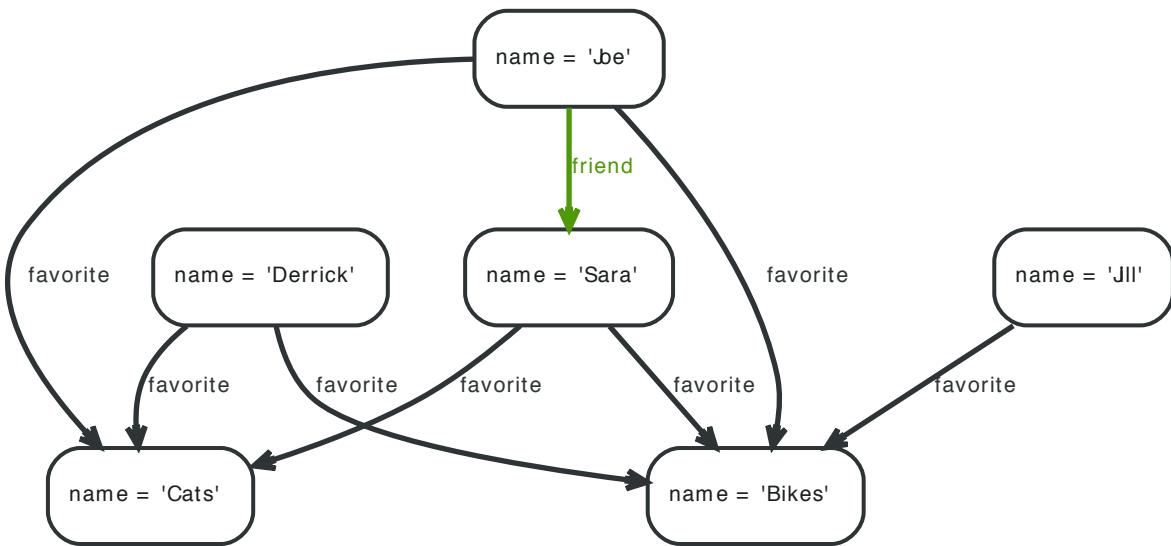
3 rows

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
  
```

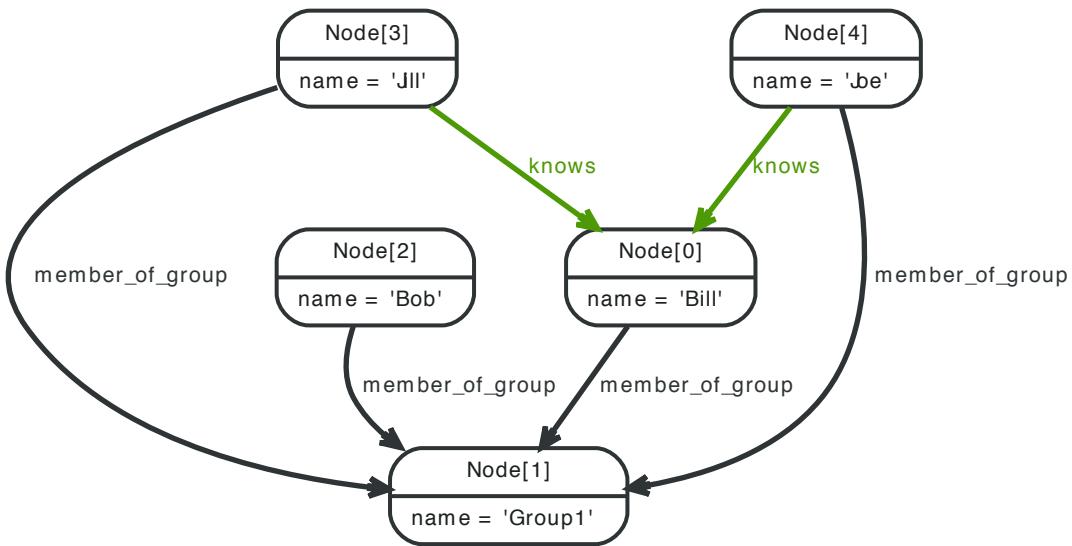
The list of possible friends ranked by them liking similar stuff that are not yet friends is returned.

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 \emptyset 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 its relationships optional. That makes the whole path optional.

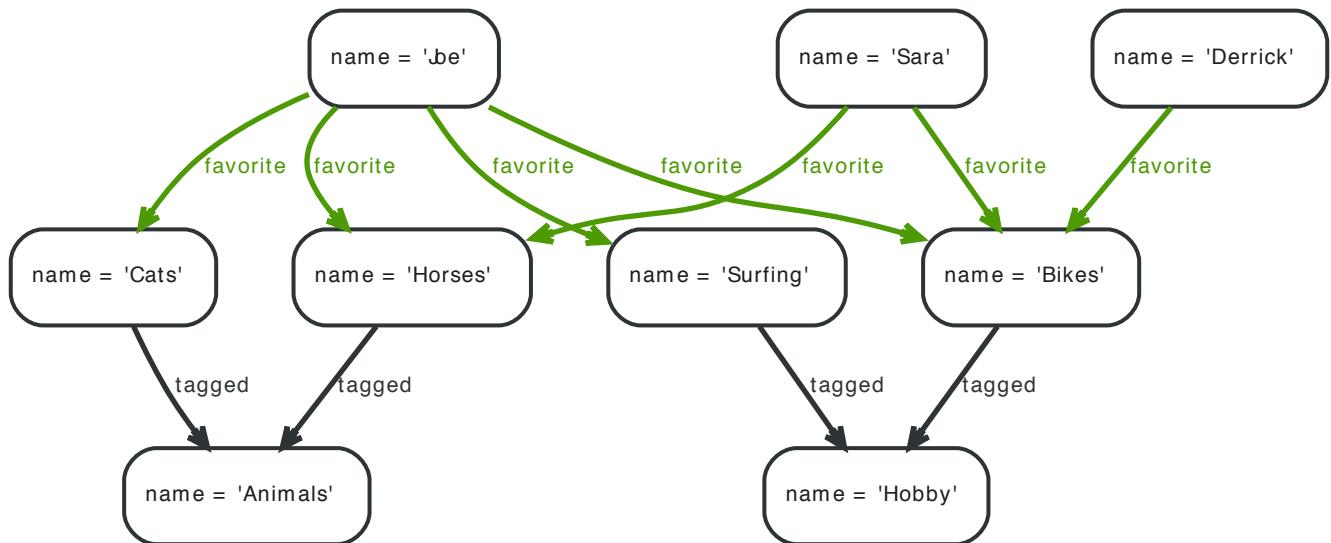
Result

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
  
```

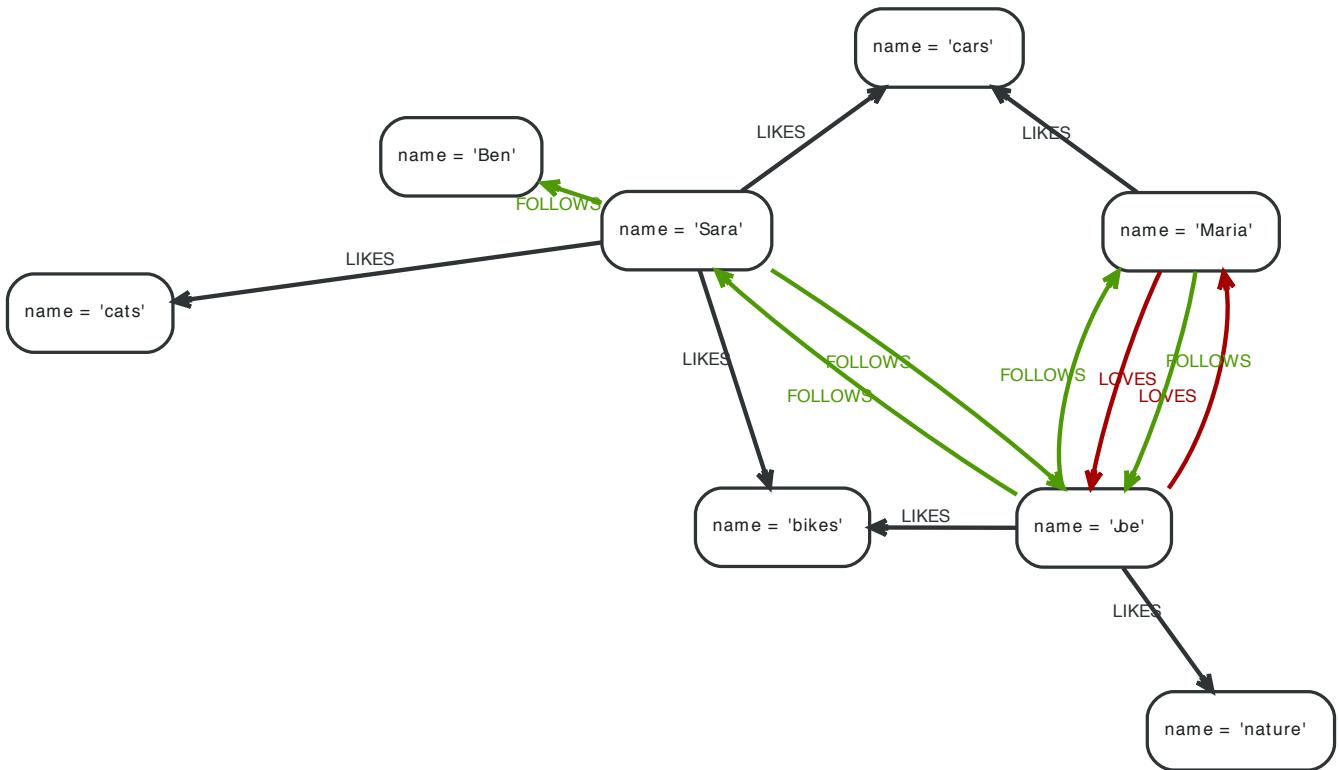
The query returns the list of possible friends ranked by them liking similar stuff that are not yet friends.

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)
  
```

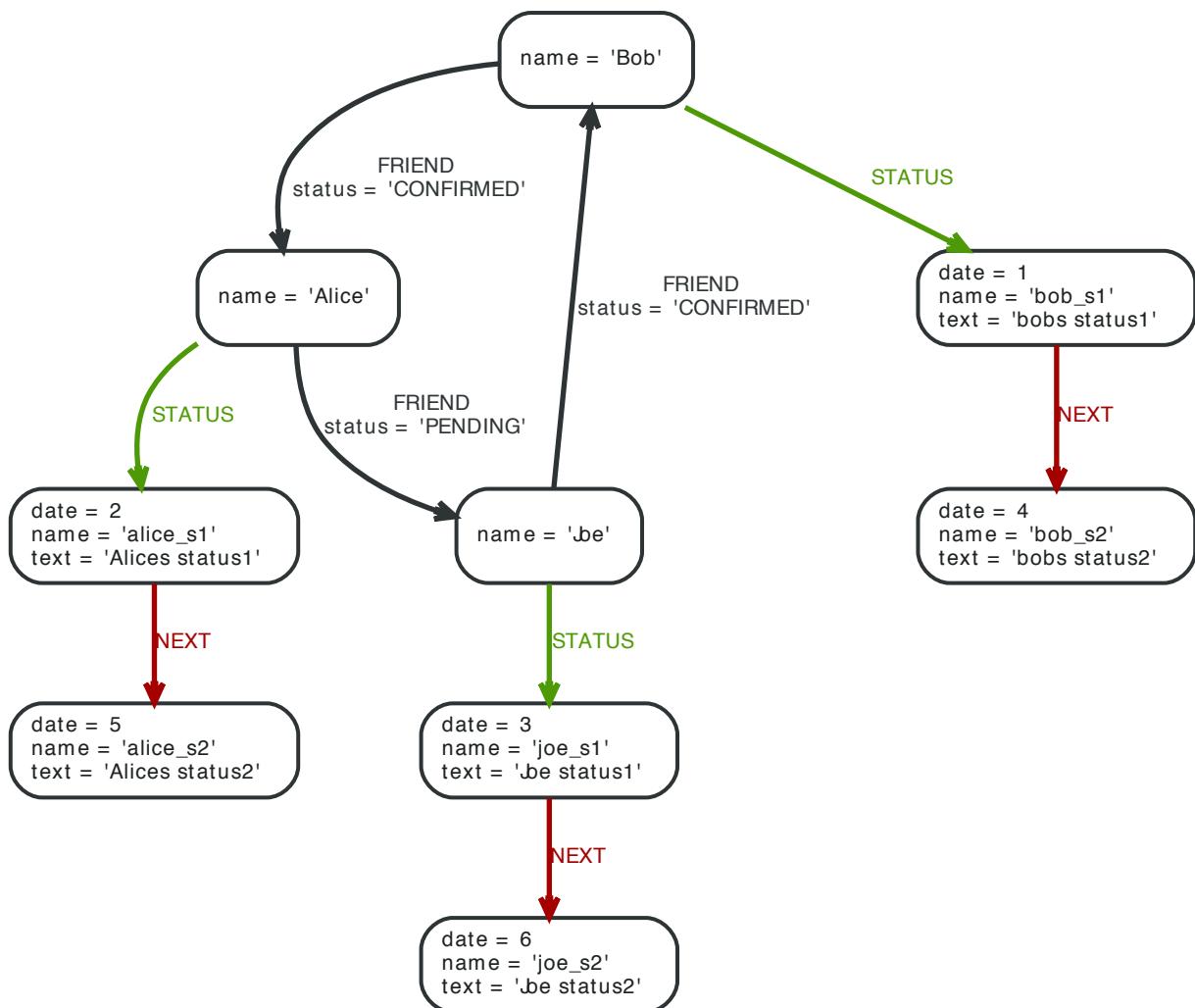
The query returns people that FOLLOWS or LOVES Joe back.

Result

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

3 rows

6.9. Implementing newsfeeds in a graph



Implementation of newsfeed or timeline feature is a frequent requirement for social applications. The following examples 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 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:

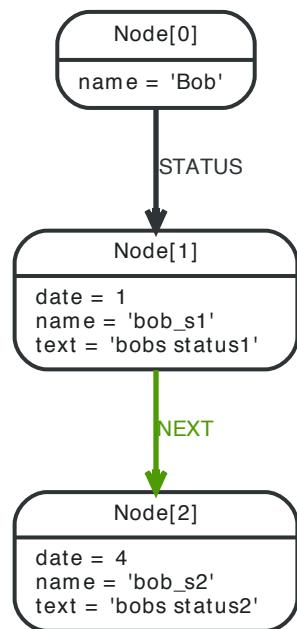
1. Get the latest update (if it exists) of the user through the STATUS relationship (OPTIONAL MATCH (me)-[r:STATUS]-(secondlatestupdate)).
2. Delete the STATUS relationship between user and secondlatestupdate (if it exists), as this would become the second latest update now and only the latest update 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

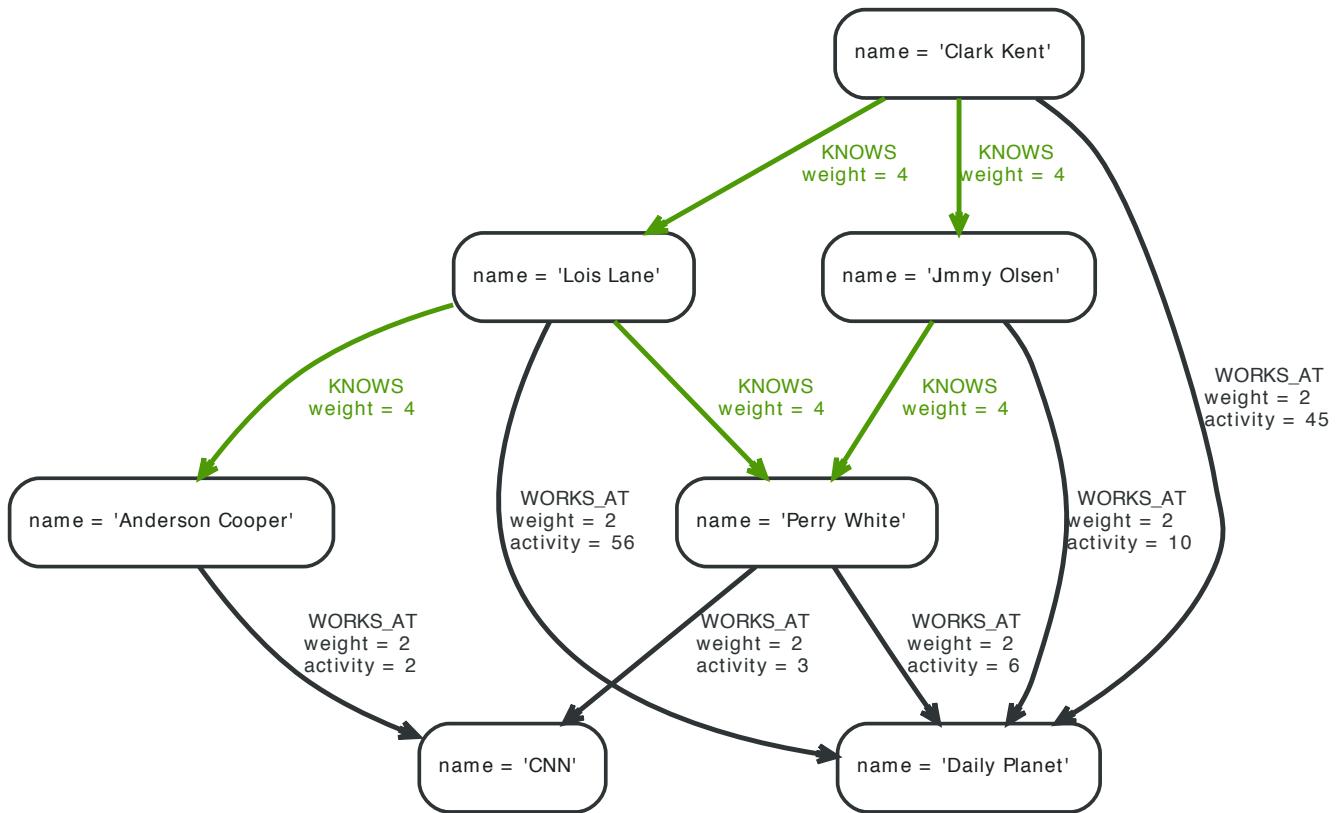
"Status"

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



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 r_2 , 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.

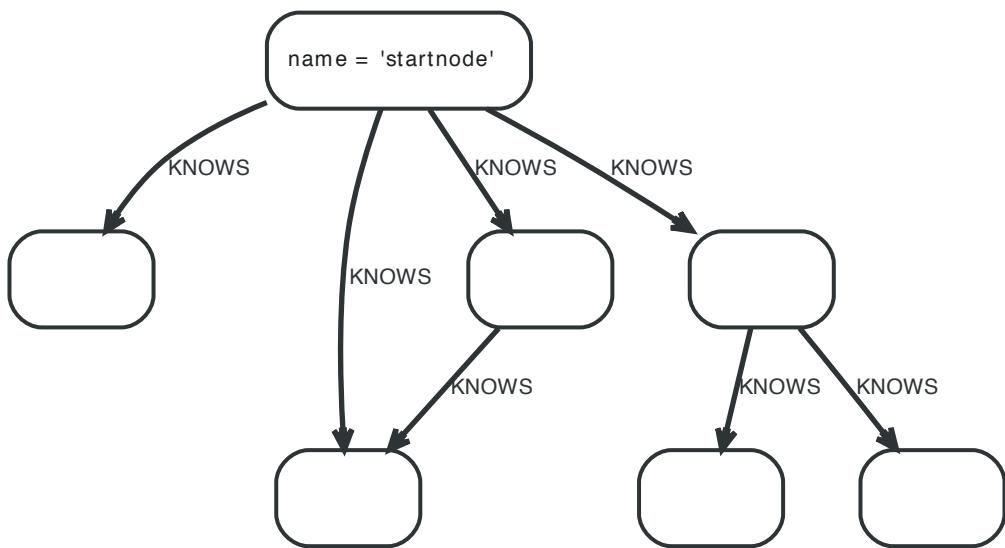
Result

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 Analysis³](#), the graph in question is showing the 2-hop relationships of a sample person as nodes with KNOWS relationships.

The [clustering coefficient⁴](#) 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.

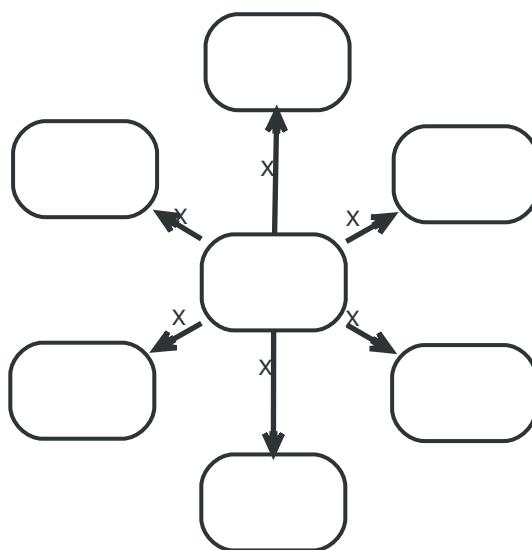
Result

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.

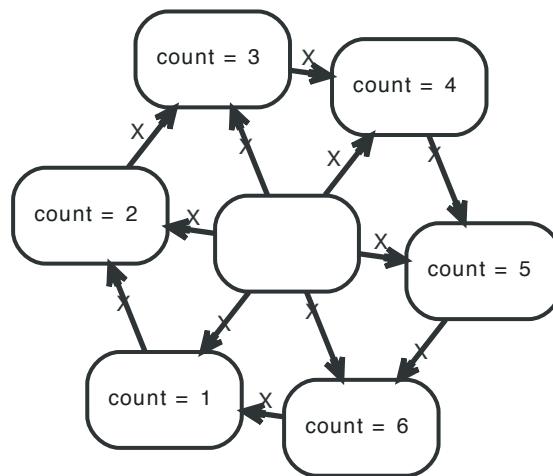
Result

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);
    
```

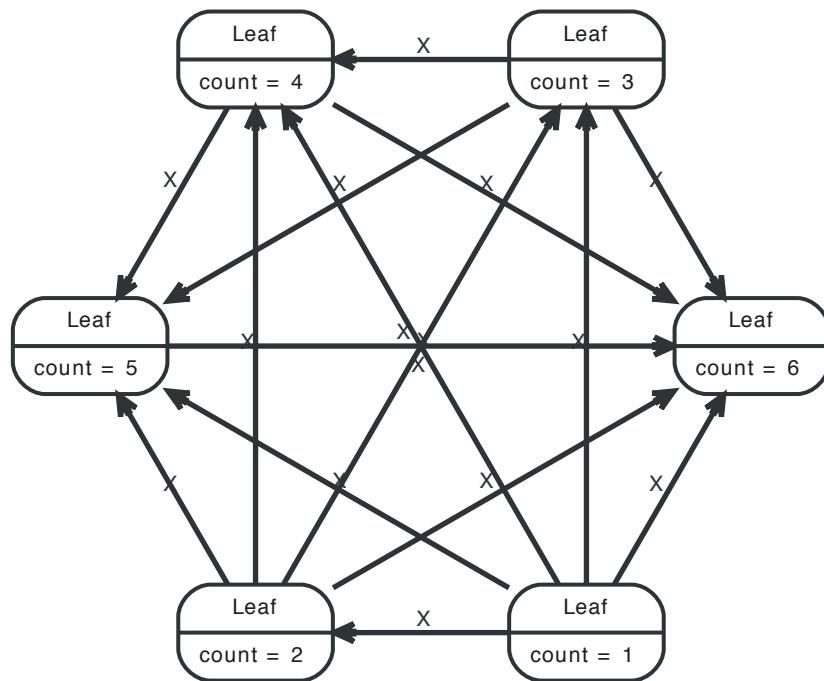
Nothing is returned by this query.

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

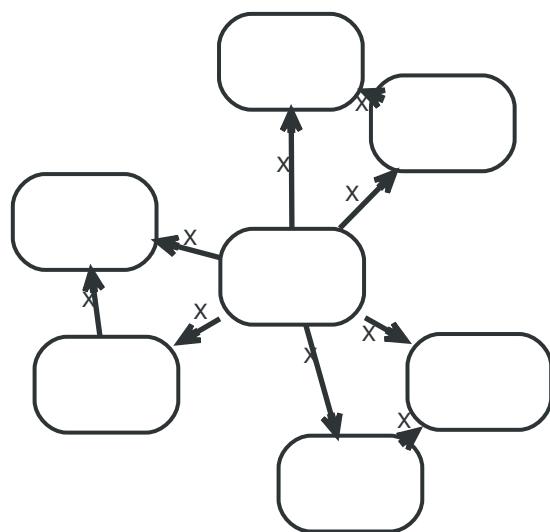
0

1 row

Nodes created: 7

Relationships created: 9

Figure 6.14. Graph



6.13. A multilevel indexing structure (path tree)

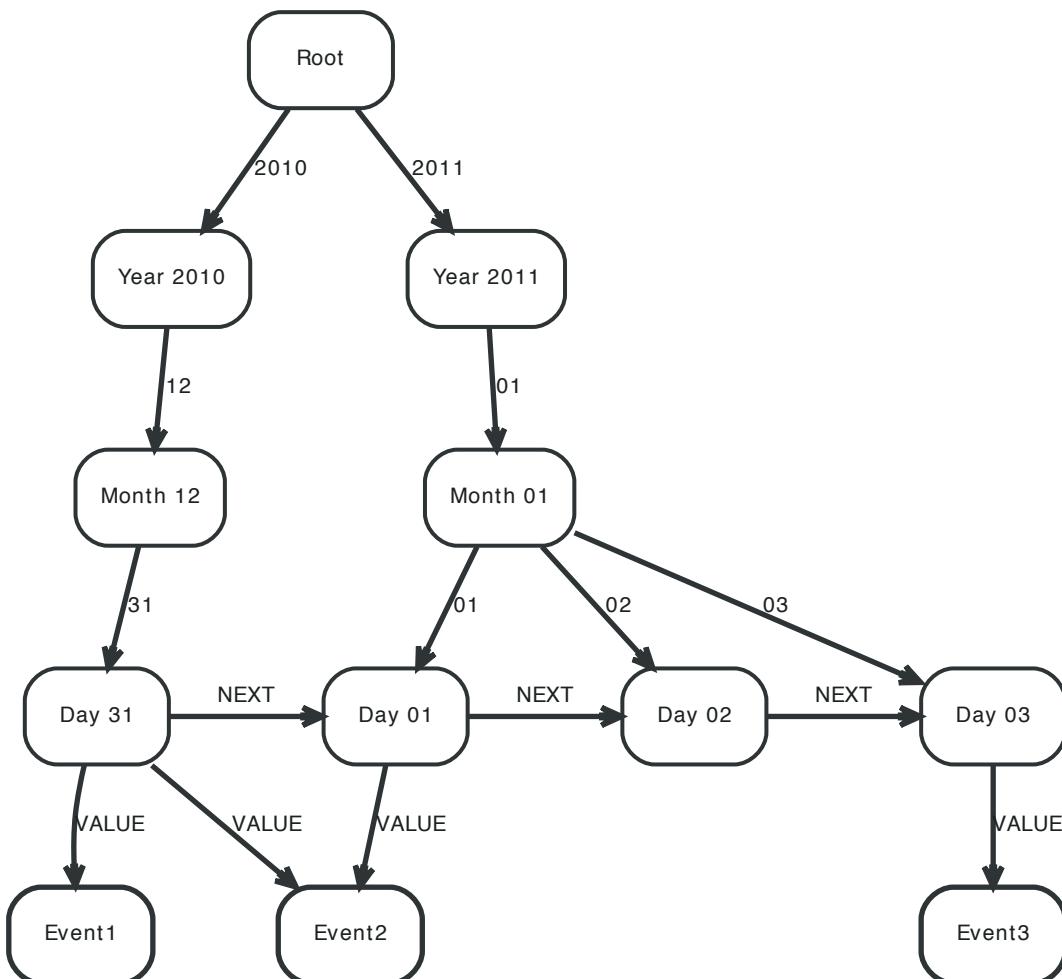
In this example, a multi-level tree structure is used to index event nodes (here Event1, Event2 and Event3, in this case with a YEAR-MONTH-DAY granularity, making this a timeline indexing structure. However, this approach should work for a wide range of multi-level ranges.

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.

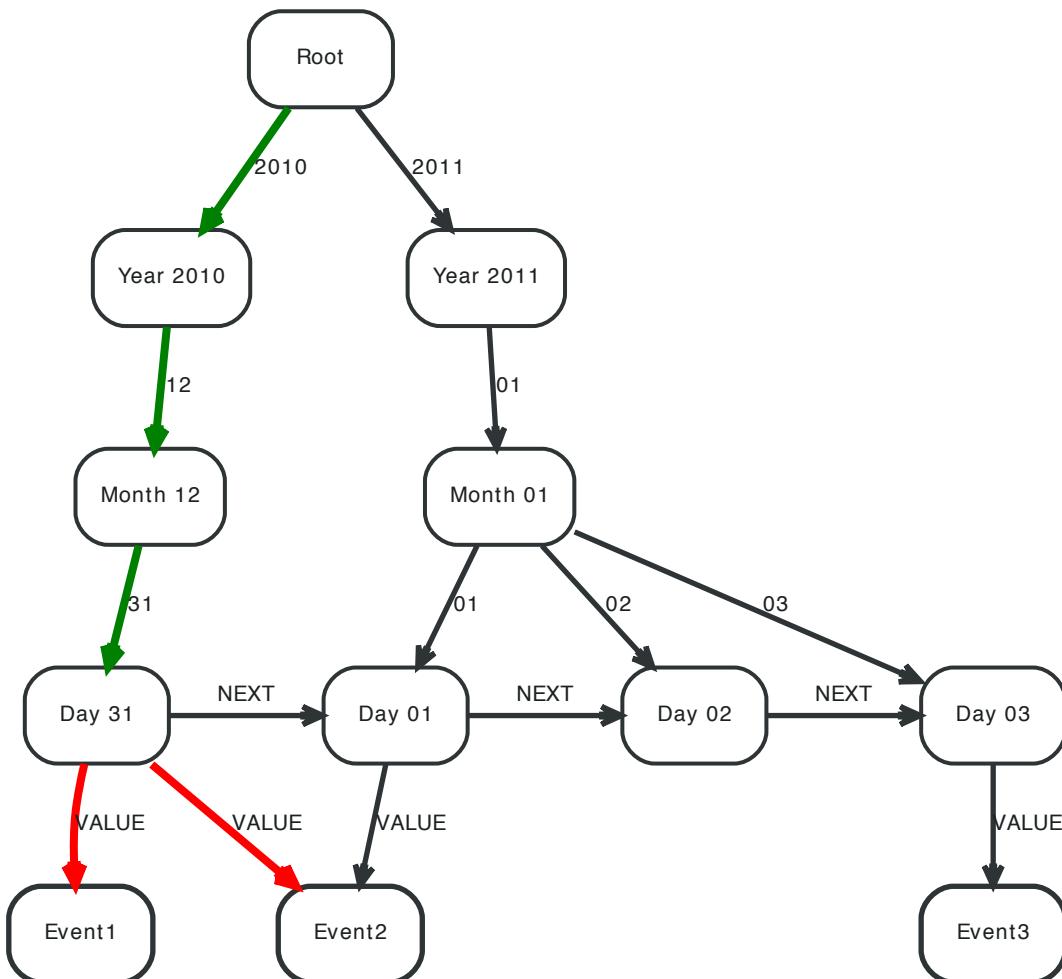
Figure 6.15. Graph



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"

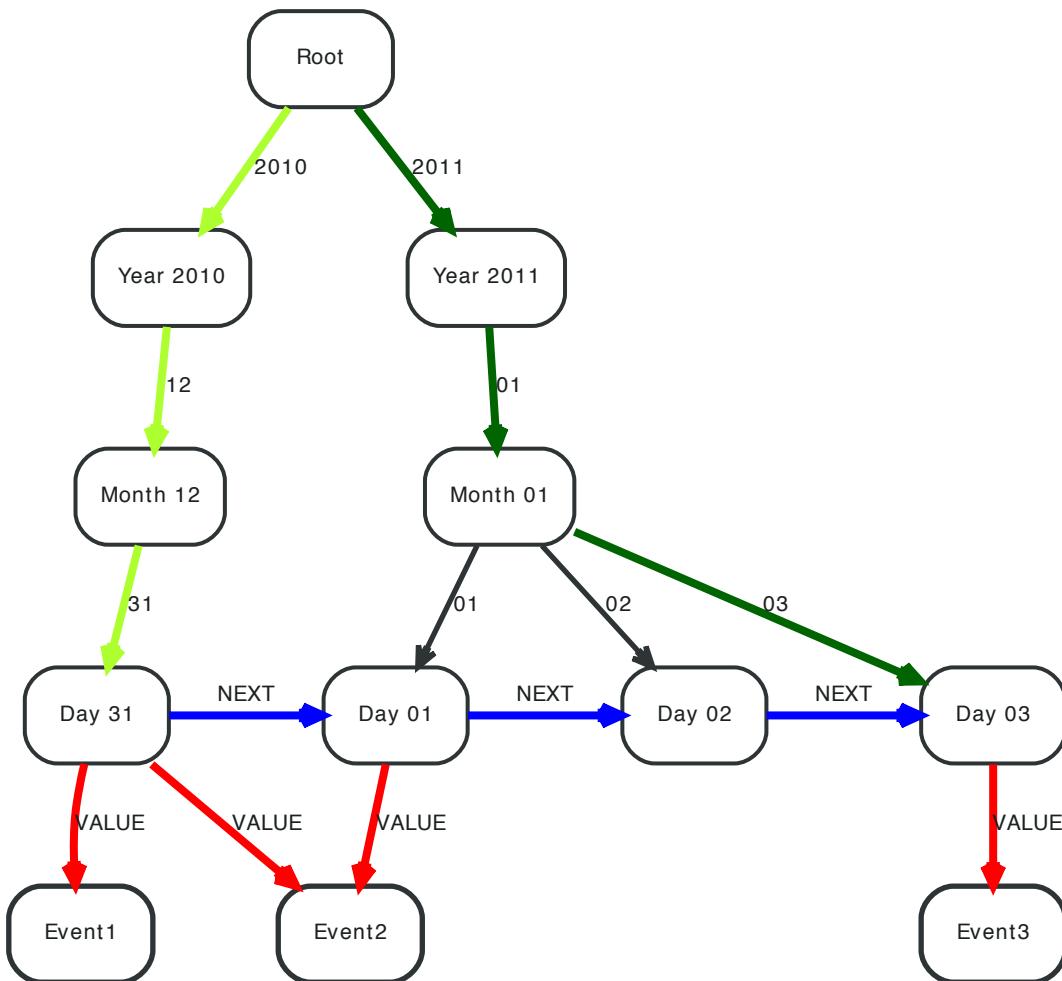
"Event2"

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

```

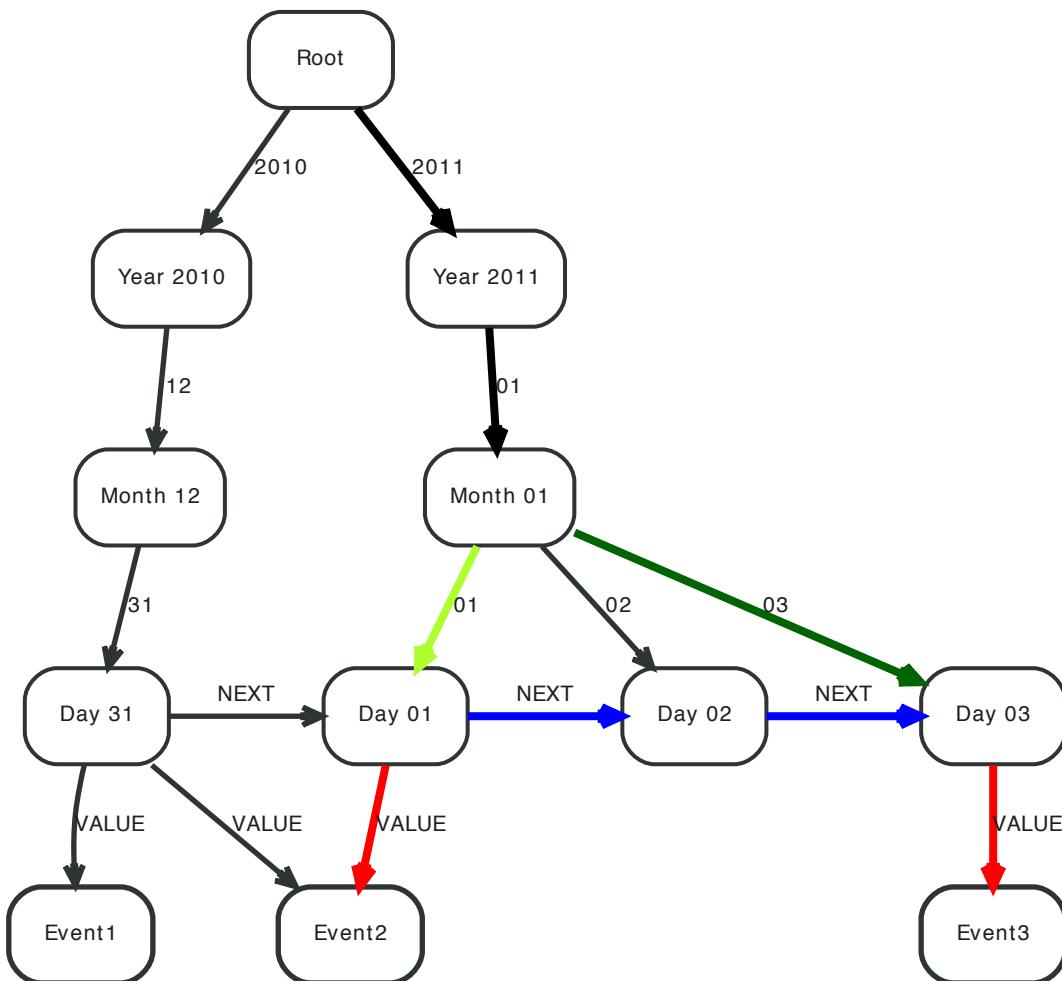
"Event1"
"Event2"
"Event2"
"Event3"
  
```

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.

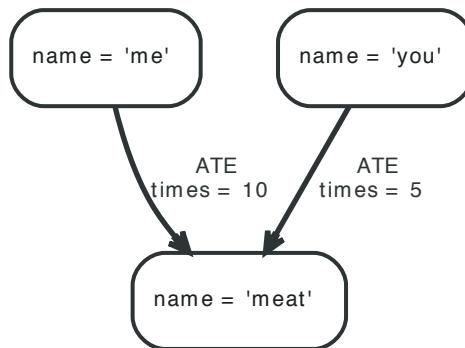
Result

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 occurred. 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" \[286\]](#) 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 from 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.

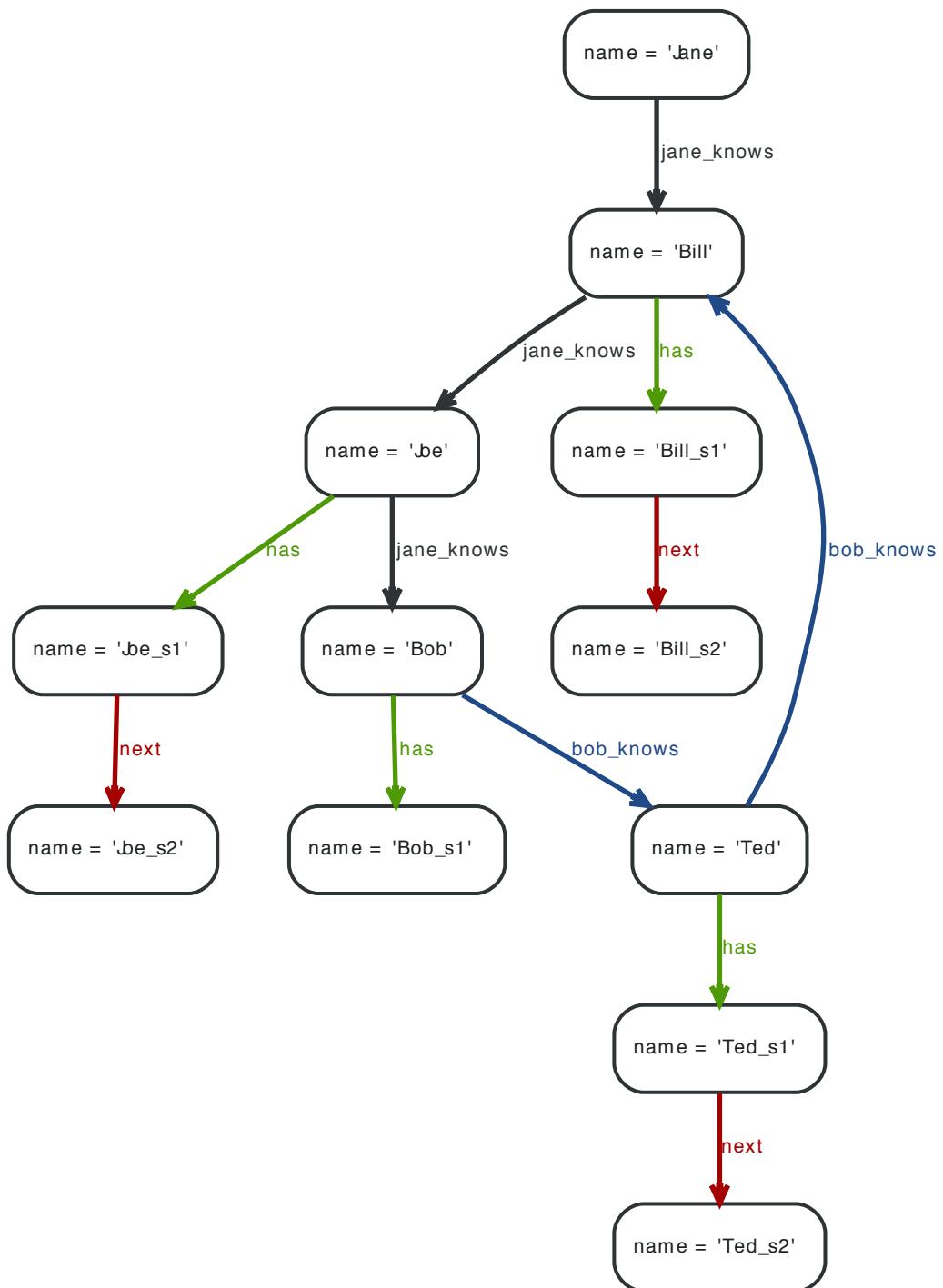
Result

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

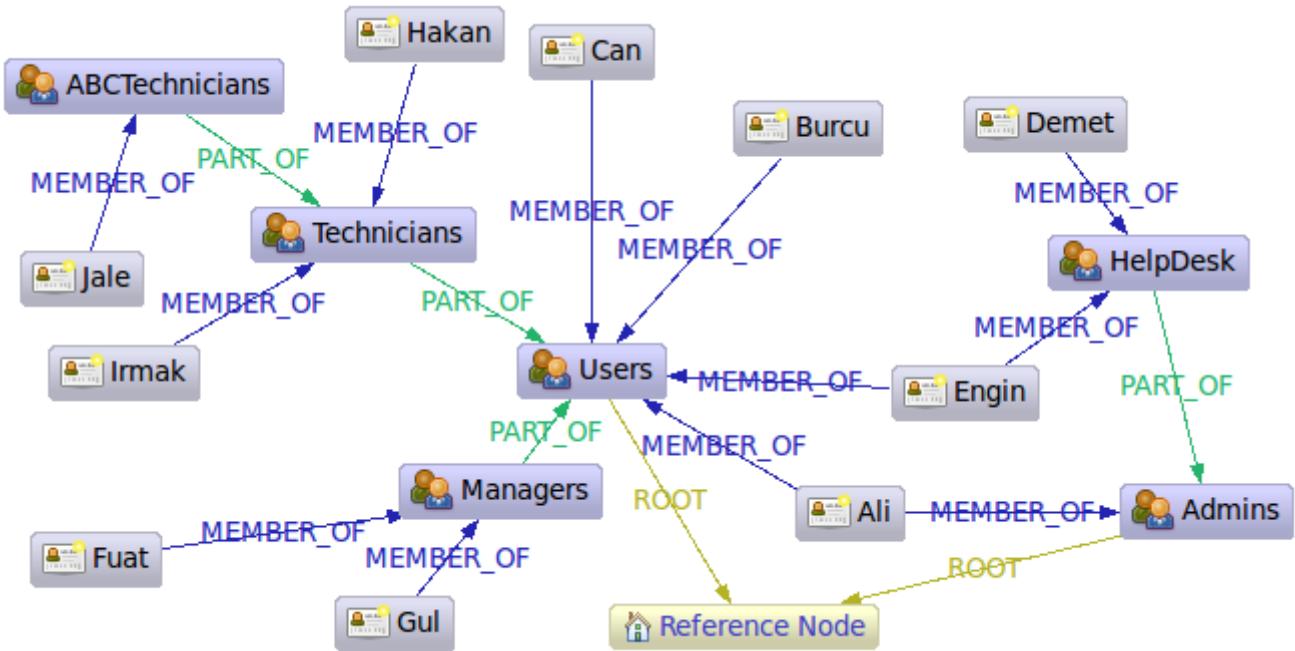
⁶ <http://www.rene-pickhardt.de/graphity-an-efficient-graph-model-for-retrieving-the-top-k-news-feeds-for-users-in-social-networks/>

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 34.2, "Traversal Framework Java API"](#) [613], which is part of [Part VII, "Advanced Usage"](#) [560]).

Get the admins

In Cypher, we could get the admins like this:

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

```
MATCH (n { 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: Ali at depth: 0
Found: HelpDesk 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 (n { 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 (n { 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 (n { 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
"Burcu"	1
10 rows	

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

10 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 getting 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” \[440\]](#). 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” \[436\]](#).



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.lineSeparator() + "%s",
    payload, txUri, response.getStatus(),
    response.getEntity( String.class ) ) );

response.close();
```

For more details, see [Section 21.1, “Transactional Cypher HTTP endpoint” \[300\]](#).

Fine-grained REST API calls

For exploratory and special purposes, there is a fine grained REST API, see [Chapter 21, REST API \[299\]](#). The following sections highlight some of the basic operations.

¹ <http://jersey.java.net/>

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

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 database, 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.lineSeparator() + "%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.java³](#)
- [Relation.java⁴](#)
- [TraversalDefinition.java⁵](#)

³ <https://github.com/neo4j/neo4j/blob/2.3.1/community/server-examples/src/main/java/org/neo4j/examples/server/CreateSimpleGraph.java>

⁴ <https://github.com/neo4j/neo4j/blob/2.3.1/community/server-examples/src/main/java/org/neo4j/examples/server/Relation.java>

⁵ <https://github.com/neo4j/neo4j/blob/2.3.1/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?” \[106\]](#). To take your first steps with Cypher, see [Chapter 3, *Introduction to Cypher* \[16\]](#). For the terminology used, see [Terminology \[638\]](#).

8. Introduction	105
8.1. What is Cypher?	106
8.2. Updating the graph	109
8.3. Transactions	110
8.4. Uniqueness	111
8.5. Parameters	113
8.6. Compatibility	117
9. Syntax	118
9.1. Values	119
9.2. Expressions	120
9.3. Identifiers	123
9.4. Operators	124
9.5. Comments	126
9.6. Patterns	127
9.7. Collections	131
9.8. Working with NULL	134
10. General Clauses	136
10.1. Return	137
10.2. Order by	140
10.3. Limit	142
10.4. Skip	144
10.5. With	146
10.6. Unwind	148
10.7. Union	150
10.8. Using	152
11. Reading Clauses	156
11.1. Match	157
11.2. Optional Match	166
11.3. Where	168
11.4. Start	176
11.5. Aggregation	178
11.6. Load CSV	184
12. Writing Clauses	188
12.1. Create	189
12.2. Merge	194
12.3. Set	202
12.4. Delete	206
12.5. Remove	207
12.6. Foreach	209
12.7. Create Unique	210
12.8. Importing CSV files with Cypher	213
12.9. Using Periodic Commit	215
13. Functions	216
13.1. Predicates	217
13.2. Scalar functions	220
13.3. Collection functions	226
13.4. Mathematical functions	231
13.5. String functions	240
14. Schema	245
14.1. Indexes	246
14.2. Constraints	249
14.3. Statistics	254
15. Query Tuning	255
15.1. How are queries executed?	256
15.2. How do I profile a query?	257
15.3. Basic query tuning example	258

16. Execution Plans	261
16.1. Starting point operators	262
16.2. Expand operators	265
16.3. Combining operators	267
16.4. Row operators	272
16.5. Update Operators	277

Chapter 8. Introduction

To get an overview of Cypher, continue reading [Section 8.1, “What is Cypher?” \[106\]](#). 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” \[102\]](#). To take your first steps with Cypher, see [Chapter 3, *Introduction to Cypher* \[16\]](#). For the terminology used, see [Terminology \[638\]](#).

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 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 its 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:

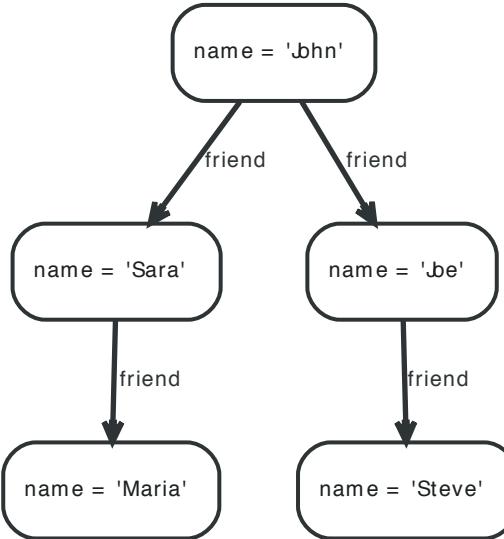
¹ <http://gremlin.tinkerpop.com>

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 \[47\]](#) as well as the rest of the Cypher part with details on the language. To use Cypher from Java, see [Section 33.14, "Execute](#)

Cypher Queries from Java” [607]. 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

- 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 part.

If your query only performs reads, 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.

The only pattern where the query parts are implicit is when you first read and then write — any other order and you have to be explicit about your query parts. The parts are separated using the WITH statement. WITH is like an 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 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.

Here's an example of updating the graph, writing the aggregated data to the graph:

```
MATCH (n {name: 'John'})-[:FRIEND]-(friend)
WITH n, count(friend) as friendsCount
SET n.friendCount = friendsCount
RETURN n.friendsCount
```

You can chain together as many query parts as the available memory permits.

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 are undefined.

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” \[300\]](#).

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

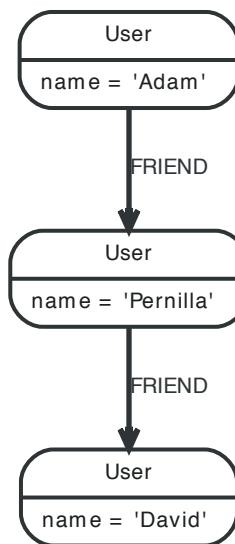
While pattern matching, Neo4j makes sure to not include matches where the same graph relationship is found multiple times in a single pattern. In most use cases, this is a sensible thing to do.

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” \[300\]](#). For details on parameters when using the Neo4j embedded Java API, see [Section 33.15, “Query Parameters” \[609\]](#).

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
```

Case-sensitive string pattern matching

Parameters

```
{  
  "name" : "Michael"  
}
```

Query

```
MATCH (n)  
WHERE n.name STARTS WITH { name }  
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, Neo4j 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" \[117\]](#)). 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 2.2` at the beginning, that particular query will be parsed with the 2.2 version of the parser. Below is an example using the `START` clause to access a legacy index:

```
CYPHER 2.2
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 1.9
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" \[157\]](#) for more information on the correct syntax for this. The `START` clause should only be used when accessing legacy indexes (see [Chapter 35, Legacy Indexing \[619\]](#)).

Supported Language Versions

Neo4j 2.3 supports the following versions of the Cypher language:

- Neo4j Cypher 2.3
- Neo4j Cypher 2.2
- 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.



Chapter 9. Syntax

The nitty-gritty details of Cypher syntax.

9.1. Values

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” \[120\]](#)). Special care must be taken when using NULL, as NULL is a value of every type (see [Section 9.8, “Working with NULL” \[134\]](#)). 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, 02127, -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 dynamic property: n["prop"], rel[n.city + n.zip], map[coll[0]].
- 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 regular expression: a.name =~ "Tob.*"
- A case-sensitive string matching expression: a.surname STARTS WITH "Sven", a.surname ENDS WITH "son" or a.surname CONTAINS "son"
- 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.



Identifiers are only visible in the same query part

Identifiers are not carried over to subsequent queries. If multiple query parts are chained together using WITH, identifiers have to be listed in the WITH clause to be carried over to the next part. For more information see [Section 10.5, "With" \[146\]](#).

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" \[124\]](#) on how they behave.

The operators STARTS WITH, ENDS WITH and CONTAINS can be used to search for a string value by its content.

Boolean operators

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

String operators

Strings can be concatenated using the + operator. For regular expression matching the =~ operator is used.

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" \[119\]](#)) 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).
- The special value `java.lang.Double.NaN` is regarded as being larger than all other numbers.
- 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.

Chaining Comparison Operations

Comparisons can be chained arbitrarily, e.g., `x < y <= z` is equivalent to `x < y AND y <= z`.

Formally, if `a`, `b`, `c`, ..., `y`, `z` are expressions and `op1`, `op2`, ..., `opN` are comparison operators, then `a op1 b op2 c ... y opN z` is equivalent to `a op1 b AND b op2 c AND ... y opN z`.

Note that `a op1 b op2 c` does not imply any kind of comparison between `a` and `c`, so that, e.g., `x < y > z` is perfectly legal (though perhaps not pretty).

The example:

```
MATCH (n) WHERE 21 < n.age <= 30 RETURN n
```

is equivalent to

```
MATCH (n) WHERE 21 < n.age AND n.age <= 30 RETURN n
```

Thus it will match all nodes where the age is between 21 and 30.

This syntax extends to all equality and inequality comparisons, as well as extending to chains longer than three.

For example:

```
a < b = c <= d <> e
```

Is equivalent to:

```
a < b AND b = c AND c <= d AND d <> e
```

For other comparison operators, see [the section called “Comparison operators” \[124\]](#).

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” \[157\]](#)
- [Section 11.2, “Optional Match” \[166\]](#)
- [Section 12.1, “Create” \[189\]](#)
- [Section 12.2, “Merge” \[194\]](#)
- the section called “Using path patterns in `WHERE`” [172]

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`.

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



Caution

Variable length pattern matching in versions 2.1.x and earlier does not enforce relationship uniqueness for patterns described inside of a single MATCH clause. This means that a query such as the following: MATCH (a)-[r]->(b), (a)-[rs*]->(c) RETURN * may include r as part of the rs set. This behavior has changed in versions 2.2.0 and later, in such a way that r will be excluded from the result set, as this better adheres to the rules of relationship uniqueness as documented here [Section 8.4, “Uniqueness” \[111\]](#). If you have a query pattern that needs to retrace relationships rather than ignoring them as the relationship uniqueness rules normally dictate, you can accomplish this using multiple match clauses, as follows: MATCH (a)-[r]->(b) MATCH (a)-[rs*]->(c) RETURN *. This will work in all versions of Neo4j that support the MATCH clause, namely 2.0.0 and later.

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 relationships, 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 identifier, 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 size of a collection like this:

Query

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

Result

size(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" }]} AS result
```

Result

result

```
{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
<code>2 IN [1, 2, 3]</code>	<code>TRUE</code>
<code>2 IN [1, NULL, 3]</code>	<code>NULL</code>
<code>2 IN [1, 2, NULL]</code>	<code>TRUE</code>
<code>2 IN [1]</code>	<code>FALSE</code>
<code>2 IN []</code>	<code>FALSE</code>
<code>NULL IN [1,2,3]</code>	<code>NULL</code>
<code>NULL IN [1,NULL,3]</code>	<code>NULL</code>
<code>NULL IN []</code>	<code>FALSE</code>

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: `[][], 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.

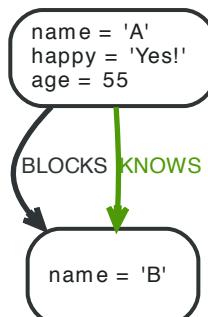
In the `RETURN` part of your query, you define which parts of the pattern you are interested in. It can be nodes, relationships, or properties on these.



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}, :BLOCKS[1]{}, Node[1]{name:"B"}]	:BLOCKS[1]{}
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]{}

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}, :BLOCKS[1]{}], Node[1]{name:"B"}], [Node[0]{name:"A", happy:"Yes!", age:55}, :KNOWS[0]{}], 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

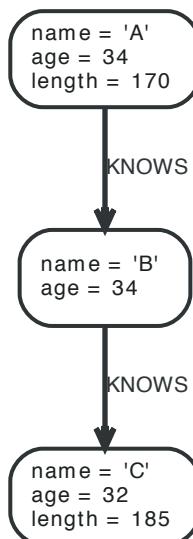
`ORDER BY` is a sub-clause following `RETURN` or `WITH`, and it specifies that the output should be sorted and how.

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" \[125\]](#).

In terms of scope of identifiers, `ORDER BY` follows special rules, depending on if the projecting `RETURN` or `WITH` clause is either aggregating or `DISTINCT`. If it is an aggregating or `DISTINCT` projection, only the identifiers available in the projection are available. If the projection does not alter the output cardinality (which aggregation and `DISTINCT` do), identifiers available from before the projecting clause are also available. When the projection clause shadows already existing identifiers, only the new identifiers are available.

Lastly, it is not allowed to use aggregating expressions in the `ORDER BY` sub-clause if they are not also listed in the projecting clause. This last rule is to make sure that `ORDER BY` does not change the results, only the order of them.

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}

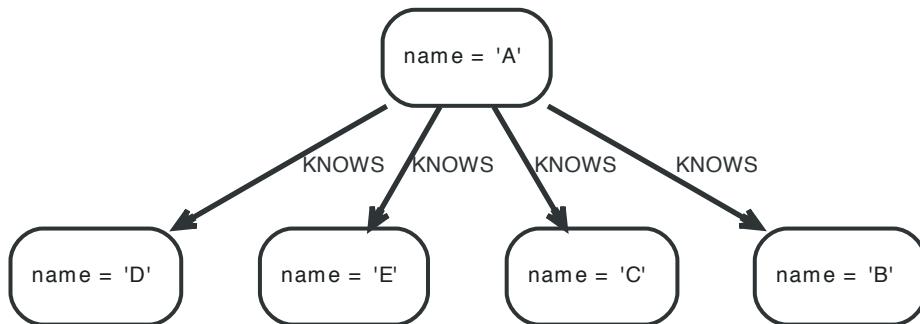
3 rows

10.3. Limit

`LIMIT` constrains the number of rows in the output.

`LIMIT` accepts any expression that evaluates to a positive integer — however the expression cannot refer to nodes or relationships.

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

Return first from expression

`Limit` accepts any expression that evaluates to a positive integer as long as it is not referring to any external identifiers:

Parameters

```

{
  "p" : 12
}
  
```

Query

```

MATCH (n)
RETURN n
ORDER BY n.name
LIMITtoInt(3 * rand()) + 1
  
```

Returns one to three top items

Result

n

Node[2]{name:"A"}

Node[3]{name:"B"}

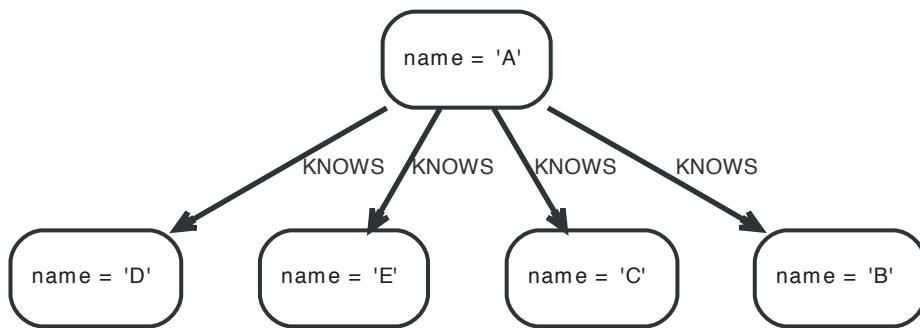
2 rows

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. `SKIP` accepts any expression that evaluates to a positive integer — however the expression cannot refer to nodes or relationships.

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

Skip first from expression

Skip accepts any expression that evaluates to a positive integer as long as it is not referring to any external identifiers:

Query

```
MATCH (n)
RETURN n
ORDER BY n.name
SKIP toInt(3*rand())+ 1
```

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

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.

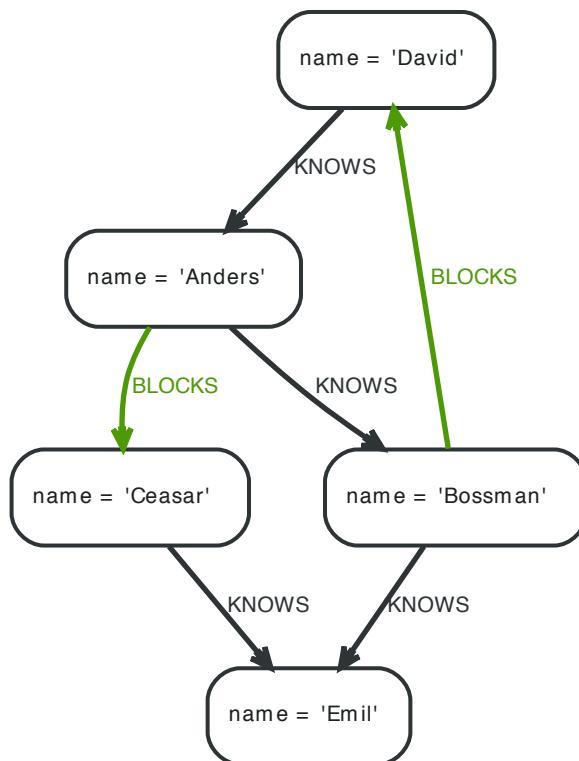
Using `WITH`, you can manipulate the output before it is passed on to the following query parts. The manipulations can be of the shape and/or number of entries in the result set.

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 be 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 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

```
"Bossman"
```

```
"Anders"
```

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 COLLECTED 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

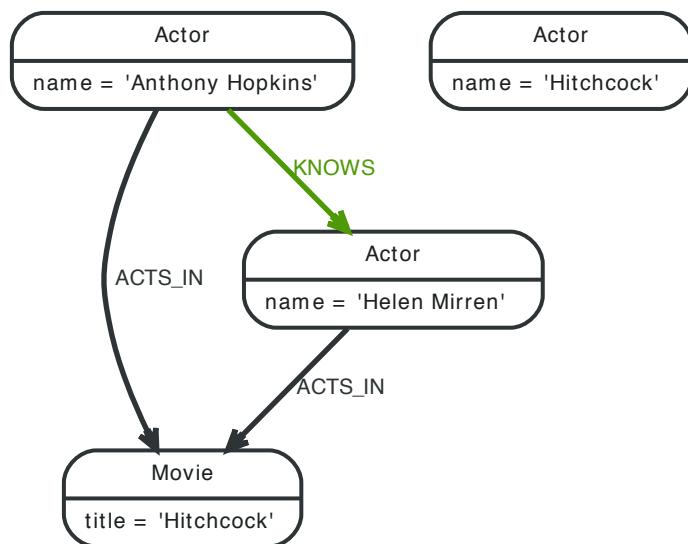
The `UNION` clause is used to combine the result of multiple queries.

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 influence the decisions of the planner when building an execution plan for a query.



Caution

Forcing planner behavior is an advanced feature, and should be used with caution by experienced developers and/or database administrators only, as it may cause queries to perform poorly.

When executing a query, Neo4j needs to decide where in the query graph to start matching. This is done by looking at the `MATCH` clause and the `WHERE` conditions and using that information to find useful indexes.

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.

You can also force Neo4j to produce plans which perform joins between query sub-graphs.

Query using an index hint

To query using an index hint, use `USING INDEX`.

Query

```
MATCH (n:Swede)
USING INDEX n:Swede(surname)
WHERE n.surname = 'Taylor'
RETURN n
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	1	1	0	n	n
+NodeIndexSeek	1	1	2	n	:Swede(surname)

Total database accesses: 2

Query using multiple index hints

To query using multiple index hints, use `USING INDEX`.

Query

```
MATCH (m:German)-->(n:Swede)
USING INDEX m:German(surname)
USING INDEX n:Swede(surname)
```

General Clauses

```
WHERE m.surname = 'Plantikow' AND n.surname = 'Taylor'  
RETURN m
```

Query Plan

Operator	Rows	DB Hits	Identifiers	Other
+ColumnFilter	1	0	m	keep columns m
+TraversalMatcher	1	11	anon[17], m, n n, anon[17], m	

Total database accesses: 11

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
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	1	1	0	m	m
+Filter	1	1	1	m	m.surname == { AUTOSTRING0}
+NodeByLabelScan	1	1	2	m	:German

Total database accesses: 3

Hinting a join on a single node

To force the query planner to produce plans with joins in them, use USING JOIN.

Query

```
MATCH (andres { name:'Andres' })-->(x)<--(emil { name: 'Emil' })  
USING JOIN ON x  
RETURN x
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	0	1	0	x	x
+Filter	0	1	6	anon[31], anon[37], andres, emil, x Ands(NOT(anon[31] == anon[37]), andres.name == { AUTOSTRING0})	

```

| | +Expand(All) | 2 | 9 | 12 | anon[31], anon[37], andres, emil, x | (x)<--(andres)
| |
| | +NodeHashJoin | 1 | 3 | 0 | anon[37], emil, x | x
| | \
| | +AllNodesScan | 5 | 5 | 6 | x |
| |
| | +Expand(All) | 1 | 3 | 4 | anon[37], emil, x | (emil)-->(x)
| |
| | +Filter | 1 | 1 | 5 | emil | emil.name == { AUTOSTRING1}
| |
| | +AllNodesScan | 5 | 5 | 6 | emil |
| |
Total database accesses: 39

```

Hinting a join on multiple nodes

To force the query planner to produce plans with joins in them, use USING JOIN.

Query

```

MATCH (andy { name:'Andres' })-[r1]->(x)<-[r2]-(y)-[r3]-(andy)
USING JOIN ON x, y
RETURN x, y

```

Query Plan

```

+-----+
| Operator      | Estimated Rows | Rows | DB Hits | Identifiers          | Other           |
+-----+
| +ProduceResults | 1 | 3 | 0 | x, y | x, y |
| |
| +Filter | 1 | 3 | 0 | andy, r1, r2, r3, x, y | Ands(NOT(r1 == r2), NOT(r2 == r3)) |
| |
| +NodeHashJoin | 1 | 3 | 0 | andy, r1, r2, r3, x, y | x, y |
| | \
| | +Expand(All) | 10 | 10 | 15 | r2, x, y | (y)-[r2:]->(x) |
| | |
| | +AllNodesScan | 5 | 5 | 6 | y | |
| |
| +Filter | 3 | 3 | 0 | andy, r1, r3, x, y | NOT(r1 == r3) |
| |
| +Expand(All) | 3 | 4 | 5 | andy, r1, r3, x, y | (andy)-[r3:]->(y) |
| |
| +Expand(All) | 1 | 1 | 2 | andy, r1, x | (andy)-[r1:]->(x) |
| |
| +Filter | 1 | 1 | 5 | andy | andy.name == { AUTOSTRING0} |
| |
| +AllNodesScan | 5 | 5 | 6 | andy | |
+-----+

```

General Clauses

Total database accesses: 39

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” \[127\]](#).

`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” \[246\]](#), and more about the specifying index hints to force Neo4j to use a specific index in [Section 10.8, “Using” \[152\]](#).

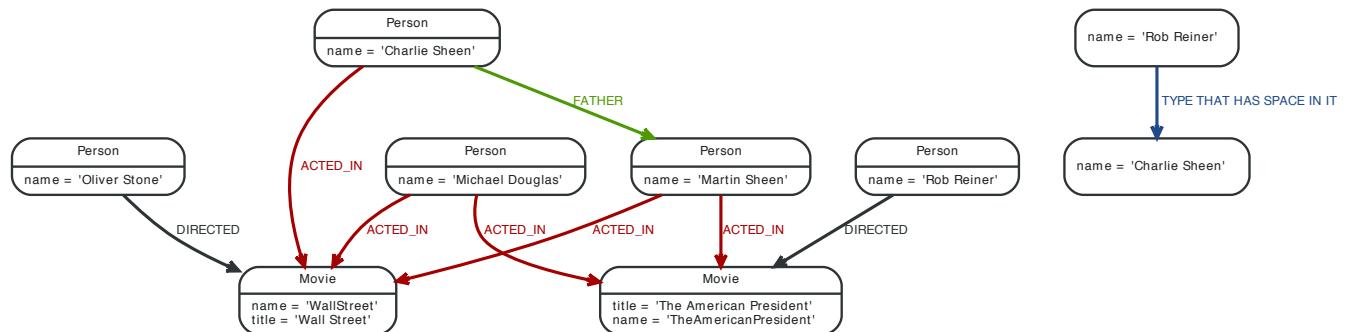


Tip

To understand more about the patterns used in the `MATCH` clause, read [Section 9.6, “Patterns” \[127\]](#).

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]{title:"The American President", name:"TheAmericanPresident"}  
Node[4]{name:"WallStreet", title:"Wall Street"}  
Node[5]{name:"Rob Reiner"}  
Node[6]{name:"Michael Douglas"}  
Node[7]{name:"Rob Reiner"}  
Node[8]{name:"Charlie Sheen"}  
9 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]{title:"The American President", name:"TheAmericanPresident"}  
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"}
```

3 rows

actor

```
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[0]{name:"Oliver Stone"}
```

```
Node[6]{name:"Michael Douglas"}
```

```
Node[2]{name:"Martin Sheen"}
```

```
Node[1]{name:"Charlie Sheen"}
```

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" \[111\]](#).

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]{title:"The American President", name:"TheAmericanPresident"}
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]{}]
```

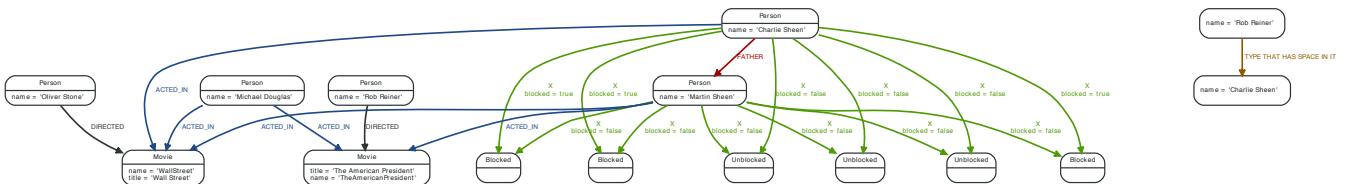
2 rows

Match with properties on a variable length path

A variable length relationship with properties defined on 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 Charlie 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[9]{blocked:false}, Node[9]{}, :X[10]{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"}
```

5 rows

x

```
Node[1]{name:"Charlie Sheen"}  
Node[2]{name:"Martin Sheen"}  
Node[6]{name:"Michael Douglas"}  
Node[0]{name:"Oliver Stone"}
```

5 rows

Named path

If you want to return or filter on a path in your pattern graph, you can 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]{title:"The American President",  
name:"TheAmericanPresident"}]  
[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]{title:"The American President", name:"TheAmericanPresident"}, :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” \[164\]](#) 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 `NULLS` 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 `NULLS` 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.

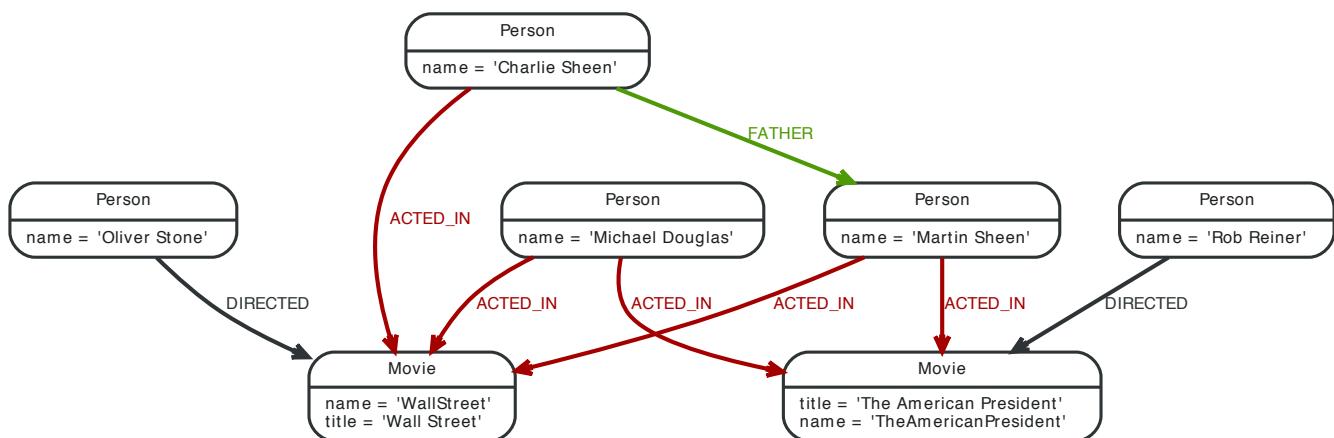


Tip

To understand the patterns used in the `OPTIONAL MATCH` clause, read [Section 9.6, "Patterns"](#) [127].

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 its 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 a MATCH or OPTIONAL MATCH clause or filters the results of a WITH clause.

WHERE is not a clause in its 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.

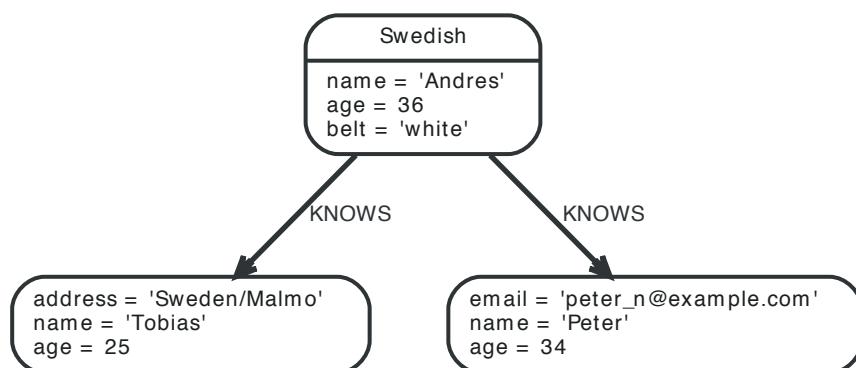
For MATCH and OPTIONAL MATCH on the other hand, WHERE adds constraints to the patterns described. *It should not be seen as a filter after the matching is finished.*



Important

In the case of multiple MATCH / OPTIONAL MATCH clauses, the predicate in WHERE is always a part of the patterns in the directly preceding MATCH / OPTIONAL 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" \[134\]](#) 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
  
```

Result

n

```

Node[0]{address:"Sweden/Malmo", name:"Tobias", age:25}
Node[1]{email:"peter_n@example.com", 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
```

"Tobias" is returned because he is younger than 30.

Result

n

```
Node[0]{address:"Sweden/Malmo", name:"Tobias", age:25}
```

1 row

Filter on dynamic node property

To filter on a property using a dynamically computed name, use square bracket syntax.

Parameters

```
{
  "prop" : "AGE"
}
```

Query

```
MATCH (n)
WHERE n[tolower({ prop })]< 30
RETURN n
```

"Tobias" is returned because he is younger than 30.

Result

n

```
Node[0]{address:"Sweden/Malmo", name:"Tobias", age:25}
```

1 row

Property exists

Use the `EXISTS()` function to only include nodes or relationships in which a property exists.

Query

```
MATCH (n)
WHERE exists(n.belt)
RETURN n
```

"Andres" will be returned because he is the only one with a `belt` property.



Important

The `HAS()` function has been superseded by `EXISTS()` and will be removed in a future release.

Result

n

```
Node[2]{name:"Andres", age:36, belt:"white"}
```

1 row

String matching

The start and end of strings can be matched using STARTS WITH and ENDS WITH. To match regardless of location in a string, use CONTAINS. The matching is *case-sensitive*.

Match the start of a string

The STARTS WITH operator is used to perform case-sensitive matching on the start of strings.

Query

```
MATCH (n)
WHERE n.name STARTS WITH 'Pet'
RETURN n
```

"Peter" will be returned because his name starts with Pet.

Result

n

```
Node[1]{email:"peter_n@example.com", name:"Peter", age:34}
```

1 row

Match the end of a string

The ENDS WITH operator is used to perform case-sensitive matching on the end of strings.

Query

```
MATCH (n)
WHERE n.name ENDS WITH 'ter'
RETURN n
```

"Peter" will be returned because his name ends with ter.

Result

n

```
Node[1]{email:"peter_n@example.com", name:"Peter", age:34}
```

1 row

Match anywhere in a string

The CONTAINS operator is used to perform case-sensitive matching regardless of location in strings.

Query

```
MATCH (n)
WHERE n.name CONTAINS 'ete'
RETURN n
```

"Peter" will be returned because his name contains ete.

Result

n

```
Node[1]{email:"peter_n@example.com", name:"Peter", age:34}
```

1 row

String matching negation

Use the NOT keyword to exclude all matches on given string from your result:

Query

```
MATCH (n)
WHERE NOT n.name ENDS WITH 's'
RETURN n
```

"Peter" will be returned because his name does not end with s.

Result

n

```
Node[1]{email:"peter_n@example.com", name:"Peter", age:34}
1 row
```

Regular expressions

Cypher supports filtering using regular expressions. The regular expression syntax is inherited from [the Java regular expressions](#)¹. This includes support for flags that change how strings are matched, including case-insensitive (?i), multiline (?m) and dotall (?s). Flags are given at the start of the regular expression, for example MATCH (n) WHERE n.name =~ '(?i)Lon.*' RETURN n will return nodes with name London or with name LonDoN.

Regular expressions

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

Query

```
MATCH (n)
WHERE n.name =~ 'Tob.*'
RETURN n
```

"Tobias" is returned because his name starts with Tob.

Result

n

```
Node[0]{address:"Sweden/Malmo", 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.address =~ 'Sweden\\\/Malmo'
RETURN n
```

"Tobias" is returned because his address is in Sweden/Malmo.

Result

n

```
Node[0]{address:"Sweden/Malmo", name:"Tobias", age:25}
1 row
```

Case insensitive regular expressions

By pre-pending a regular expression with (?i), the whole expression becomes case insensitive.

Query

¹ <https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>

```
MATCH (n)
WHERE n.name =~ '(?i)ANDR.*'
RETURN n
```

"Andres" is returned because his name starts with ANDR regardless of case.

Result

n

```
Node[2]{name:"Andres", age:36, belt:"white"}  
1 row
```

Using path 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]{address:"Sweden/Malmo", name:"Tobias", age:25}  
Node[1]{email:"peter_n@example.com", 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]{}
```

```
: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]{address:"Sweden/Malmo", name:"Tobias", age:25}
```

```
Node[1]{email:"peter_n@example.com", name:"Peter", age:34}
```

2 rows

Missing properties and values

Default to false if property is missing

As missing properties evaluate to NULL, the comparison 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 white belts 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"}  
Node[1]{email:"peter_n@example.com", name:"Peter", age:34}  
Node[0]{address:"Sweden/Malmo", 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]{email:"peter_n@example.com", name:"Peter", age:34}  
1 row
```

Using ranges

Simple range

To check for an element being inside a specific range, use the inequality operators <, <=, >=, >.

Query

```
MATCH (a)
WHERE a.name >= 'Peter'
RETURN a
```

Nodes having a name property lexicographically greater than or equal to *Peter* are returned.

Result

a

```
Node[0]{address:"Sweden/Malmo", name:"Tobias", age:25}  
Node[1]{email:"peter_n@example.com", name:"Peter", age:34}
```

2 rows

Composite range

Several inequalities can be used to construct a range.

Query

```
MATCH (a)  
WHERE a.name > 'Andres' AND a.name < 'Tobias'  
RETURN a
```

Nodes having a name property lexicographically between *Andres* and *Tobias* are returned.

Result

a

```
Node[1]{email:"peter_n@example.com", name:"Peter", age:34}
```

1 row

11.4. Start

Find starting points through legacy indexes.



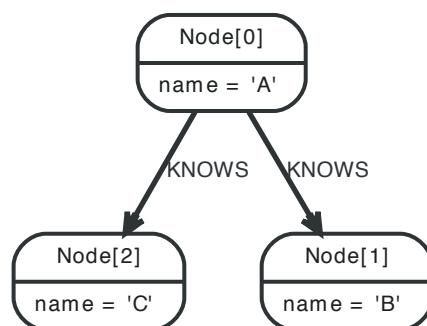
Important

The START clause should only be used when accessing legacy indexes (see [Chapter 35, Legacy Indexing \[619\]](#)). In all other cases, use MATCH instead (see [Section 11.1, "Match" \[157\]](#)).

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 seeks. 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 seek

When the starting point can be found by using index seeks, 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 seek

When the starting point can be found by using index seeks, 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:

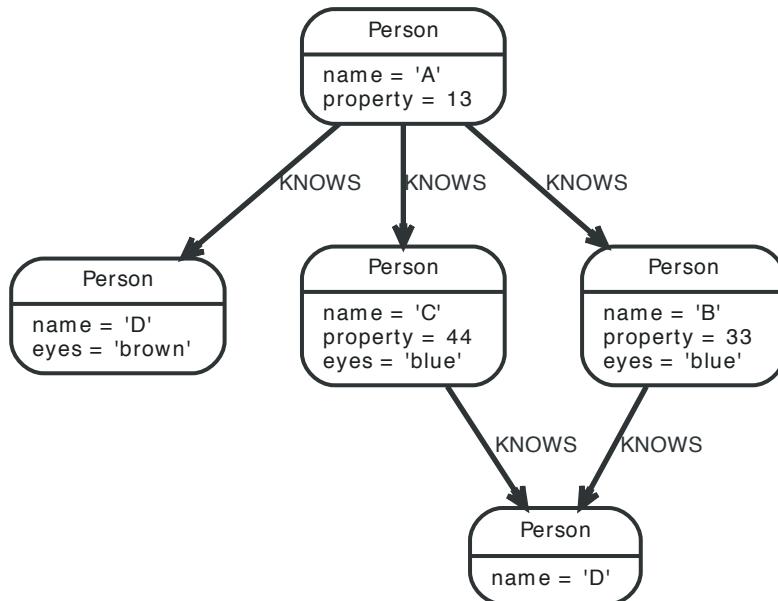
```
RETURN n, count(*)
```

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)
```

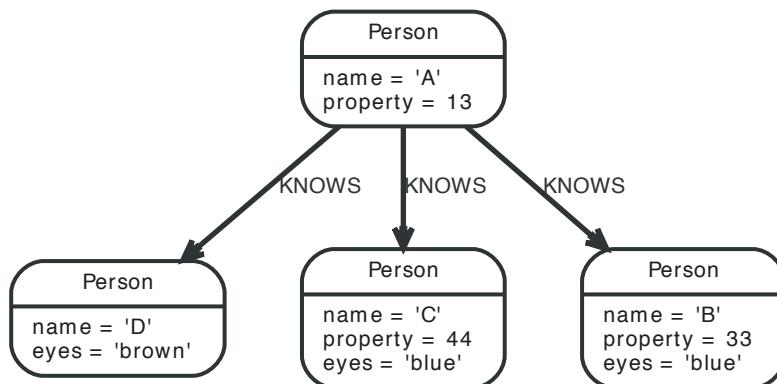
In this example we are trying to find all our friends of friends, and count them. The first aggregate function, `count(DISTINCT friend_of_friend)`, will only see a `friend_of_friend` once — `DISTINCT` removes the duplicates. The latter aggregate function, `count(friend_of_friend)`, might very well see 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. NULLs 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 NULLS.

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.

- The URL of the CSV file is specified by using `FROM` followed by an arbitrary expression evaluating to the URL in question.
- It is required to specify an identifier for the CSV data using `AS`.
- `LOAD CSV` supports resources compressed with *gzip*, *Deflate*, as well as *ZIP* archives.
- CSV files can be stored on the database server and are then accessible using a `file:///` URL. Alternatively, `LOAD CSV` also supports accessing CSV files via *HTTPS*, *HTTP*, and *FTP*.
- `LOAD CSV` will follow *HTTP* redirects but for security reasons it will not follow redirects that changes the protocol, for example if the redirect is going from *HTTPS* to *HTTP*.



Configuration settings for file URLs

`allow_file_urls` [467]

This setting determines if Cypher will allow the use of `file:///` URLs when loading data using `LOAD CSV`. Such URLs identify files on the filesystem of the database server. Default is *true*.

`dbms.security.load_csv_file_url_root` [470] Sets the root directory for `file:///` URLs used with the Cypher `LOAD CSV` clause. This must be set to a single directory on the filesystem of the database server, and will make all requests to load from `file:///` URLs relative to the specified directory (similar to how a unix chroot operates). By default, this setting is not configured.

- When not set, file URLs will be resolved as relative to the root of the database server filesystem. If this is the case, a file URL will typically look like `file:///home/username/myfile.csv` or `file:///C:/Users/username/myfile.csv`. Using these URLs in `LOAD CSV` will read content from files on the database server filesystem, specifically `/home/username/myfile.csv` and `C:\Users\username\myfile.csv` respectively. *For security reasons you may not want users to be able to load files located anywhere on the database server filesystem and should set dbms.security.load_csv_file_url_root to a safe directory to load files from.*
- When set, file URLs will be resolved as relative to the directory it's set to. In this case a file URL will typically look like `file:///myfile.csv` or `file:///myproject/myfile.csv`.
 - If set to `data/import` using the above URLs in `LOAD CSV` would read content from `data/import/myfile.csv` and `data/import/myproject/myfile.csv` respectively, where both are relative to the database install directory.
 - If set to `/home/neo4j` using the above URLs in `LOAD CSV` would read content from `/home/neo4j/myfile.csv` and `/home/neo4j/myproject/myfile.csv` respectively.

See the examples below for further details.

There is also a worked example, see [Section 12.8, “Importing CSV files with Cypher” \[213\]](#).

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/2.3.1/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/2.3.1/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/2.3.1/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” \[215\]](#).

Query

```
USING PERIODIC COMMIT  
LOAD CSV FROM 'http://neo4j.com/docs/2.3.1/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/2.3.1/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/2.3.1/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” \[127\]](#) 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 multiple nodes

Creating multiple nodes is done by separating them with a comma.

Query

```
CREATE (n),(m)
```

Result

(empty result)

Nodes created: 2

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.

Parameters

```
{
  "props" : [ {
    "name" : "Andres",
    "position" : "Developer"
  }, {
    "name" : "Michael",
    "position" : "Developer"
  } ]
}
```

Query

```
UNWIND { props } AS map
CREATE (n)
SET n = map
```

Result

(empty result)

Nodes created: 2
Properties set: 4

Create multiple nodes with a parameter for their properties using old syntax

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.



Note

This syntax is deprecated in Neo4j version 2.3. It may be removed in a future major release.
See the above example using UNWIND for how to achieve the same functionality.

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.

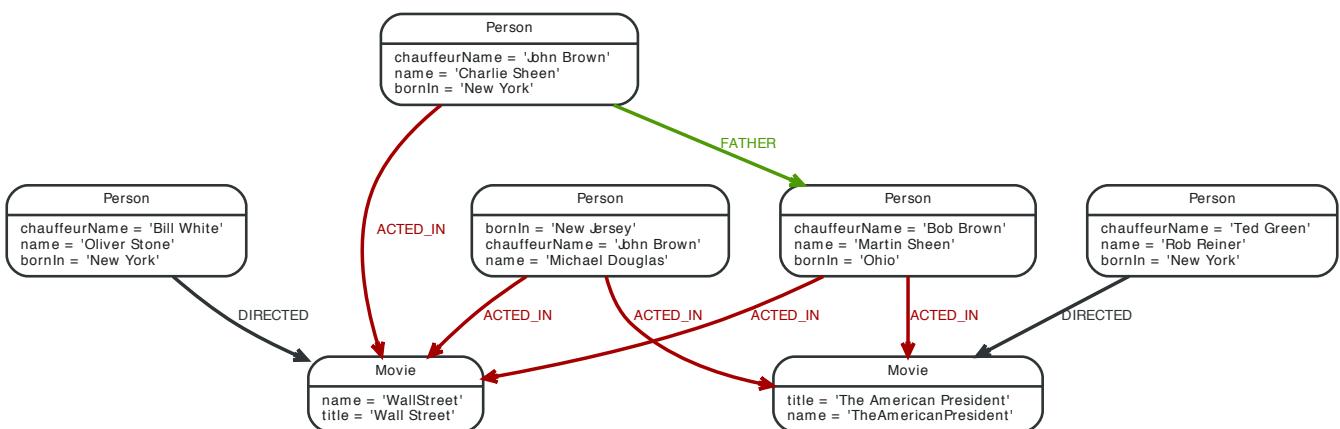
As with `MATCH`, `MERGE` can match multiple occurrences of a pattern. If there are multiple matches, they will all be passed on to later stages of the query.

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 `MATCHed` in the database or if it was `CREATED`.

The rule planner (see [Section 15.1, "How are queries executed?" \[256\]](#)) expands a `MERGE` pattern from the end point that has the identifier with the lowest lexicographical order. This means that it might choose a suboptimal expansion path, expanding from a node with a higher degree. The pattern `MERGE (a:A)-[:R]->(b:B)` will always expand from a to b, so if it is known that b nodes are a better choice for start point, renaming identifiers could improve performance.

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)
  
```

A new node is created because there are no nodes labeled `Critic` in the database.

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.name, michael.bornIn
```

Michael Douglas will be matched and the *name* and *bornIn* properties returned.

Result

michael.name	michael.bornIn
"Michael Douglas"	"New Jersey"

1 row

Merge single node derived from an existing node property

For some property *p* in each bound node in a set of nodes, a single new node is created for each unique value for *p*.

Query

```
MATCH (person:Person)
MERGE (city:City { name: person.bornIn })
RETURN person.name, person.bornIn, city
```

Three nodes labeled *city* are created, each of which contains a *name* property with the value of *New York*, *Ohio*, and *New Jersey*, respectively. Note that even though the MATCH clause results in three bound nodes having the value *New York* for the *bornIn* property, only a single *New York* node (i.e. a city node with a name of *New York*) is created. As the *New York* node is not matched for the first bound node, it is created. However, the newly-created *New York* node is matched and bound for the second and third bound nodes.

Result

person.name	person.bornIn	city
"Oliver Stone"	"New York"	Node[7]{name:"New York"}
"Charlie Sheen"	"New York"	Node[7]{name:"New York"}
"Martin Sheen"	"Ohio"	Node[8]{name:"Ohio"}
"Rob Reiner"	"New York"	Node[7]{name:"New York"}
"Michael Douglas"	"New Jersey"	Node[9]{name:"New Jersey"}

5 rows

Nodes created: 3

Properties set: 3

Labels added: 3

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.name, keanu.created
```

The query creates the *keanu* node and sets a timestamp on creation time.

Result

keanu.name	keanu.created
"Keanu Reeves"	1447164221581

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.name, person.found
```

The query finds all the Person nodes, sets a property on them, and returns them.

Result

person.name	person.found
"Oliver Stone"	true
"Charlie Sheen"	true
"Martin Sheen"	true
"Rob Reiner"	true
"Michael Douglas"	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.name, keanu.created, keanu.lastSeen
```

The query creates the *keanu* node, and sets a timestamp on creation time. If *keanu* had already existed, a different property would have been set.

Result

keanu.name	keanu.created	keanu.lastSeen
"Keanu Reeves"	1447164225447	<null>

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.name, person.found, person.lastAccessed
```

Result

person.name	person.found	person.lastAccessed
"Oliver Stone"	true	1447164224164
"Charlie Sheen"	true	1447164224164
"Martin Sheen"	true	1447164224164
"Rob Reiner"	true	1447164224164
"Michael Douglas"	true	1447164224164

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 charlie.name, type(r), wallStreet.title
```

Charlie Sheen had already been marked as acting in *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

charlie.name	type(r)	wallStreet.title
"Charlie Sheen"	"ACTED_IN"	"Wall Street"

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, Neo4j 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

```

As *Charlie Sheen* and *Oliver Stone* do not know each other, 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

Merge on a relationship between two existing nodes

MERGE can be used in conjunction with preceding MATCH and MERGE clauses to create a relationship between two bound nodes *m* and *n*, where *m* is returned by MATCH and *n* is created or matched by the earlier MERGE.

Query

```

MATCH (person:Person)
MERGE (city:City { name: person.bornIn })
MERGE (person)-[r:BORN_IN]->(city)
RETURN person.name, person.bornIn, city

```

This builds on the example from [the section called “Merge single node derived from an existing node property” \[195\]](#). The second MERGE creates a BORN_IN relationship between each person and a city corresponding to the value of the person’s *bornIn* property. *Charlie Sheen*, *Rob Reiner* and *Oliver Stone* all have a BORN_IN relationship to the same city node (*New York*).

Result

person.name	person.bornIn	city
"Oliver Stone"	"New York"	Node[7]{name:"New York"}

5 rows
 Nodes created: 3
 Relationships created: 5
 Properties set: 3
 Labels added: 3

person.name	person.bornIn	city
"Charlie Sheen"	"New York"	Node[7]{name:"New York"}
"Martin Sheen"	"Ohio"	Node[8]{name:"Ohio"}
"Rob Reiner"	"New York"	Node[7]{name:"New York"}
"Michael Douglas"	"New Jersey"	Node[9]{name:"New Jersey"}

5 rows
 Nodes created: 3
 Relationships created: 5
 Properties set: 3
 Labels added: 3

Merge on a relationship between an existing node and a merged node derived from a node property

MERGE can be used to simultaneously create both a new node n and a relationship between a bound node m and n .

Query

```

MATCH (person:Person)
MERGE (person)-[r:HAS_CHAUFFEUR]->(chauffeur:Chauffeur { name: person.chauffeurName })
RETURN person.name, person.chauffeurName, chauffeur
  
```

As MERGE found no matches — in our example graph, there are no nodes labeled with Chauffeur and no HAS_CHAUFFEUR relationships — MERGE creates five nodes labeled with Chauffeur, each of which contains a *name* property whose value corresponds to each matched Person node's *chauffeurName* property value. MERGE also creates a HAS_CHAUFFEUR relationship between each Person node and the newly-created corresponding Chauffeur node. As *Charlie Sheen* and *Michael Douglas* both have a chauffeur with the same name — *John Brown* — a new node is created in each case, resulting in two Chauffeur nodes having a *name* of *John Brown*, correctly denoting the fact that even though the *name* property may be identical, these are two separate people. This is in contrast to the example shown above in the section called “Merge on a relationship between two existing nodes” [198], where we used the first MERGE to bind the City nodes to prevent them from being recreated (and thus duplicated) in the second MERGE.

Result

person.name	person.chauffeurName	chauffeur
"Oliver Stone"	"Bill White"	Node[7]{name:"Bill White"}
"Charlie Sheen"	"John Brown"	Node[8]{name:"John Brown"}
"Martin Sheen"	"Bob Brown"	Node[9]{name:"Bob Brown"}
"Rob Reiner"	"Ted Green"	Node[10]{name:"Ted Green"}
"Michael Douglas"	"John Brown"	Node[11]{name:"John Brown"}

5 rows
 Nodes created: 5
 Relationships created: 5
 Properties set: 5
 Labels added: 5

Using unique constraints with MERGE

Cypher prevents getting conflicting results from MERGE when using patterns that involve uniqueness constraints. 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.name
```

The query creates the *laurence* node. If *laurence* had already existed, MERGE would just match the existing node.

Result

laurence.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.name, oliver.bornIn
```

The *oliver* node already exists, so MERGE just matches it.

Result

oliver.name	oliver.bornIn
"Oliver Stone"	"New York"

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” \[113\]](#).

Parameters

```
{
  "param" : {
    "name" : "Keanu Reeves",
    "role" : "Neo"
  }
}
```

Query

```
MERGE (person:Person { name: { param }.name, role: { param }.role })
RETURN person.name, person.role
```

Result

person.name	person.role
"Keanu Reeves"	"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.

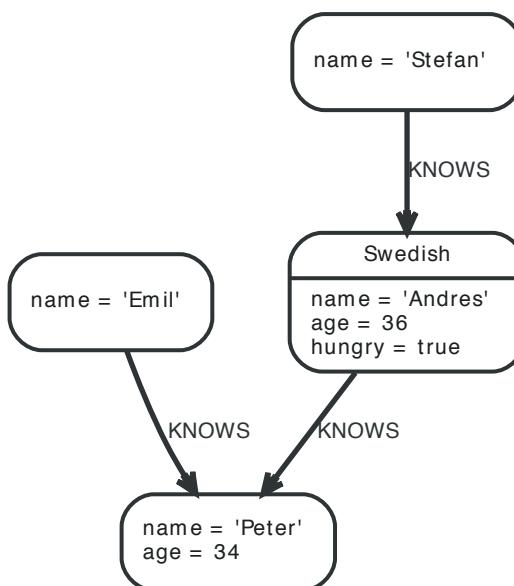
`SET` can also be used with maps from parameters to set properties.



Note

Setting labels on a node is an idempotent operation — 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]{surname:"Taylor", name:"Andres", age:36, hungry:true}
  
```

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]{hungry:true, age:36}

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 its 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, parameter, 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]{surname:"Taylor", name:"Andres", age:36, hungry:true}

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

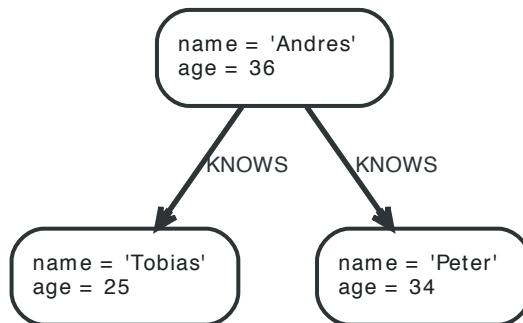
Labels added: 2

12.4. Delete

The `DELETE` clause is used to delete graph elements — nodes, relationships or paths.

For removing properties and labels, see [Section 12.5, “Remove” \[207\]](#). Remember that you can not delete a node without also deleting relationships that start or end on said node. Either explicitly delete the relationships, or use `DETACH DELETE`.

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
```

Result

(empty result)

Nodes deleted: 1

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)
DETACH DELETE n
```

Result

(empty result)

Nodes deleted: 3

Relationships deleted: 2

Delete a node with all its relationships

When you want to delete a node and any relationship going to or from it, use `DETACH DELETE`.

Query

```
MATCH (n { name:'Andres' })
DETACH DELETE n
```

Result

(empty result)

Nodes deleted: 1

Relationships deleted: 2

12.5. Remove

The `REMOVE` clause is used to remove properties and labels from graph elements.

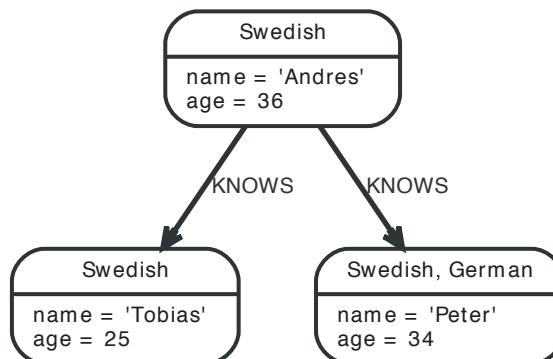
For deleting nodes and relationships, see [Section 12.4, “Delete” \[206\]](#).



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.

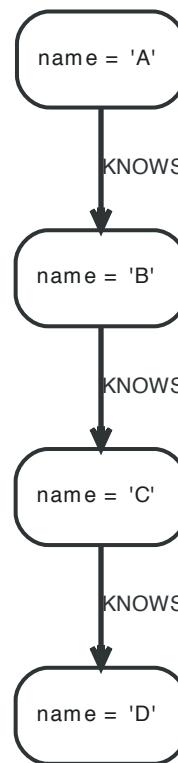
Collections and paths are key concepts in Cypher. To use them for updating data, you can use the `FOREACH` construct. It allows you to do updating commands on elements in a collection — a path, or a collection created by aggregation.

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” \[148\]](#)) 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.

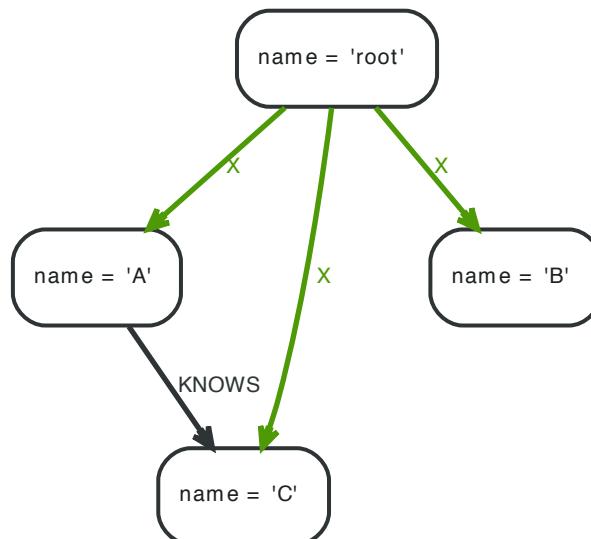
Another difference to `MATCH` is that `CREATE UNIQUE` assumes the pattern to be unique. If multiple matching subgraphs are found an error will be generated.



Tip

In the `CREATE UNIQUE` clause, patterns are used a lot. Read [Section 9.6, “Patterns” \[127\]](#) 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 against 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. `LOAD CSV` will follow HTTP redirects but for security reasons it will not follow redirects that changes the protocol, for example if the redirect is going from HTTPS to HTTP.

For more details, see [Section 11.6, “Load CSV” \[184\]](#).

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/2.3.1/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” \[246\]](#)) or a unique constraint (see [Section 14.2, “Constraints” \[249\]](#)) 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/2.3.1/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” \[215\]](#)) 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/2.3.1/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” \[213\]](#) on how to import data from CSV files.

Importing large amounts of data using `LOAD CSV` 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 using `LOAD CSV`. 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” \[186\]](#) in [Section 11.6, “Load CSV” \[184\]](#) 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 importing large amounts of data. However, it will also break transactional isolation and 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” [124].



Note

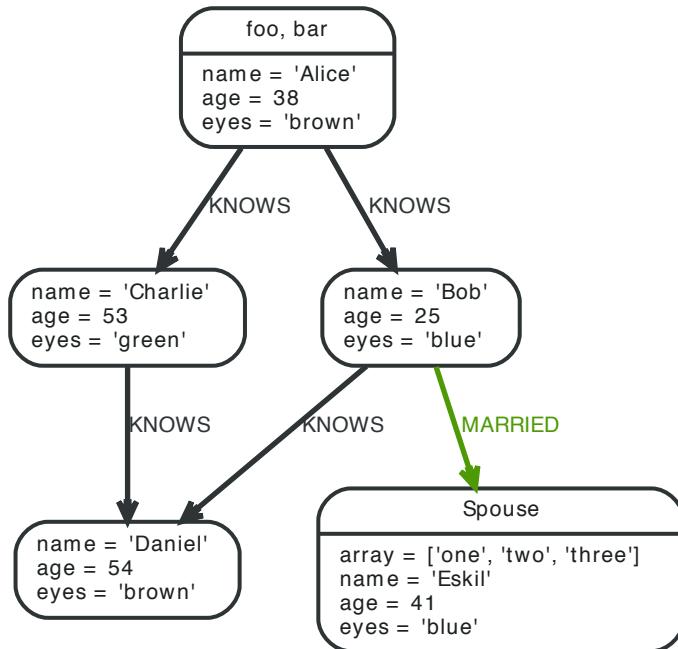
Most functions in Cypher will return NULL if an input parameter is NULL.

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” [124].

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]{array:["one", "two", "three"], name:"Eskil", age:41, eyes:"blue"}
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

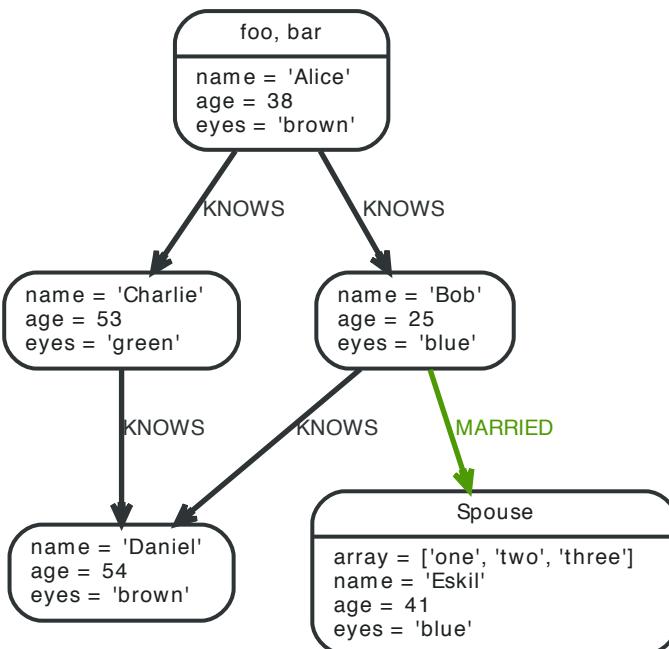
13.2. Scalar functions

Scalar functions return a single value.

Important

The LENGTH and SIZE functions are quite similar, and so it is important to take note of the difference. Due to backwards compatibility LENGTH currently works on four types: strings, paths, collections and pattern expressions. However, for clarity it is recommended to only use LENGTH on strings and paths, and use the new SIZE function on collections and pattern expressions. LENGTH on those types may be deprecated in future.

Figure 13.2. Graph



SIZE

To return or filter on the size of a collection, use the SIZE() function.

Syntax: SIZE(collection)

Arguments:

- *collection*: An expression that returns a collection

Query

```
RETURN size(['Alice', 'Bob']) AS col
```

The number of items in the collection is returned by the query.

Result

col

2

1 row

SIZE of pattern expression

This is the same SIZE() 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 size of the result is calculated, not the length of the expression itself.

Syntax: SIZE(pattern expression)

Arguments:

- *pattern expression*: A pattern expression that returns a collection

Query

```
MATCH (a)
WHERE a.name='Alice'
RETURN size((a)-->()-->()) AS fofof
```

The number of sub-graphs matching the pattern expression is returned by the query.

Result

fof

3

1 row

LENGTH

To return or filter on the length of a path, use the LENGTH() function.

Syntax: LENGTH(path)

Arguments:

- *path*: An expression that returns a path

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 string

To return or filter on the length of a string, use the LENGTH() function.

Syntax: LENGTH(string)

Arguments:

- *string*: An expression that returns a string

Query

```
MATCH (a)
WHERE length(a.name)> 6
RETURN length(a.name)
```

The length of the name Charlie is returned by the query.

Result

length(a.name)

7

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

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()

```
1447164269722
```

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

```
RETURNtoFloat("11.5"), toFloat("not a number")
```

Result

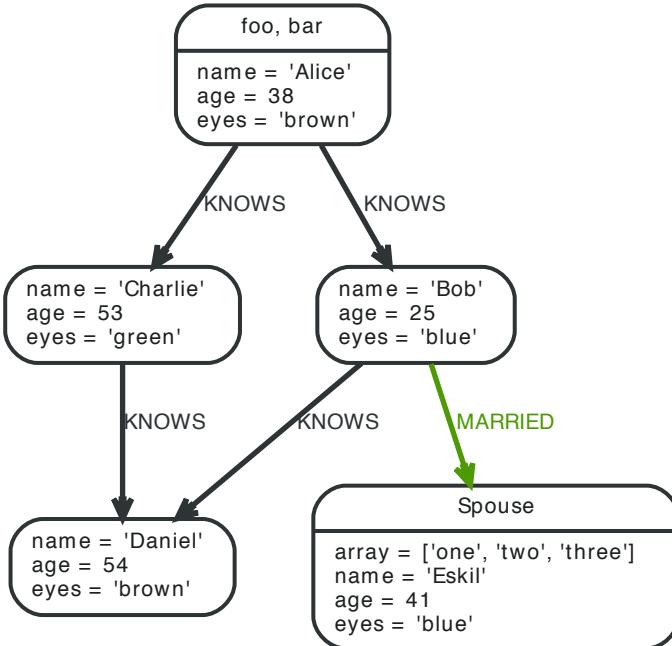
toFloat("11.5")	toFloat("not a number")
11.5	<null>
1 row	

13.3. Collection functions

Collection functions return collections of things — nodes in a path, and so on.

See also the section called “Collection operators” [124].

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]{array:
["one", "two", "three"], name:"Eskil", age:41, eyes:"blue"}]
  
```

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 its 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 size(x)= 3)
```

This returns the property named `array` and a list of values in it, which have size 3.

Result

a.array

["one", "two", "three"]

filter(x in a.array WHERE size(x) = 3)

["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 its 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

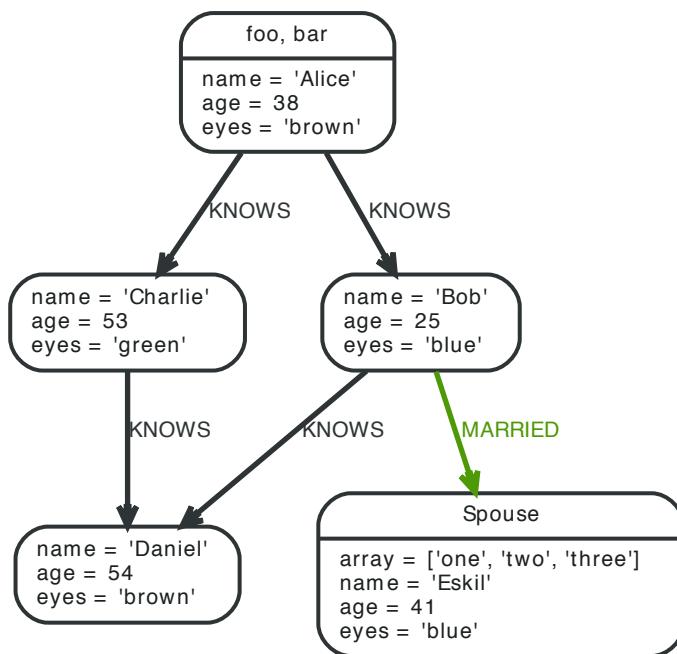
1 row

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” [124].

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

CEIL

CEIL returns the smallest integer greater than or equal to the number.

Syntax: CEIL(*expression*)

Arguments:

- *expression*: A numeric expression.

Query

```
RETURN ceil(0.1)
```

The ceil of 0.1

Result

ceil(0.1)

1.0

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. 6366287987295618
```

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)

-1.0

1 row

sign(0.1)

1.0

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

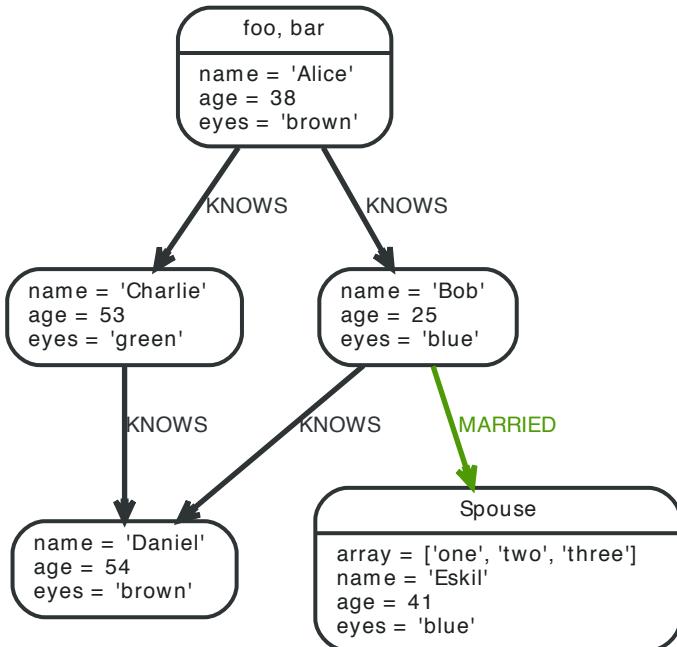
1 row

13.5. String functions

These functions all operate on string expressions only, and will return an error if used on any other values. The exception to this rule is `TOSTRING()`, which also accepts numbers.

See also the section called "String operators" [124].

Figure 13.5. Graph



STR

`STR` returns a string representation of the expression. If the expression returns a string the result will be wrapped in quotation marks.

Syntax: `STR(expression)`

Arguments:

- *expression*: An expression that returns anything

Query

```
RETURN str(1), str("hello")
```

Result

<code>str(1)</code>	<code>str("hello")</code>
"1"	""hello""

1 row



Note

The `STR()` function is deprecated from Neo4j version 2.3 and onwards. This means it may be removed in a future Neo4j major release.

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)

"ell"

substring("hello", 2)

"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 which are delimited by split patterns.

Syntax: SPLIT(*original*, *splitPattern*)

Arguments:

- *original*: An expression that returns a string
- *splitPattern*: The string to split the original string with

Query

```
RETURN split("one,two", ",")
```

Result**split("one,two", ",")**

["one", "two"]

1 row

REVERSE

REVERSE returns the original string reversed.

Syntax: REVERSE(*original*)**Arguments:**

- *original*: An expression that returns a string

Query

```
RETURN reverse("anagram")
```

Result**reverse("anagram")**

"margana"

1 row

TOSTRING

TOSTRING converts the argument to a string. It converts integral 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 a number or a string

Query

```
RETURN toString(11.5), toString("already a string")
```

Result**toString(11.5)**

"11.5"

toString("already a string")

"already a string"

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, so deciding what to index and what not to index is an important and often non-trivial task.

Cypher allows the creation of indexes over a property for all nodes that have a given label. Once an index has been created, it will automatically be managed and kept up to date by the database whenever the graph is changed. Neo4j will automatically pick up and start using the index once it has been created and brought online.

Create an index

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.

Query

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

Result

(empty result)

Drop an index

To drop an index on all nodes that have a label and property combination, 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 want Cypher to use specific indexes, you can enforce it using hints. See [Section 10.8, “Using” \[152\]](#).

Query

```
MATCH (person:Person { name: 'Andres' })
RETURN person
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	1	1	0	person	person
+NodeIndexSeek	1	1	2	person	:Person(name)

Total database accesses: 2

Use index with WHERE using equality

Indexes are also automatically used for equality comparisons of an indexed property in the `WHERE` clause. If you want Cypher to use specific indexes, you can enforce it using hints. See [Section 10.8, “Using” \[152\]](#).

Query

```
MATCH (person:Person)
WHERE person.name = 'Andres'
RETURN person
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	1	1	0	person	person
+NodeIndexSeek	1	1	2	person	:Person(name)

Total database accesses: 2

Use index with WHERE using inequality

Indexes are also automatically used for inequality (range) comparisons of an indexed property in the WHERE clause. If you want Cypher to use specific indexes, you can enforce it using hints. See [Section 10.8, “Using” \[152\]](#).

Query

```
MATCH (person:Person)
WHERE person.name > 'B'
RETURN person
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	33	1	0	person	person
+NodeIndexSeekByRange	33	1	2	person	:Person(name) > { AUTOSTRING0 }

Total database accesses: 2

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 want Cypher to use specific indexes, you can enforce it using hints. See [Section 10.8, “Using” \[152\]](#).

Query

```
MATCH (person:Person)
WHERE person.name IN ['Andres', 'Mark']
RETURN person
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	2	2	0	person	person
+NodeIndexSeek	2	2	4	person	:Person(name)

Total database accesses: 4

Use index with STARTS WITH

The STARTS WITH predicate on person.name in the following query will use the Person(name) index, if it exists.



Note

The similar operators ENDS WITH and CONTAINS cannot currently be solved using indexes.

Query

```
MATCH (person:Person)
WHERE person.name STARTS WITH 'And'
RETURN person
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	26	1	0	person	person
+NodeIndexSeekByRange	26	1	2	person	:Person(name STARTS WITH { AUTOSTRING0})

Total database accesses: 2

Use index when checking for the existence of a property

The has(p.name) predicate in the following query will use the Person(name) index, if it exists.

Query

```
MATCH (p:Person)
WHERE HAS (p.name)
RETURN p
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	2	2	0	p	p
+NodeIndexScan	2	2	3	p	:Person(name)

Total database accesses: 3

14.2. Constraints

Neo4j helps enforce data integrity with the use of constraints. Constraints can be applied to either nodes or relationships. Unique node property constraints can be created, as well as node and relationship property existence constraints.

You can use unique property 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.

You can use property existence constraints to ensure that a property exists for all nodes with a specific label or for all relationships with a specific type. All queries that try to create new nodes or relationships without the property, or queries that try to remove the mandatory property will now fail.



Note

Property existence constraints are only available in the Neo4j Enterprise Edition. Note that databases with property existence constraints cannot be opened using Neo4j Community Edition.

You can have multiple constraints for a given label and you can also combine unique and property existence constraints on the same property.

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”.

Note that adding a unique property 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 unique property constraint and still want an index on the property, you will have to create the index.

The existing constraints can be listed using the REST API, see [Section 21.16, “Constraints” \[371\]](#).

Unique node property constraints

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)

Unique 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)

Unique constraints removed: 1

Create a node that complies with unique property 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 unique property 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 unique property constraint due to conflicting nodes

Create a unique property 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" \[246\]](#) 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)
```

Node property existence constraints**Create node property existence constraint**

To create a constraint that makes sure that all nodes with a certain label have a certain property, use the ASSERT exists(identifier.propertyName) syntax.

Query

```
CREATE CONSTRAINT ON (book:Book) ASSERT exists(book.isbn)
```

Result

(empty result)

Property existence constraints added: 1

Drop node property existence constraint

By using DROP CONSTRAINT, you remove a constraint from the database.

Query

```
DROP CONSTRAINT ON (book:Book) ASSERT exists(book.isbn)
```

Result

(empty result)

Property existence constraints removed: 1

Create a node that complies with property existence constraints

Create a Book node with an existing isbn property.

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 property existence constraint

Trying to create a Book node without an isbn property, given a property existence constraint on :Book(isbn).

Query

```
CREATE (book:Book { title: 'Graph Databases' })
```

In this case the node isn't created in the graph.

Error message

```
Node 1 with label "Book" must have the property "isbn" due to a constraint
```

Removing an existence constrained node property

Trying to remove the isbn property from an existing node book, given a property existence constraint on :Book(isbn).

Query

```
MATCH (book:Book { title: 'Graph Databases' })  
REMOVE book.isbn
```

In this case the property is not removed.

Error message

```
Node 0 with label "Book" must have the property "isbn" due to a constraint
```

Failure to create a node property existence constraint due to existing node

Create a constraint on the property isbn on nodes with the Book label when there already exists a node without an isbn.

Query

```
CREATE CONSTRAINT ON (book:Book) ASSERT exists(book.isbn)
```

In this case the constraint can't be created because it is violated by existing data. We may choose to remove the offending nodes and then re-apply the constraint.

Error message

```
Unable to create CONSTRAINT ON ( book:Book ) ASSERT exists(book.isbn):  
Node(0) with label 'Book' has no value for property 'isbn'
```

Relationship property existence constraints

Create relationship property existence constraint

To create a constraint that makes sure that all relationships with a certain type have a certain property, use the `ASSERT exists(identifier.propertyName)` syntax.

Query

```
CREATE CONSTRAINT ON ()-[like:LIKED]-() ASSERT exists(like.day)
```

Result

(empty result)

Property existence constraints added: 1

Drop relationship property existence constraint

To remove a constraint from the database, use `DROP CONSTRAINT`.

Query

```
DROP CONSTRAINT ON ()-[like:LIKED]-() ASSERT exists(like.day)
```

Result

(empty result)

Property existence constraints removed: 1

Create a relationship that complies with property existence constraints

Create a LIKED relationship with an existing day property.

Query

```
CREATE (user:User)-[like:LIKED { day: 'yesterday' }]->(book:Book)
```

Result

(empty result)

Nodes created: 2

Relationships created: 1

Properties set: 1

Labels added: 2

Create a relationship that breaks a property existence constraint

Trying to create a LIKED relationship without a day property, given a property existence constraint `:LIKED(day)`.

Query

```
CREATE (user:User)-[like:LIKED]->(book:Book)
```

In this case the relationship isn't created in the graph.

Error message

```
Relationship 1 with type "LIKED" must have the property "day" due to a constraint
```

Removing an existence constrained relationship property

Trying to remove the day property from an existing relationship like of type LIKED, given a property existence constraint `:LIKED(day)`.

Query

```
MATCH (user:User)-[like:LIKED]->(book:Book)
```

```
REMOVE like.day
```

In this case the property is not removed.

Error message

```
Relationship 0 with type "LIKED" must have the property "day" due to a constraint
```

Failure to create a relationship property existence constraint due to existing relationship

Create a constraint on the property day on relationships with the LIKED type when there already exists a relationship without a property named day.

Query

```
CREATE CONSTRAINT ON ()-[like:LIKED]-() ASSERT exists(like.day)
```

In this case the constraint can't be created because it is violated by existing data. We may choose to remove the offending relationships and then re-apply the constraint.

Error message

```
Unable to create CONSTRAINT ON ()-[ liked:LIKED ]-() ASSERT exists(liked.day):  
Relationship(0) with type 'LIKED' has no value for property 'day'
```

14.3. Statistics

When you issue a Cypher query, it gets compiled to an execution plan (see [Chapter 16, Execution Plans \[261\]](#)) 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

Execution plans are cached and will not be replanned until the statistical information used to produce the plan has changed. The following configuration options allows you to control how sensitive replanning should be to updates of the database.

`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.

`dbms.cypher.statistics_divergence` Controls how much the above statistical information is allowed to change before an execution plan is considered stale and has to be replanned. If the relative change in any of statistics is larger than this threshold, the plan will be thrown away and a new one will be created. A threshold of 0.0 means *always replan*, and a value of 1.0 means *never replan*.

Managing statistics from the shell

Usage:

<code>schema sample -a</code>	will sample all indexes.
<code>schema sample -l Person -p name</code>	will sample the index for label Person on property name (if existing).
<code>schema sample -a -f</code>	will force a sample of all indexes.
<code>schema sample -f -l :Person -p name</code>	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 \[261\]](#).

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 \[469\]](#)), 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 <i>profiling your query uses more resources</i> , so you should not profile unless you are actively working on a query. |

See [Chapter 16, Execution Plans \[261\]](#) 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/2.3.1/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/2.3.1/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/2.3.1/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?” \[257\]](#) but in this case we're going to prefix our query with PROFILE:

```
PROFILE
MATCH (p { name:"Tom Hanks" })
RETURN p
```

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	16	1	0	p	p
+Filter	16	1	163	p	p.name == { AUTOSTRING0}
+AllNodesScan	163	163	164	p	

Total database accesses: 327

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
```

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	13	1	0	p	p
+Filter	13	1	125	p	p.name == { AUTOSTRING0 }
+NodeByLabelScan	125	125	126	p	:Person

Total database accesses: 251

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
```

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	1	1	0	p	p
+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 seek (see [Section 14.1, “Indexes” \[246\]](#)) to find the appropriate node.

Chapter 16. Execution Plans

Neo4j 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 <i>database hit</i> is an abstract unit of this storage engine work.

See [Section 15.2, “How do I profile a query?” \[257\]](#) for how to view the execution plan for your query.

For a deeper understanding of how each operator works, see the relevant section. Operators are grouped into high-level categories. Please remember that the statistics of the actual database where the queries run on will decide the plan used. There is no guarantee that a specific query will always be solved with the same plan.

16.1. Starting point operators

These operators find parts of the graph from which to start.

All Nodes Scan

Reads all nodes from the node store. The identifier that will contain the nodes is seen in the arguments. If your query is using this operator, you are very likely to see performance problems on any non-trivial database.

Query

```
MATCH (n)
RETURN n
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	35	35	0	n	
+AllNodesScan	35	35	36	n	

Total database accesses: 36

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.

Query

```
MATCH (n1)-[r]->()
WHERE id(r)= 0
RETURN r, n1
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	1	1	0	n1, r	r, n1
+DirectedRelationshipByIdSeekPipe	1	1	1	anon[17], n1, r	EntityByIdRhs(SingleSeekArg({AUTOINT0}))

Total database accesses: 1

Node by Id seek

Reads one or more nodes by id from the node store.

Query

```
MATCH (n)
WHERE id(n)= 0
```

```
RETURN n
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	1	1	0	n	n
+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 from the node label index.

Query

```
MATCH (person:Person)
RETURN person
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	14	14	0	person	person
+NodeByLabelScan	14	14	15	person	:Person

Total database accesses: 15

Node index seek

Finds nodes using an index seek. The node identifier and the index used is shown in the arguments of the operator. If the index is a unique index, the operator is called NodeUniqueIndexSeek instead.

Query

```
MATCH (location:Location { name: "Malmo" })
RETURN location
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	1	1	0	location	location
+NodeIndexSeek	1	1	2	location	:Location(name)

Total database accesses: 2

Node index range seek

Finds nodes using an index seek where the value of the property matches a given prefix string. This operator can be used for STARTS WITH and comparators such as <, >, <= and >=

Query

```
MATCH (l:Location)
```

```
WHERE l.name STARTS WITH 'Lon'
RETURN 1
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	26	1	0	1	1
+NodeIndexSeekByRange	26	1	2	1	:Location(name STARTS WITH { AUTOSTRING0})

Total database accesses: 2

Node index scan

An index scan goes through all values stored in an index, and can be used to find all nodes with a particular label having a specified property (e.g. exists(n.prop)).

Query

```
MATCH (l:Location)
WHERE HAS (l.name)
RETURN 1
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	10	10	0	1	1
+NodeIndexScan	10	10	11	1	:Location(name)

Total database accesses: 11

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.

Query

```
MATCH (n1)-[r]-()
WHERE id(r)= 1
RETURN r, n1
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	1	2	0	n1, r	r, n1
+UndirectedRelationshipByIdSeek	1	2	1	anon[16], n1, r	

Total database accesses: 1

16.2. Expand operators

These operators explore the graph by expanding graph patterns.

Expand All

Given a start node, expand-all will follow relationships coming in or out, depending on the pattern relationship. Can also handle variable length pattern relationships.

Query

```
MATCH (p:Person { name: "me" })-[:FRIENDS_WITH]-(fof)
RETURN fof
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	0	1	0	fof	fof
+Expand(All)	0	1	2	anon[30], fof, p (p)-[:FRIENDS_WITH]->(fof)	
+NodeIndexSeek	1	1	2	p	:Person(name)

Total database accesses: 4

Expand Into

When both the start and end node have already been found, expand-into is used to find all connecting relationships between the two nodes.

Query

```
MATCH (p:Person { name: "me" })-[:FRIENDS_WITH]-(fof)-->(p)
RETURN fof
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	0	0	0	fof	fof
+Filter	0	0	0	anon[30], anon[53], fof, p NOT(anon[30] == anon[53])	
+Expand(Into)	0	0	0	anon[30], anon[53], fof, p (p)-[:FRIENDS_WITH]->(fof)	
+Expand(All)	0	0	1	anon[53], fof, p	(p)<--(fof)
+NodeIndexSeek	1	1	2	p	:Person(name)

Total database accesses: 3

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.

Query

```
MATCH (p:Person)
OPTIONAL MATCH (p)-[works_in:WORKS_IN]->(l)
WHERE works_in.duration > 180
RETURN p, l
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	14	15	0	l, p	p, l
+OptionalExpand(All)	14	15	44	l, p, works_in (p)-[works_in:WORKS_IN]->(l)	
+NodeByLabelScan	14	14	15	p	:Person

Total database accesses: 59

16.3. Combining operators

Node Hash Join

Using a hash table, a node hash join joins the inputs coming from the left with the inputs coming from the right. The join key is specified in the arguments of the operator.

Query

```
MATCH (andy:Person { name:'Andreas' })-[:WORKS_IN]->(loc)<-[:WORKS_IN]-(matt:Person { name:'Mattis' })
})
RETURN loc
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	35	0	0	loc	loc
+Filter	35	0	0	anon[37], anon[56], andy, loc, matt	NOT(anon[37] == anon[56])
+NodeHashJoin	35	0	0	anon[37], anon[56], andy, loc, matt	loc
+Expand(All)	35	0	0	anon[56], loc, matt	(matt)-[:WORKS_IN]->(loc)
+NodeIndexSeek	1	0	1	matt	:Person(name)
+Expand(All)	35	0	1	anon[37], andy, loc	(andy)-[:WORKS_IN]->(loc)
+NodeIndexSeek	1	1	2	andy	:Person(name)

Total database accesses: 4

Apply

Apply works by performing a nested loop. Every row being produced on the left hand side of the Apply operator will be fed to the Argument operator on the right hand side, and then Apply will yield the results coming from the RHS. Apply, being a nested loop, can be seen as a warning that a better plan was not found.

Query

```
MATCH (p:Person)-[:FRIENDS_WITH]->(f)
WITH p, count(f) AS fs
WHERE fs > 0
OPTIONAL MATCH (p)-[:WORKS_IN*1..2]->(city)
RETURN p, city
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	1	2	0	city, p	p, city
+Apply	1	2	0	anon[92], anon[126], city, fs, p	
+Apply	1	2	0	anon[92], anon[126], city, fs, p	
+Optional	1	2	0	anon[126], city, p	
+VarLengthExpand(All)	1	2	6	anon[126], city, p	(p)-[:WORKS_IN*]->(city)

Execution Plans

		+-----+ +-----+ +-----+					
+Argument		1 2 0 p					
		+-----+ +-----+ +-----+					
+Filter		1 2 0 anon[92], fs, p			anon[92]		
		+-----+ +-----+ +-----+					
+Argument		1 2 0 anon[92], fs, p					
		+-----+ +-----+ +-----+					
+Projection		1 2 0 anon[92], fs, p			p; fs; fs > { AUTOINT0 }		
		+-----+ +-----+ +-----+					
+EagerAggregation		1 2 0 fs, p			p		
		+-----+ +-----+ +-----+					
+Expand(All)		2 2 16 anon[17], f, p			(p)-[:FRIENDS_WITH]->(f)		
		+-----+ +-----+ +-----+					
+NodeByLabelScan		14 14 15 p			:Person		
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+							

Total database accesses: 37

Anti Semi Apply

Tests for the absence of a pattern predicate. A pattern predicate that is prepended by NOT is solved with AntiSemiApply.

Query

```
MATCH (me:Person { name: "me" }), (other:Person)
WHERE NOT (me)-[:FRIENDS_WITH]->(other)
RETURN other
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other		
+ProduceResults	4	13	0	other	other		
	+-----+ +-----+ +-----+						
+AntiSemiApply	4	13	0	me, other			
\	+-----+ +-----+ +-----+						
+Expand(Into)	0	0	48	anon[73], me, other (me)-[:FRIENDS_WITH]->(other)			
	+-----+ +-----+ +-----+						
+Argument	14	14	0	me, other			
	+-----+ +-----+ +-----+						
+CartesianProduct	14	14	0	me, other			
\	+-----+ +-----+ +-----+						
+NodeByLabelScan	14	14	15	other	:Person		
	+-----+ +-----+ +-----+						
+NodeIndexSeek	1	1	2	me	:Person(name)		
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+							

Total database accesses: 65

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 any friend 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

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	11	14	0	other	other
+SelectOrSemiApply	11	14	0	anon[42], other	anon[42]
\+Expand(All)	15	0	2	anon[82], anon[96], other	(other)-[:WORKS_IN]->()
+Argument	14	2	0	other	
+LetAntiSemiApply	14	14	0	anon[42], other	
\+Expand(All)	2	0	14	anon[50], anon[68], other	(other)-[:FRIENDS_WITH]->()
+Argument	14	14	0	other	
+NodeByLabelScan	14	14	15	other	:Person

Total database accesses: 31

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

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	13	14	0	other	other
+SelectOrSemiApply	13	14	0	anon[38], other	anon[38]
\+Expand(All)	15	0	12	anon[77], anon[91], other	(other)-[:WORKS_IN]->()
+Argument	14	12	0	other	
+LetSemiApply	14	14	0	anon[38], other	
\+Expand(All)	2	0	14	anon[46], anon[64], other	(other)-[:FRIENDS_WITH]->()
+Argument	14	14	0	other	
+NodeByLabelScan	14	14	15	other	:Person

Total database accesses: 41

Select Or Anti Semi Apply

Tests for the absence of a pattern predicate and evaluates a predicate.

Query

```
MATCH (other:Person)
WHERE other.age > 25 OR NOT (other)-[:FRIENDS_WITH]->()
RETURN other
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	4	12	0	other	other
+SelectOrAntiSemiApply	4	12	28	other	other.age > { AUTOINT0}
+Expand(All)	2	0	14	anon[68], anon[86], other	(other)-[:FRIENDS_WITH]->()
+Argument	14	14	0	other	
+NodeByLabelScan	14	14	15	other	:Person

Total database accesses: 57

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.

Query

```
MATCH (other:Person)
WHERE other.age > 25 OR (other)-[:FRIENDS_WITH]->()
RETURN other
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	11	2	0	other	other
+SelectOrSemiApply	11	2	28	other	other.age > { AUTOINT0}
+Expand(All)	2	0	14	anon[64], anon[82], other	(other)-[:FRIENDS_WITH]->()
+Argument	14	14	0	other	
+NodeByLabelScan	14	14	15	other	:Person

Total database accesses: 57

Semi Apply

Tests for the existence of a pattern predicate. `SemiApply` takes a row from its 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.

Query

```
MATCH (other:Person)
```

Execution Plans

```
WHERE (other)-[:FRIENDS_WITH]->()
RETURN other
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	11	2	0	other	other
+SemiApply	11	2	0	other	
+Expand(All)	2	0	14	anon[46], anon[64], other	(other)-[:FRIENDS_WITH]->()
+Argument	14	14	0	other	
+NodeByLabelScan	14	14	15	other	:Person

Total database accesses: 29

Triadic

Triadic is used to solve triangular queries, such as the very common "find my friend-of-friends that are not already my friend". It does so by putting all the "friends" in a set, and use that set to check if the friend-of-friends are already connected to me.

Query

```
MATCH (me:Person)-[:FRIENDS_WITH]-()-(:FRIENDS_WITH)-(other)
WHERE NOT (me)-[:FRIENDS_WITH]-(other)
RETURN other
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	0	2	0	other	other
+TriadicSelection	0	2	0	anon[18], anon[35], anon[37], me, other	me, anon[35], other
+Filter	0	2	0	anon[18], anon[35], anon[37], me, other	NOT(anon[18] == anon[37])
+Expand(All)	0	6	10	anon[18], anon[35], anon[37], me, other	()-[:FRIENDS_WITH]-(other)
+Argument	4	4	0	anon[18], anon[35], me	
+Expand(All)	4	4	18	anon[18], anon[35], me	(me)-[:FRIENDS_WITH]-()
+NodeByLabelScan	14	14	15	me	:Person

Total database accesses: 43

16.4. Row operators

These operators take rows produced by another operator and transform them to a different set of rows

Eager

For isolation purposes this operator makes sure that operations that affect subsequent operations are executed fully for the whole dataset before continuing execution. Otherwise it could trigger endless loops, matching data again, that was just created. The Eager operator can cause high memory usage when importing data or migrating graph structures. In such cases split up your operations into simpler steps e.g. you can import nodes and relationships separately. Alternatively return the records to be updated and run an update statement afterwards.

Query

```
MATCH (p:Person)
MERGE (:Person:Clone { name:p.name })
```

Query Plan

Operator	Rows	DB Hits	Identifiers	Other
+EmptyResult	0	0		
+UpdateGraph	14	195	anon[23], p	MergeNode; p.name; :Person(name)
+Eager	14	0	p	
+NodeByLabel	14	15	p	:Person

Total database accesses: 210

Distinct

Removes duplicate rows from the incoming stream of rows.

Query

```
MATCH (l:Location)-[:WORKS_IN]-(p:Person)
RETURN DISTINCT l
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	14	6	0	1	1
+Distinct	14	6	0	1	1
+Filter	15	15	15	anon[19], l, p	:Person
+Expand(All)	15	15	25	anon[19], l, p	(1)-[:WORKS_IN]-(p)
+NodeByLabelScan	10	10	11	1	:Location

Total database accesses: 51

Eager Aggregation

Eagerly loads underlying results and stores it in a hash-map, using the grouping keys as the keys for the map.

Query

```
MATCH (:Location)<-[<:WORKS_IN]-(:Person)
RETURN l.name AS location, COLLECT(p.name) AS people
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	4	6	0	location, people	location, people
+EagerAggregation	4	6	15	location, people	location
+Projection	15	15	15	anon[19], l, location, p l.name; p	
+Filter	15	15	15	anon[19], l, p	p:Person
+Expand(All)	15	15	25	anon[19], l, p	(l)<-[<:WORKS_IN]-(:Person)
+NodeByLabelScan	10	10	11	1	:Location

Total database accesses: 81

Filter

Filters each row coming from the child operator, only passing through rows that evaluate the predicates to TRUE.

Query

```
MATCH (p:Person)
WHERE p.name =~ "^.a.*"
RETURN p
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	14	0	0	p	p
+Filter	14	0	14	p	p.name =~ /{ AUTOSTRING0}/
+NodeByLabelScan	14	14	15	p	:Person

Total database accesses: 29

Limit

Returns the first n rows from the incoming input.

Query

```
MATCH (p:Person)
RETURN p
LIMIT 3
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	3	3	0	p	p

Execution Plans

	+-----+ +-----+ +-----+ +-----+			
+Limit	3 3 0 p	Literal(3)		
+NodeByLabelScan	14 3 4 p	:Person		
+-----+ +-----+ +-----+ +-----+				

Total database accesses: 4

Projection

For each row from its input, projection evaluates a set of expressions and produces a row with the results of the expressions.

Query

```
RETURN "hello" AS greeting
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	1 1 0 greeting	greeting			
+Projection	1 1 0 greeting	{ AUTOSTRING0}			

Total database accesses: 0

Skip

Skips n rows from the incoming rows

Query

```
MATCH (p:Person)
RETURN p
ORDER BY p.id
SKIP 1
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	14 13 0 p	p			
+Projection	14 13 0 anon[35], anon[59], p, p anon[35]				
+Skip	14 13 0 anon[35], anon[59], p { AUTOINT0}				
+Sort	14 14 0 anon[35], anon[59], p anon[59]				
+Projection	14 14 28 anon[35], anon[59], p anon[35]; anon[35].id				
+Projection	14 14 0 anon[35], p	p			
+NodeByLabelScan	14 14 15 p	:Person			

Total database accesses: 43

Sort

Sorts rows by a provided key.

Query

```
MATCH (p:Person)
RETURN p
ORDER BY p.name
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	14	14	0	p	p
+Projection	14	14	0	anon[24], anon[37], p, p anon[24]	
+Sort	14	14	0	anon[24], anon[37], p	anon[37]
+Projection	14	14	14	anon[24], anon[37], p	anon[24]; anon[24].name
+Projection	14	14	0	anon[24], p	p
+NodeByLabelScan	14	14	15	p	:Person

Total database accesses: 29

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.

Query

```
MATCH (p:Person)
RETURN p
ORDER BY p.name
LIMIT 2
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	2	2	0	p	p
+Projection	2	2	0	anon[24], anon[37], p, p anon[24]	
+Top	2	2	0	anon[24], anon[37], p	Literal(2);
+Projection	14	14	14	anon[24], anon[37], p	anon[24]; anon[24].name
+Projection	14	14	0	anon[24], p	p
+NodeByLabelScan	14	14	15	p	:Person

Total database accesses: 29

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

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	10	11	0	p.name	p.name
+Union	10	11	0	p.name	
+Projection	1	1	1	p, p.name	p.name
+NodeByLabelScan	1	1	2	p	:Country
+Projection	10	10	10	p, p.name	p.name
+NodeByLabelScan	10	10	11	p	:Location

Total database accesses: 24

Unwind

Takes a collection of values and returns one row per item in the collection.

Query

```
UNWIND range(1,5) AS value
RETURN value;
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Identifiers	Other
+ProduceResults	10	5	0	value	value
+UNWIND	10	5	0	value	
+Argument	1	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.

Query

```
CREATE CONSTRAINT ON (c:Country) ASSERT c.name IS UNIQUE
```

Query Plan

Operator	Rows	DB Hits
+ConstraintOperation	0	3

Total database accesses: 3

Empty Result

Eagerly loads everything coming in to the `EmptyResult` operator and discards it.

Query

```
CREATE (:Person)
```

Query Plan

Operator	Rows	DB Hits	Identifiers	Other
+EmptyResult	0	0		
+UpdateGraph	1	2	anon[7]	CreateNode

Total database accesses: 2

Update Graph

Applies updates to the graph.

Query

```
CREATE (:Person { name: "Alistair" })
```

Query Plan

Operator	Rows	DB Hits	Identifiers	Other
+EmptyResult	0	0		
+UpdateGraph	1	4	anon[7]	CreateNode

Total database accesses: 4

Merge Into

When both the start and end node have already been found, merge-into is used to find all connecting relationships or creating a new relationship between the two nodes.

Query

```
MATCH (p:Person { name: "me" }),(f:Person { name: "Andres" })
MERGE (p)-[:FRIENDS_WITH]->(f)
```

Query Plan

Operator	Rows	DB Hits	Identifiers	Other
+EmptyResult	0	0		
+Merge(Into)	1	5	anon[68], f, p	(p)-[:FRIENDS_WITH]->(f)
+SchemaIndex	1	2	f, p	{ AUTOSTRING1}; :Person(name)
+SchemaIndex	1	2	p	{ AUTOSTRING0}; :Person(name)

Total database accesses: 9

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.

17. Capabilities	281
17.1. Data Security	282
17.2. Data Integrity	283
17.3. Data Integration	284
17.4. Availability and Reliability	285
17.5. Capacity	286
18. Transaction Management	287
18.1. Interaction cycle	288
18.2. Isolation levels	289
18.3. Default locking behavior	290
18.4. Deadlocks	291
18.5. Delete semantics	294
18.6. Creating unique nodes	295
18.7. Transaction events	296
19. Data Import	297
20. Graph Algorithms	298
21. REST API	299
21.1. Transactional Cypher HTTP endpoint	300
21.2. Neo4j Status Codes	309
21.3. REST API Authentication and Authorization	313
21.4. Service root	317
21.5. Streaming	318
21.6. Legacy Cypher HTTP endpoint	319
21.7. Property values	333
21.8. Nodes	334
21.9. Relationships	339
21.10. Relationship types	350
21.11. Node properties	351
21.12. Relationship properties	355
21.13. Node labels	361
21.14. Node degree	367
21.15. Indexing	369
21.16. Constraints	371
21.17. Traversals	375
21.18. Graph Algorithms	402
21.19. Batch operations	409
21.20. Legacy indexing	417
21.21. Unique Indexing	423
21.22. WADL Support	433
21.23. Using the REST API from WebLogic	434
22. Deprecations	435

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, file sizes 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 has a built-in page cache, that will cache the contents of the storage files. If there is not enough RAM to keep the storage files resident, then Neo4j will page parts of the files in and out as necessary, while keeping the most popular parts of the files resident at all times. 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^{16} (~ 65 000)

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 VII, “Advanced Usage” \[560\]](#)) 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.

Lost Updates in Cypher

In Cypher it is possible to acquire write locks to simulate improved isolation in some cases. Consider the case where multiple concurrent Cypher queries increment the value of a property. Due to the limitations of the read-committed isolation level, the increments will not result in a deterministic final value.

For example, the following query, if run by one hundred concurrent clients, will very likely not increment the property `n.prop` to 100, but some value lower than 100.

```
MATCH (n:X {id: 42})
SET n.prop = n.prop + 1
```

This is because all queries will read the value of `n.prop` within their own transaction. They will not see the incremented value from any other transaction that has not yet committed. In the worst case scenario the final value could be as low as 1, if all threads perform the read before any has committed their transaction.

To ensure deterministic behavior, it is necessary to grab a write lock on the node in question. In Cypher there is no explicit support for this, but we can work around this limitation by writing to a temporary property.

```
MATCH (n:X {id: 42})
SET n._LOCK_ = true
SET n.prop = n.prop + 1
REMOVE n._LOCK_
```

The existence of the `SET n._LOCK_` statement before the read of the `n.prop` read ensures the lock is acquired before the read action, and no updates will be lost due to enforced serialization of all concurrent queries on that specific node.

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:

```
TransactionTemplate template = new TransactionTemplate( ).retries( 5 ).backoff( 3, TimeUnit.SECONDS );
```

¹ <https://github.com/neo4j/neo4j/blob/2.3.1/community/kernel/src/test/java/examples/DeadlockDocTest.java>

² <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/helpers/TransactionTemplate.html>

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” \[603\]](#) 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 guarantee uniqueness across threads and transactions.

See [the section called “Get or create unique node using a legacy index” \[603\]](#) for how to do this using the core Java API. When using the REST API, see [Section 21.21, “Unique Indexing” \[423\]](#).

Pessimistic locking



Important

While this is a working solution, please consider using the preferred [the section called “Get or create” \[295\]](#) 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” \[604\]](#) for how to do this using the core Java API.

³ <http://neo4j.com/docs/2.3.1/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 in all registered handlers 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`. In `afterCommit` the transaction has been closed and so accessing data outside of what `TransactionData` covers requires a new transaction to be opened. `TransactionEventHandler` gets notified about transactions that have any change accessible via `TransactionData` so some indexing and schema changes will not be triggering these events.

Chapter 19. Data Import

For importing data using Cypher and CSV, see [Section 12.8, “Importing CSV files with Cypher” \[213\]](#).

For high-performance data import, see [Chapter 29, *Import tool* \[531\]](#).

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*.

The graph algorithms are included with Neo4j.

For usage examples, see [Section 21.18, “Graph Algorithms” \[402\]](#) (REST API) and [Section 33.10, “Graph Algorithm examples” \[600\]](#) (embedded database). The shortest path algorithm can be used from Cypher as well, see [the section called “Shortest path” \[163\]](#).

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” \[317\]](#) 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 \[96\]](#).

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” \[318\]](#) 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” \[319\]](#)) 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” \[215\]](#)) 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” \[300\]](#) 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” \[113\]](#) 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" : [ 18 ]  
    } ]  
  },  
  "errors" : [ ]  
}
```

Execute multiple statements

You can send multiple Cypher statements in the same request. The response will contain the result of each statement.

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)"  
  }, {  
    "statement" : "CREATE (n {props}) RETURN n",  
    "parameters" : {  
      "props" : {  
        "name" : "My Node"  
      }  
    }  
  } ]  
}
```

Example response

- 200: OK
- Content-Type: `application/json`

```
{  
  "results" : [ {  
    "columns" : [ "id(n)" ],  
    "data" : [ {  
      "row" : [ 14 ]  
    } ]  
  }, {  
    "columns" : [ "n" ],  
    "data" : [ {  
      "row" : [ {  
        "name" : "My Node"  
      } ]  
    } ]  
  },  
  "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/9>

```
{  
  "commit": "http://localhost:7474/db/data/transaction/9/commit",  
  "results": [ {  
    "columns": [ "n" ],  
    "data": [ [ {  
      "row": [ {  
        "name": "My Node"  
      } ]  
    } ]  
  },  
  "transaction": {  
    "expires": "Tue, 10 Nov 2015 13:14:14 +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/11>
- 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/11/commit",
```

```

"results" : [ {
  "columns" : [ "n" ],
  "data" : [ {
    "row" : [ { } ]
  } ],
} ],
"transaction" : {
  "expires" : "Tue, 10 Nov 2015 13:14:14 +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",
        "all_typed_relationships" : "http://localhost:7474/db/data/node/12/relationships/all/{-list|&|types}",
        "traverse" : "http://localhost:7474/db/data/node/12/traverse/{returnType}",
        "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" : "Tue, 10 Nov 2015 13:14:10 +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" : "Tue, 10 Nov 2015 13:14:13 +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/6/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" : [ 17 ]  
    } ]  
  },  
  "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" : [ ]  
}
```

Include query statistics

By setting `includeStats` to `true` for a statement, query statistics will be returned for it.

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)",  
    "includeStats" : true  
  } ]  
}
```

Example response

- 200: OK
- Content-Type: `application/json`

```
{  
  "results" : [ {  
    "columns" : [ "id(n)" ],  
    "data" : [ {  
      "row" : [ 16 ]  
    } ],  
    "stats" : {  
      "contains_updates" : true,  
      "nodes_created" : 1,  
      "nodes_deleted" : 0,  
      "nodes_returned" : 1,  
      "query_time_ms" : 0,  
      "statements_executed" : 1,  
      "transactions_committed" : 1  
    }  
  } ]  
}
```

```

    "properties_set" : 0,
    "relationships_created" : 0,
    "relationship_deleted" : 0,
    "labels_added" : 0,
    "labels_removed" : 0,
    "indexes_added" : 0,
    "indexes_removed" : 0,
    "constraints_added" : 0,
    "constraints_removed" : 0
  }
}

} ],
"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" : "19",

```

```

    "labels" : [ "Bike" ],
    "properties" : {
      "weight" : 10
    }
  }, {
    "id" : "21",
    "labels" : [ "Wheel" ],
    "properties" : {
      "spokes" : 32
    }
  }, {
    "id" : "20",
    "labels" : [ "Wheel" ],
    "properties" : {
      "spokes" : 3
    }
  }],
  "relationships" : [ {
    "id" : "9",
    "type" : "HAS",
    "startNode" : "19",
    "endNode" : "20",
    "properties" : {
      "position" : 1
    }
  }, {
    "id" : "10",
    "type" : "HAS",
    "startNode" : "19",
    "endNode" : "21",
    "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” \[309\]](#).

Example request

- POST `http://localhost:7474/db/data/transaction/10/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\"\\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.	Rollback
ClientNotification	There are notifications about the request sent by the client.	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.	Rollback

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.
Neo.ClientError.Transaction.MarkedAsFailed	Transaction was marked as both successful and failed. Failure takes precedence and so this

Status Code	Description
	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.ClientNotification.Statement.CartesianProduct	This query builds a cartesian product between disconnected patterns.
Neo.ClientNotification.Statement.DeprecationWarning	This feature is deprecated and will be removed in future versions.
Neo.ClientNotification.Statement.DynamicPropertyWarning	Queries using dynamic properties will use neither index seeks nor index scans for those properties
Neo.ClientNotification.Statement.EagerWarning	The execution plan for this query contains the Eager operator, which forces all dependent data to be materialized in main memory before proceeding
Neo.ClientNotification.Statement.IndexMissingWarning	Adding a schema index may speed up this query.
Neo.ClientNotification.Statement.JoinHintUnfulfillableWarning	The database was unable to plan a hinted join.
Neo.ClientNotification.Statement.JoinHintUnsupportedWarning	Queries with join hints are not supported by the RULE planner.
Neo.ClientNotification.Statement.LabelMissingWarning	The provided label is not in the database.
Neo.ClientNotification.Statement.PlannerUnsupportedWarning	This query is not supported by the COST planner.
Neo.ClientNotification.Statement.PropertyNameMissingWarning	The provided property name is not in the database
Neo.ClientNotification.Statement.RelTypeMissingWarning	The provided relationship type is not in the database.
Neo.ClientNotification.Statement.RuntimeUnsupportedWarning	This query is not supported by the compiled runtime.
Neo.ClientNotification.Statement.UnboundedPatternWarning	The provided pattern is unbounded, consider adding an upper limit to the number of node hops.
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.DuplicateSchemaRule	The request referred to a schema rule that defined multiple times.

Status Code	Description
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.
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.ConstraintsChanged	Database constraints changed since the start of this transaction
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" \[503\]](#) 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" \[314\]](#)) 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" \[316\]](#)).

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” \[314\]](#) 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" : "2.3.1"  
}
```

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" : "2.3.1"  
}
```

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.

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" : "2.3.1"  
}
```

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” \[300\]](#)). 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” \[102\]](#). 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

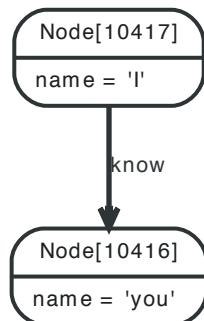
In order to speed up queries in repeated scenarios, try not to use literals but replace them with parameters wherever possible in order to let the server cache query plans, see [the section called “Use parameters” \[319\]](#) for details. Also see [Section 8.5, “Parameters” \[113\]](#) for where parameters 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 21.2. 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

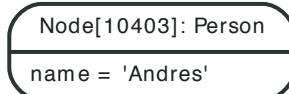
Create a node with a label and a property using Cypher. See the request for the parameter sent with the query.

```

CREATE (n:Person { name : { name } })
RETURN n

```

Figure 21.3. 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

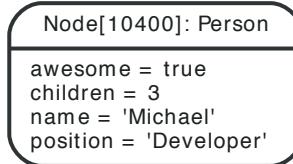
```
{
  "columns" : [ "n" ],
  "data" : [ [ {
    "labels" : "http://localhost:7474/db/data/node/10403/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10403/relationships/out",
    "data" : {
      "name" : "Andres"
    },
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10403/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/10403/traverse/{returnType}",
    "self" : "http://localhost:7474/db/data/node/10403",
    "property" : "http://localhost:7474/db/data/node/10403/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10403/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10403/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10403/relationships/in",
    "extensions" : { },
    "create_relationship" : "http://localhost:7474/db/data/node/10403/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/10403/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10403/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10403/relationships/in/-list|&|types",
    "metadata" : {
      "id" : 10403,
      "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 21.4. 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 { 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" : [ [ {
    "labels" : "http://localhost:7474/db/data/node/10400/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10400/relationships/out",
    "data" : {
      "position" : "Developer",
      "awesome" : true,
      "name" : "Michael",
      "children" : 3
    },
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10400/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/10400/traverse/{returnType}",
    "self" : "http://localhost:7474/db/data/node/10400",
    "property" : "http://localhost:7474/db/data/node/10400/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10400/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10400/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10400/relationships/in",
    "extensions" : { },
    "create_relationship" : "http://localhost:7474/db/data/node/10400/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/10400/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10400/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10400/relationships/in/-list|&|types",
    "metadata" : {
    }
  } ]
}
```

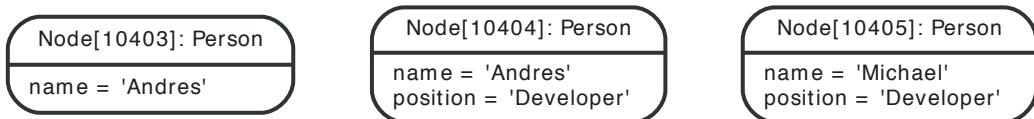
```
{
  "id" : 10400,
  "labels" : [ "Person" ]
}
}
}
```

Create multiple nodes with properties

Create multiple nodes with properties using Cypher. See the request for the parameter sent with the query.

```
UNWIND { props } AS map
CREATE (n:Person)
SET n = map
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" : "UNWIND {props} as map CREATE (n:Person) SET n = map 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" : [ [ {
    "labels" : "http://localhost:7474/db/data/node/10404/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10404/relationships/out",
    "data" : {
      "position" : "Developer",
      "name" : "Andres"
    },
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10404/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/10404/traverse/{returnType}",
    "self" : "http://localhost:7474/db/data/node/10404",
    "property" : "http://localhost:7474/db/data/node/10404/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10404/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10404/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10404/relationships/in",
    "extensions" : { }
  } ] ]
}
```

```

"create_relationship" : "http://localhost:7474/db/data/node/10404/relationships",
"paged_traverse" : "http://localhost:7474/db/data/node/10404/paged/traverse/{returnType}{?pageSize,leaseTime}",
"all_relationships" : "http://localhost:7474/db/data/node/10404/relationships/all",
"incoming_typed_relationships" : "http://localhost:7474/db/data/node/10404/relationships/in/{-list|&|types}",
"metadata" : {
  "id" : 10404,
  "labels" : [ "Person" ]
}
},
],
[ {
  "labels" : "http://localhost:7474/db/data/node/10405/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/10405/relationships/out",
  "data" : {
    "position" : "Developer",
    "name" : "Michael"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/10405/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/10405/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/10405",
  "property" : "http://localhost:7474/db/data/node/10405/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/10405/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10405/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/10405/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/10405/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/10405/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/10405/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10405/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 10405,
    "labels" : [ "Person" ]
  }
}
]
}

```

Set all properties on a node using Cypher

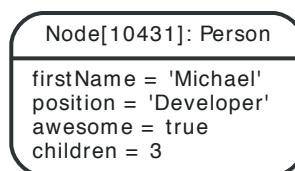
Set all properties on a node.

```

CREATE (n:Person { name: 'this property is to be deleted' })
SET n = { props }
RETURN n

```

Figure 21.6. 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 { 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

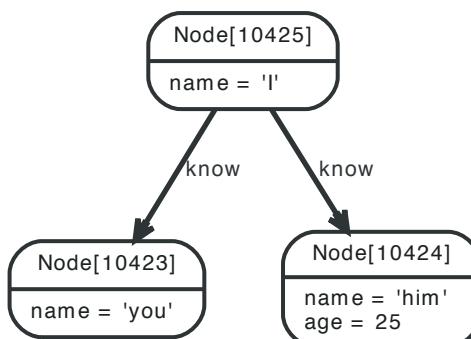
```
{
  "columns" : [ "n" ],
  "data" : [ [ {
    "labels" : "http://localhost:7474/db/data/node/10431/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10431/relationships/out",
    "data" : {
      "position" : "Developer",
      "awesome" : true,
      "children" : 3,
      "firstName" : "Michael"
    },
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10431/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/10431/traverse/{returnType}",
    "self" : "http://localhost:7474/db/data/node/10431",
    "property" : "http://localhost:7474/db/data/node/10431/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10431/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10431/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10431/relationships/in",
    "extensions" : { },
    "create_relationship" : "http://localhost:7474/db/data/node/10431/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/10431/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10431/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10431/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 10431,
      "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 21.7. 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

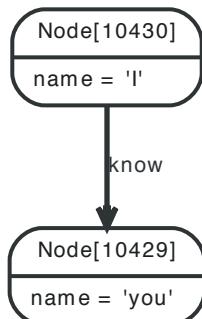
```
{
  "columns" : [ "type(r)", "n.name", "n.age" ],
  "data" : [ [ "know", "you", null ], [ "know", "him", 25 ] ]
}
```

Return paths

Paths can be returned just like other return types.

```
MATCH path = (x { name: 'I' })--(friend)
RETURN path, friend.name
```

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 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/10430",
    "nodes" : [ "http://localhost:7474/db/data/node/10430", "http://localhost:7474/db/data/node/10429" ],
    "length" : 1,
    "relationships" : [ "http://localhost:7474/db/data/relationship/200" ],
    "end" : "http://localhost:7474/db/data/node/10429"
  } ] }
```

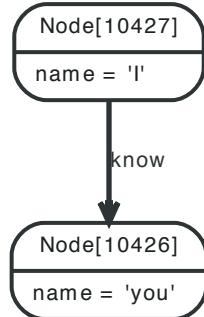
```
  }, "you" ] ]
}
```

Nested results

When sending queries that return nested results like lists 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 21.9. 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

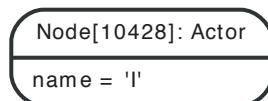
```
{
  "columns" : [ "collect(n.name)" ],
  "data" : [ [ [ "you", "I" ] ] ]
}
```

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 21.10. 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`

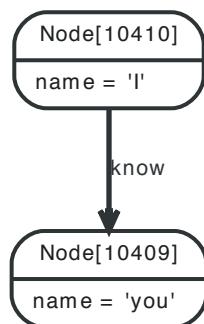
```
{
  "columns" : [ "labels(n)" ],
  "data" : [ [ [ "Actor" ] ] ],
  "stats" : {
    "relationships_created" : 0,
    "nodes_deleted" : 0,
    "relationship_deleted" : 0,
    "indexes_added" : 0,
    "properties_set" : 0,
    "constraints_removed" : 0,
    "indexes_removed" : 0,
    "labels_removed" : 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 21.11. 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:245)",
    "org.neo4j.server.rest.repr.OutputFormat.formatRepresentation(OutputFormat.java:177)",
    "org.neo4j.server.rest.repr.OutputFormat.response(OutputFormat.java:160)",
    "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.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectorsUserAgentFilter.doFilter(CollectorsUserAgentFilter.java:69)",
    "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"
          }
        ],
        "cause": {
          "errors": [
            {
              "code": "Neo.DatabaseError.General.UnknownFailure",
              "stackTrace": "org.neo4j.cypher.internal.frontend.v2_3.ArithmetricException\n\tat org.neo4j.cypher.internal.compiler.v2_3.commands.expressions.Divide.apply(Divide.scala:36)\n\tat org.neo4j.cypher.internal.compiler.v2_3.pipes.ProjectionPipe$$anonfun$internalCreateResults$1$$anonfun$apply$1.apply(ProjectionPipe.scala:48)\n\tat org.neo4j.cypher.internal.compiler.v2_3.pipes.ProjectionPipe$$anonfun$internalCreateResults$1$$anonfun$apply$1.apply(ProjectionPipe.scala:46)\n\tat scala.collection.immutable.Map"
        }
      ]
    }
  }
}
```

```
$Map1.foreach(Map.scala:116)\n\tat org.neo4j.cypher.internal.compiler.v2_3.pipes.ProjectionPipe$$anonfun$internalCreateResults$1.apply(ProjectionPipe.scala:46)\n\tat org.neo4j.cypher.internal.compiler.v2_3.pipes.ProjectionPipe$$anonfun$internalCreateResults$1.apply(ProjectionPipe.scala:45)\n\tat scala.collection.Iterator$$anon$11.next(Iterator.scala:370)\n\tat org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator$$anonfun$next$1.apply(ResultIterator.scala:75)\n\tat org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator$$anonfun$next$1.apply(ResultIterator.scala:72)\n\tat org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator$$anonfun$failIfThrows$1.apply(ResultIterator.scala:121)\n\tat org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator.decoratedCypherException(ResultIterator.scala:130)\n\tat org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator.failIfThrows(ResultIterator.scala:119)\n\tat org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator.next(ResultIterator.scala:72)\n\tat org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator.next(ResultIterator.scala:50)\n\tat org.neo4j.cypher.internal.compiler.v2_3.PipeExecutionResult.next(PipeExecutionResult.scala:77)\n\tat org.neo4j.cypher.internal.compiler.v2_3.PipeExecutionResult$$anon$2.next(PipeExecutionResult.scala:70)\n\tat org.neo4j.cypher.internal.compiler.v2_3.PipeExecutionResult$$anon$2.next(PipeExecutionResult.scala:68)\n\tat org.neo4j.cypher.internal.compatibility.ExecutionResultWrapperFor2_3$$anon$1$$anonfun$next$1.apply(CompatibilityFor2_3.scala:226)\n\tat org.neo4j.cypher.internal.compatibility.ExecutionResultWrapperFor2_3$$anon$1$$anonfun$next$1.apply(CompatibilityFor2_3.scala:226)\n\tat org.neo4j.cypher.internal.compatibility.exceptionHandlerFor2_3$.runSafely(CompatibilityFor2_3.scala:114)\n\tat org.neo4j.cypher.internal.compatibility.ExecutionResultWrapperFor2_3$$anon$1.next(CompatibilityFor2_3.scala:226)\n\tat org.neo4j.cypher.internal.compatibility.ExecutionResultWrapperFor2_3$$anon$1.next(CompatibilityFor2_3.scala:221)\n\tat org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:233)\n\tat org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:55)\n\tat org.neo4j.helpers.collection.ExceptionHandlingIterable$1.next(ExceptionHandlingIterable.java:53)\n\tat org.neo4j.helpers.collection.IteratorWrapper.next(IteratorWrapper.java:47)\n\tat org.neo4j.server.rest.repr.ListRepresentation.serialize(ListRepresentation.java:64)\n\tat org.neo4j.server.rest.repr.Serializer.serialize(Serializer.java:75)\n\tat org.neo4j.server.rest.repr.MappingSerializer.putList(MappingSerializer.java:61)\n\tat org.neo4j.server.rest.repr.CypherResultRepresentation.serialize(CypherResultRepresentation.java:58)\n\tat org.neo4j.server.rest.repr.MappingRepresentation.serialize(MappingRepresentation.java:41)\n\tat org.neo4j.server.rest.repr.OutputFormat.assemble(OutputFormat.java:245)\n\tat org.neo4j.server.rest.repr.OutputFormat.formatRepresentation(OutputFormat.java:177)\n\tat org.neo4j.server.rest.repr.OutputFormat.response(OutputFormat.java:160)\n\tat org.neo4j.server.rest.repr.OutputFormat.ok(OutputFormat.java:73)\n\tat org.neo4j.server.web.CypherService.cypher(CypherService.java:127)\n\tat sun.reflect.GeneratedMethodAccessor103.invoke(Unknown Source)\n\tat sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)\n\tat java.lang.reflect.Method.invoke(Method.java:606)\n\tat com.sun.jersey.spi.container.JavaMethodInvokerFactory$1.invoke(JavaMethodInvokerFactory.java:60)\n\tat com.sun.jersey.server.impl.model.method.dispatch.AbstractResourceMethodDispatchProvider$ResponseOutInvoker._dispatch(AbstractResourceMethodDispatchProvider.java:205)\n\tat com.sun.jersey.server.impl.model.method.dispatch.ResourceJavaMethodDispatcher.dispatch(ResourceJavaMethodDispatcher.java:75)\n\tat org.neo4j.server.rest.transactional.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)\n\tat com.sun.jersey.server.impl.uri.rules.HttpMethodRule.accept(HttpMethodRule.java:302)\n\tat com.sun.jersey.server.impl.uri.rules.ResourceClassRule.accept(ResourceClassRule.java:108)\n\tat com.sun.jersey.server.impl.uri.rules.RightHandPathRule.accept(RightHandPathRule.java:147)\n\tat com.sun.jersey.server.impl.uri.rules.RootResourceClassesRule.accept(RootResourceClassesRule.java:84)\n\tat com.sun.jersey.server.impl.application.WebApplicationImpl._handleRequest(WebApplicationImpl.java:1542)\n\tat com.sun.jersey.server.impl.application.WebApplicationImpl._handleRequest(WebApplicationImpl.java:1473)\n\tat com.sun.jersey.server.impl.application.WebApplicationImpl.handleRequest(WebApplicationImpl.java:1419)\n\tat com.sun.jersey.server.impl.application.WebApplicationImpl.handleRequest(WebApplicationImpl.java:1409)\n\tat com.sun.jersey.spi.container.servlet.WebComponent.service(WebComponent.java:409)\n\tat com.sun.jersey.spi.container.servlet.ServletContainer.service(ServletContainer.java:558)\n\tat com.sun.jersey.spi.container.servlet.ServletContainer.service(ServletContainer.java:733)\n\tat javax.servlet.http.HttpServlet.service(HttpServlet.java:790)\n\tat org.eclipse.jetty.servlet.ServletHolder.handle(ServletHolder.java:800)\n\tat org.eclipse.jetty.servlet.ServletHandler$CachedChain.doFilter(ServletHandler.java:1669)\n\tat org.neo4j.server.rest.web.CollectorsUserAgentFilter.doFilter(CollectorsUserAgentFilter.java:69)\n\tat org.eclipse.jetty.servlet.ServletHandler$CachedChain.doFilter(ServletHandler.java:1652)\n\tat org.eclipse.jetty.servlet.ServletHandler.doHandle(ServletHandler.java:585)\n\tat org.eclipse.jetty.server.session.SessionHandler.doHandle(SessionHandler.java:221)\n\tat org.eclipse.jetty.server.handler.ContextHandler.doHandle(ContextHandler.java:1125)\n\tat org.eclipse.jetty.servlet.ServletHandler.doScope(ServletHandler.java:515)\n\tat org.eclipse.jetty.server.session.SessionHandler.doScope(SessionHandler.java:185)\n\tat org.eclipse.jetty.server.handler.ContextHandler.doScope(ContextHandler.java:1059)\n\tat org.eclipse.jetty.server.handler.ScopedHandler.handle(ScopedHandler.java:141)\n
```

```

\tat org.eclipse.jetty.server.handler.HandlerList.handle(HandlerList.java:52)\n\tat
org.eclipse.jetty.server.handler.HandlerWrapper.handle(HandlerWrapper.java:97)\n\tat
org.eclipse.jetty.server.Server.handle(Server.java:497)\n\tat
org.eclipse.jetty.server.HttpChannel.handle(HttpChannel.java:310)\n\tat
org.eclipse.jetty.server.HttpConnection.onFillable(HttpConnection.java:248)\n\tat org.eclipse.jetty.io.AbstractConnection
$2.run(AbstractConnection.java:540)\n\tat org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:620)\n
\tat org.eclipse.jetty.util.thread.QueuedThreadPool$3.run(QueuedThreadPool.java:540)\n\tat
java.lang.Thread.run(Thread.java:745)\n"
}

],
"exception": "ArithmaticException",
"fullname": "org.neo4j.cypher.internal.frontend.v2_3.ArithmaticException",
"stackTrace": [
    "org.neo4j.cypher.internal.compiler.v2_3.commands.expressions.Divide.apply(Divide.scala:36)",
    "org.neo4j.cypher.internal.compiler.v2_3.pipes.ProjectionPipe$$anonfun$internalCreateResults$1$$anonfun$apply
$1.apply(ProjectionPipe.scala:48)",
    "org.neo4j.cypher.internal.compiler.v2_3.pipes.ProjectionPipe$$anonfun$internalCreateResults$1$$anonfun$apply
$1.apply(ProjectionPipe.scala:46)",
    "scala.collection.immutable.Map$Map1.foreach(Map.scala:116)",
    "org.neo4j.cypher.internal.compiler.v2_3.pipes.ProjectionPipe$$anonfun$internalCreateResults
$1.apply(ProjectionPipe.scala:46)",
    "org.neo4j.cypher.internal.compiler.v2_3.pipes.ProjectionPipe$$anonfun$internalCreateResults
$1.apply(ProjectionPipe.scala:45)",
    "scala.collection.Iterator$$anon$11.next(Iterator.scala:370)",
    "scala.collection.Iterator$$anon$11.next(Iterator.scala:370)",
    "org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator$$anonfun$next$1.apply(ResultIterator.scala:75)",
    "org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator$$anonfun$next$1.apply(ResultIterator.scala:72)",
    "org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator$$anonfun$failIfThrows$1.apply(ResultIterator.scala:121)",
    "org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator.decoratedCypherException(ResultIterator.scala:130)",
    "org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator.failIfThrows(ResultIterator.scala:119)",
    "org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator.next(ResultIterator.scala:72)",
    "org.neo4j.cypher.internal.compiler.v2_3.ClosingIterator.next(ResultIterator.scala:50)",
    "org.neo4j.cypher.internal.compiler.v2_3.PipeExecutionResult.next(PipeExecutionResult.scala:77)",
    "org.neo4j.cypher.internal.compiler.v2_3.PipeExecutionResult$$anon$2.next(PipeExecutionResult.scala:70)",
    "org.neo4j.cypher.internal.compiler.v2_3.PipeExecutionResult$$anon$2.next(PipeExecutionResult.scala:68)",
    "org.neo4j.cypher.internal.compatibility.ExecutionResultWrapperFor2_3$$anon$1$$anonfun$next
$1.apply(CompatibilityFor2_3.scala:226)",
    "org.neo4j.cypher.internal.compatibility.ExecutionResultWrapperFor2_3$$anon$1$$anonfun$next
$1.apply(CompatibilityFor2_3.scala:226)",
    "org.neo4j.cypher.internal.compatibility.exceptionHandlerFor2_3$.runSafely(CompatibilityFor2_3.scala:114)",
    "org.neo4j.cypher.internal.compatibility.ExecutionResultWrapperFor2_3$$anon
$1.next(CompatibilityFor2_3.scala:226)",
    "org.neo4j.cypher.internal.compatibility.ExecutionResultWrapperFor2_3$$anon
$1.next(CompatibilityFor2_3.scala:221)",
    "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:233)",
    "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:55)",
    "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:245)",
    "org.neo4j.server.rest.repr.OutputFormat.formatRepresentation(OutputFormat.java:177)",
    "org.neo4j.server.rest.repr.OutputFormat.response(OutputFormat.java:160)",
    "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)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
    "java.lang.Thread.run(Thread.java:745)"
]
},

```

```

"exception": "ArithmeticsException",
"fullname": "org.neo4j.cypher.ArithmeticsException",
"stackTrace": [
    "org.neo4j.cypher.internal.compatibility.exceptionHandlerFor2_3$.arithmeticsException(CompatibilityFor2_3.scala:61)",
    "org.neo4j.cypher.internal.compatibility.exceptionHandlerFor2_3$.arithmeticsException(CompatibilityFor2_3.scala:58)",
    "org.neo4j.cypher.internal.frontend.v2_3.ArithmeticsException.mapToPublic(CypherException.scala:111)",
    "org.neo4j.cypher.internal.compatibility.exceptionHandlerFor2_3$.runSafely(CompatibilityFor2_3.scala:119)",
    "org.neo4j.cypher.internal.compatibility.ExecutionResultWrapperFor2_3$anon$1.next(CompatibilityFor2_3.scala:226)",
    "org.neo4j.cypher.internal.compatibility.ExecutionResultWrapperFor2_3$anon$1.next(CompatibilityFor2_3.scala:221)",
    "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:233)",
    "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:55)",
    "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:245)",
    "org.neo4j.server.rest.repr.OutputFormat.formatRepresentation(OutputFormat.java:177)",
    "org.neo4j.server.rest.repr.OutputFormat.response(OutputFormat.java:160)",
    "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)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
    "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:391)",
    "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:237)",
    "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:55)",
    "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:245)",
    "org.neo4j.server.rest.repr.OutputFormat.formatRepresentation(OutputFormat.java:177)",
    "org.neo4j.server.rest.repr.OutputFormat.response(OutputFormat.java:160)",
    "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)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
    "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:391)",
    "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:237)",
    "org.neo4j.cypher.javacompat.ExecutionResult.next(ExecutionResult.java:55)",
    "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:245)",
"org.neo4j.server.rest.repr.OutputFormat.formatRepresentation(OutputFormat.java:177)",
"org.neo4j.server.rest.repr.OutputFormat.response(OutputFormat.java:160)",
"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.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
"org.neo4j.server.rest.web.CollectorsUserAgentFilter.doFilter(CollectorsUserAgentFilter.java:69)",
"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

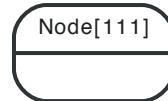
- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
[ "PYear", "name", "ReleaseRegion", "ParentalAdvisory", "some-key", "#öäüÖÜß", "since", "isbn", "LabelName", "MediaCount",  
"ArtistString", "position", "MainArtistAlias", "Title", "array1", "Name", "ID", "array2", "IsCanon", "happy", "TrackCount",  
"Cline", "Duration", "children", "dummy", "cost", "kvkey2", "non-existent", "a", "id", "ReleaseYear", "sequence", "i",  
"CYear", "awesome", "FormatNames", "IsLimitedEdition", "ReleaseDate", "kvkey1", "DecibelID", "age", "prop", "Pline",  
"firstName", "foo", "OriginalReleaseDate", "array" ]
```

21.8. Nodes

Create node

Figure 21.12. Final Graph



Example request

- POST <http://localhost:7474/db/data/node>
- Accept: application/json; charset=UTF-8

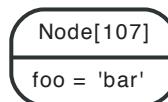
Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8
- Location: <http://localhost:7474/db/data/node/111>

```
{
  "extensions" : { },
  "labels" : "http://localhost:7474/db/data/node/111/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/111/relationships/out",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/111/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/111/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/111",
  "property" : "http://localhost:7474/db/data/node/111/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/111/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/111/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/111/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/111/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/111/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/111/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/111/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 111,
    "labels" : [ ]
  },
  "data" : { }
}
```

Create node with properties

Figure 21.13. 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

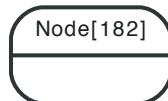
- 201: Created
- Content-Length: 1237
- Content-Type: application/json; charset=UTF-8
- Location: http://localhost:7474/db/data/node/107

```
{
  "extensions" : { },
  "labels" : "http://localhost:7474/db/data/node/107/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/107/relationships/out",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/107/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/107/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/107",
  "property" : "http://localhost:7474/db/data/node/107/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/107/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/107/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/107/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/107/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/107/paged/traverse{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/107/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/107/relationships/in/-list|&|types",
  "metadata" : {
    "id" : 107,
    "labels" : [ ]
  },
  "data" : {
    "foo" : "bar"
  }
}
```

Get node

Note that the response contains URI/templates for the available operations for getting properties and relationships.

Figure 21.14. Final Graph



Example request

- GET http://localhost:7474/db/data/node/182
- Accept: application/json; charset=UTF-8

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

```
{
  "extensions" : { },
  "labels" : "http://localhost:7474/db/data/node/182/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/182/relationships/out",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/182/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/182/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/182",
  "property" : "http://localhost:7474/db/data/node/182/properties/{key}"}
```

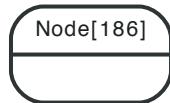
```

"properties" : "http://localhost:7474/db/data/node/182/properties",
"outgoing_typed_relationships" : "http://localhost:7474/db/data/node/182/relationships/out/{-list|&|types}",
"incoming_relationships" : "http://localhost:7474/db/data/node/182/relationships/in",
"create_relationship" : "http://localhost:7474/db/data/node/182/relationships",
"paged_traverse" : "http://localhost:7474/db/data/node/182/paged/traverse/{returnType}{?pageSize,leaseTime}",
"all_relationships" : "http://localhost:7474/db/data/node/182/relationships/all",
"incoming_typed_relationships" : "http://localhost:7474/db/data/node/182/relationships/in/{-list|&|types}",
"metadata" : {
  "id" : 182,
  "labels" : [ ]
},
"data" : { }
}

```

Get non-existent node

Figure 21.15. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/18600000`
- Accept: `application/json; charset=UTF-8`

Example response

- 404: Not Found
- Content-Type: `application/json; charset=UTF-8`

```
{
  "message": "Cannot find node with id [18600000] in database.",
  "exception": "NodeNotFoundException",
  "fullname": "org.neo4j.server.rest.web.NodeNotFoundException",
  "stackTrace": [
    "org.neo4j.server.rest.web.DatabaseActions.node(DatabaseActions.java:174)",
    "org.neo4j.server.rest.web.DatabaseActions.getNode(DatabaseActions.java:219)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.getNode(RestfulGraphDatabase.java:279)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
    "java.lang.Thread.run(Thread.java:745)"
  ],
  "cause": {
    "message": "Node 18600000 not found",
    "errors": [
      {
        "message": "Node 18600000 not found",
        "code": "Neo.ClientError.Statement.EntityNotFound"
      }
    ],
    "cause": {
      "message": "Unable to load NODE with id 18600000.",
      "errors": [
        {
          "message": "Unable to load NODE with id 18600000.",
          "code": "Neo.ClientError.Statement.EntityNotFound"
        }
      ],
      "exception": "EntityNotFoundException",
      "fullname": "org.neo4j.kernel.api.exceptions.EntityNotFoundException",
    }
  }
}
```

```

"stackTrace": [
    "org.neo4j.kernel.impl.factory.GraphDatabaseFacade.getNodeById(GraphDatabaseFacade.java:228)",
    "org.neo4j.server.rest.web.DatabaseActions.node(DatabaseActions.java:170)",
    "org.neo4j.server.rest.web.DatabaseActions.getNode(DatabaseActions.java:219)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.getNode(RestfulGraphDatabase.java:279)",
    "java.lang.reflect.Method.invoke(Method.java:606)",

"org.neo4j.server.rest.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
    "java.lang.Thread.run(Thread.java:745)"
]
},
"exception": "NotFoundException",
"fullname": "org.neo4j.graphdb.NotFoundException",
"stackTrace": [
    "org.neo4j.kernel.impl.factory.GraphDatabaseFacade.getNodeById(GraphDatabaseFacade.java:228)",
    "org.neo4j.server.rest.web.DatabaseActions.node(DatabaseActions.java:170)",
    "org.neo4j.server.rest.web.DatabaseActions.getNode(DatabaseActions.java:219)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.getNode(RestfulGraphDatabase.java:279)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
    "java.lang.Thread.run(Thread.java:745)"
]
},
"errors": [
{
    "message": "Cannot find node with id [18600000] in database.",
    "code": "Neo.ClientError.Statement.EntityNotFound"
}
]
}

```

Delete node

Figure 21.16. Starting Graph

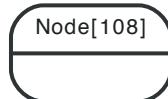


Figure 21.17. Final Graph

Example request

- DELETE `http://localhost:7474/db/data/node/108`
- Accept: `application/json; charset=UTF-8`

Example response

- 204: No Content

Nodes with relationships cannot be deleted

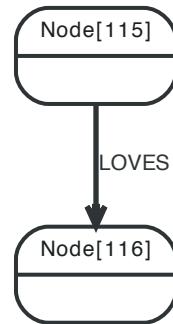
The relationships on a node has to be deleted before the node can be deleted.



Tip

You can use DETACH DELETE in Cypher to delete nodes and their relationships in one go.

Figure 21.18. Starting Graph

*Example request*

- DELETE <http://localhost:7474/db/data/node/115>
- Accept: application/json; charset=UTF-8

Example response

- 409: Conflict
- Content-Type: application/json; charset=UTF-8

```
{
  "message": "The node with id 115 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:228)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteNode(RestfulGraphDatabase.java:293)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.translational.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
    "java.lang.Thread.run(Thread.java:745)"
  ],
  "errors": [
    {
      "message": "The node with id 115 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.

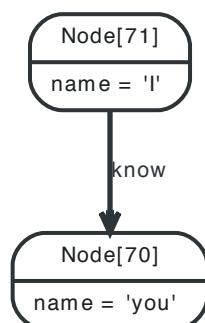
The general pattern to get relationships from a node is:

```
GET http://localhost:7474/db/data/node/123/relationships/{dir}/{-list|&|types}
```

Where `dir` is one of `all`, `in` and `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/46`
- 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/71",
  "property" : "http://localhost:7474/db/data/relationship/46/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/46",
  "properties" : "http://localhost:7474/db/data/relationship/46/properties",
  "type" : "know",
  "end" : "http://localhost:7474/db/data/node/70",
  "metadata" : {
    "id" : 46,
    "type" : "know"
  },
  "data" : { }
}
```

Create relationship

Upon successful creation of a relationship, the new relationship is returned.

Figure 21.20. Starting Graph

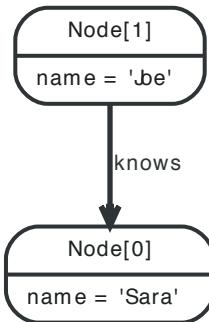
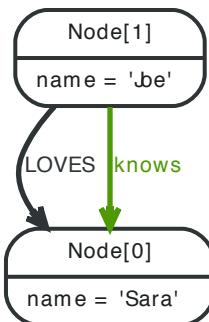


Figure 21.21. Final Graph

*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 21.22. Starting Graph

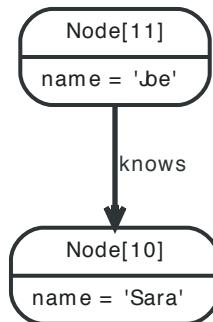
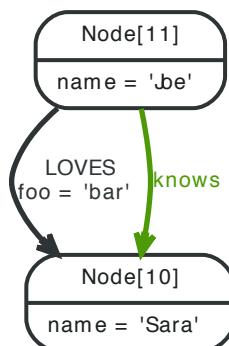


Figure 21.23. Final Graph

*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 21.24. Starting Graph

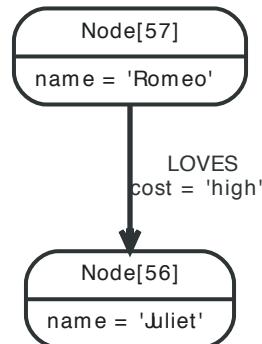


Figure 21.25. Final Graph



Example request

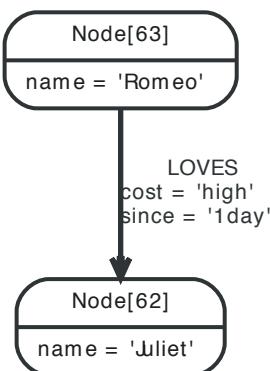
- DELETE <http://localhost:7474/db/data/relationship/39>
- Accept: application/json; charset=UTF-8

Example response

- 204: No Content

Get all properties on a relationship

Figure 21.26. Final Graph



Example request

- GET <http://localhost:7474/db/data/relationship/42/properties>
- Accept: application/json; charset=UTF-8

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

```
{
  "since" : "1day",
  "cost" : "high"
}
```

Set all properties on a relationship

Figure 21.27. Starting Graph

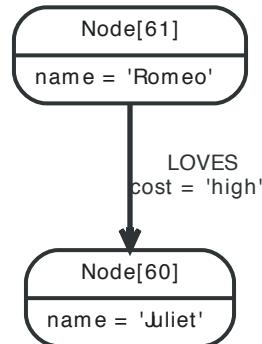
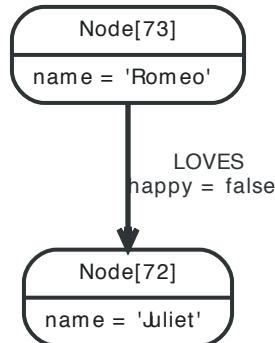


Figure 21.28. Final Graph



Example request

- PUT http://localhost:7474/db/data/relationship/47/properties
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

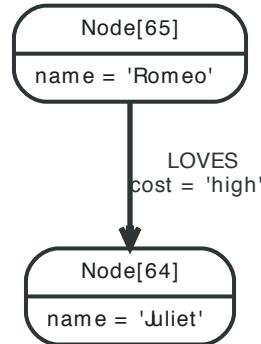
```
{
  "happy" : false
}
```

Example response

- 204: No Content

Get single property on a relationship

Figure 21.29. Final Graph



Example request

- GET `http://localhost:7474/db/data/relationship/43/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 21.30. Starting Graph

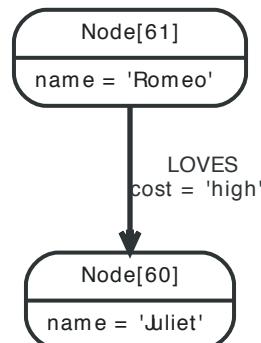
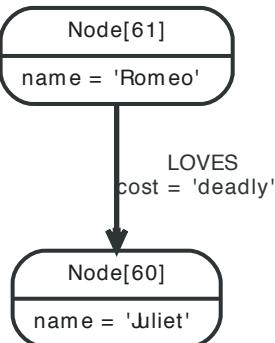


Figure 21.31. Final Graph



Example request

- PUT `http://localhost:7474/db/data/relationship/41/properties/cost`

- Accept: application/json; charset=UTF-8
- Content-Type: application/json

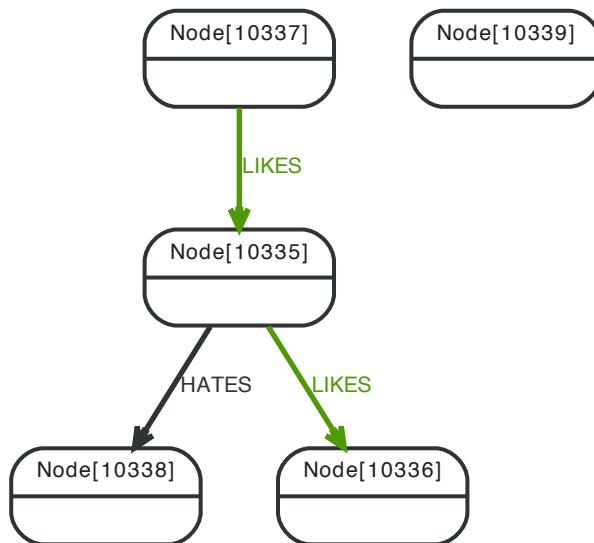
```
"deadly"
```

Example response

- 204: No Content

Get all relationships

Figure 21.32. Final Graph



Example request

- GET http://localhost:7474/db/data/node/10335/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/10335",
    "data" : { },
    "self" : "http://localhost:7474/db/data/relationship/152",
    "property" : "http://localhost:7474/db/data/relationship/152/properties/{key}",
    "properties" : "http://localhost:7474/db/data/relationship/152/properties",
    "type" : "HATES",
    "extensions" : { },
    "end" : "http://localhost:7474/db/data/node/10338",
    "metadata" : {
      "id" : 152,
      "type" : "HATES"
    }
  },
  {
    "start" : "http://localhost:7474/db/data/node/10337",
    "data" : { },
    "self" : "http://localhost:7474/db/data/relationship/151",
    "property" : "http://localhost:7474/db/data/relationship/151/properties/{key}",
    "properties" : "http://localhost:7474/db/data/relationship/151/properties",
    "type" : "LIKES"
  }
]
```

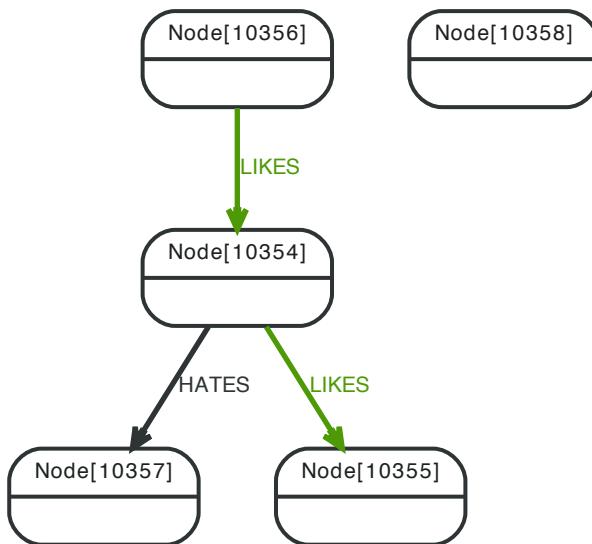
```

    "type" : "LIKES",
    "extensions" : { },
    "end" : "http://localhost:7474/db/data/node/10335",
    "metadata" : {
      "id" : 151,
      "type" : "LIKES"
    }
  },
  {
    "start" : "http://localhost:7474/db/data/node/10335",
    "data" : { },
    "self" : "http://localhost:7474/db/data/relationship/150",
    "property" : "http://localhost:7474/db/data/relationship/150/properties/{key}",
    "properties" : "http://localhost:7474/db/data/relationship/150/properties",
    "type" : "LIKES",
    "extensions" : { },
    "end" : "http://localhost:7474/db/data/node/10336",
    "metadata" : {
      "id" : 150,
      "type" : "LIKES"
    }
  }
]

```

Get incoming relationships

Figure 21.33. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/10354/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/10356",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/162",
  "property" : "http://localhost:7474/db/data/relationship/162/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/162/properties",
  "type" : "LIKES",
  "extensions" : { },
}
]

```

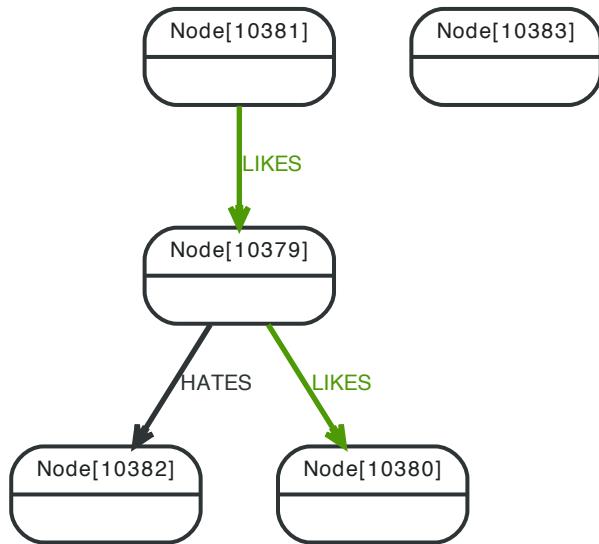
```

"end" : "http://localhost:7474/db/data/node/10354",
"metadata" : {
  "id" : 162,
  "type" : "LIKES"
}
} ]
}

```

Get outgoing relationships

Figure 21.34. Final Graph



Example request

- GET <http://localhost:7474/db/data/node/10379/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/10379",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/178",
  "property" : "http://localhost:7474/db/data/relationship/178/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/178/properties",
  "type" : "HATES",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/10382",
  "metadata" : {
    "id" : 178,
    "type" : "HATES"
  }
}, {
  "start" : "http://localhost:7474/db/data/node/10379",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/176",
  "property" : "http://localhost:7474/db/data/relationship/176/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/176/properties",
  "type" : "LIKES",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/10380",
  "metadata" : {
    "id" : 176,
    "type" : "LIKES"
  }
}
]

```

```

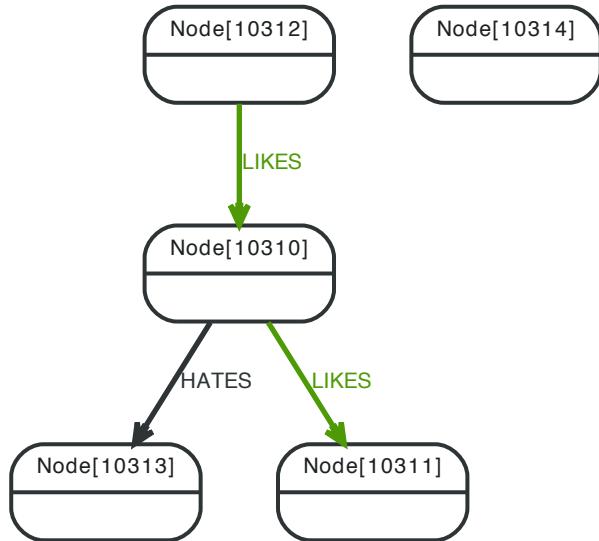
    "id" : 176,
    "type" : "LIKES"
}
} ]

```

Get typed relationships

Note that the "&" needs to be encoded like "%26" for example when using [cURL](#)² from the terminal.

Figure 21.35. Final Graph



Example request

- GET <http://localhost:7474/db/data/node/10310/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/10310",
    "data" : { },
    "self" : "http://localhost:7474/db/data/relationship/137",
    "property" : "http://localhost:7474/db/data/relationship/137/properties/{key}",
    "properties" : "http://localhost:7474/db/data/relationship/137/properties",
    "type" : "HATES",
    "extensions" : { },
    "end" : "http://localhost:7474/db/data/node/10313",
    "metadata" : {
        "id" : 137,
        "type" : "HATES"
    }
}, {
    "start" : "http://localhost:7474/db/data/node/10312",
    "data" : { },
    "self" : "http://localhost:7474/db/data/relationship/136",
    "property" : "http://localhost:7474/db/data/relationship/136/properties/{key}",
    "properties" : "http://localhost:7474/db/data/relationship/136/properties",
    "type" : "LIKES",
    "extensions" : { }
}
]

```

² <http://curl.haxx.se/>

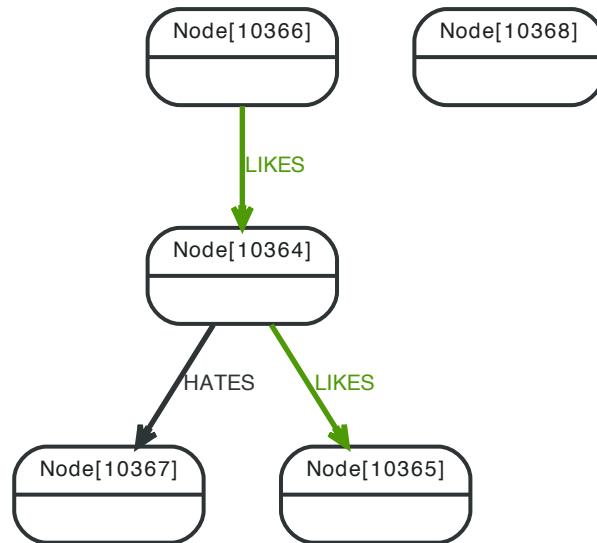
```

"end" : "http://localhost:7474/db/data/node/10310",
"metadata" : {
  "id" : 136,
  "type" : "LIKES"
}
}, {
  "start" : "http://localhost:7474/db/data/node/10310",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/135",
  "property" : "http://localhost:7474/db/data/relationship/135/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/135/properties",
  "type" : "LIKES",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/10311",
  "metadata" : {
    "id" : 135,
    "type" : "LIKES"
  }
}
]

```

Get relationships on a node without relationships

Figure 21.36. Final Graph



Example request

- GET <http://localhost:7474/db/data/node/10368/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`

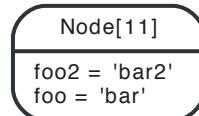
```
[ "KNOWS", "LOVES" ]
```

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 21.37. Final Graph



Example request

- PUT `http://localhost:7474/db/data/node/11/properties/foo`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
"bar"
```

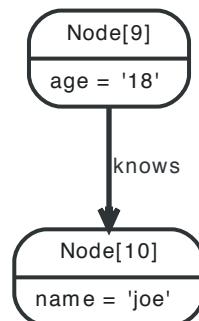
Example response

- 204: No Content

Update node properties

This will replace all existing properties on the node with the new set of attributes.

Figure 21.38. Final Graph



Example request

- PUT `http://localhost:7474/db/data/node/9/properties`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

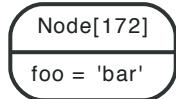
```
{
  "age" : "18"
}
```

Example response

- 204: No Content

Get properties for node

Figure 21.39. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/172/properties`
- Accept: `application/json; charset=UTF-8`

Example response

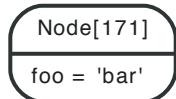
- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
{  
    "foo" : "bar"  
}
```

Get property for node

Get a single node property from a node.

Figure 21.40. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/171/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:205)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.createNode(RestfulGraphDatabase.java:252)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectorsUserAgentFilter.doFilter(CollectorsUserAgentFilter.java:69)",
    "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:205)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.createNode(RestfulGraphDatabase.java:252)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectorsUserAgentFilter.doFilter(CollectorsUserAgentFilter.java:69)",
    "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 21.41. Starting Graph

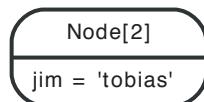
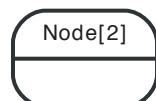


Figure 21.42. Final Graph



Example request

- DELETE <http://localhost:7474/db/data/node/2/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 21.43. Starting Graph

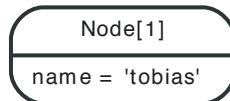
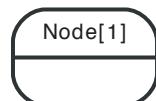


Figure 21.44. Final Graph



Example request

- DELETE <http://localhost:7474/db/data/node/1/properties/name>
- Accept: application/json; charset=UTF-8

Example response

- 204: No Content

21.12. Relationship properties

Update relationship properties

Figure 21.45. Starting Graph

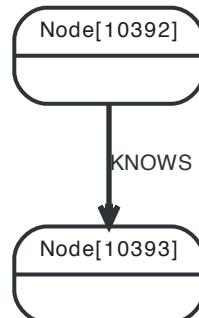
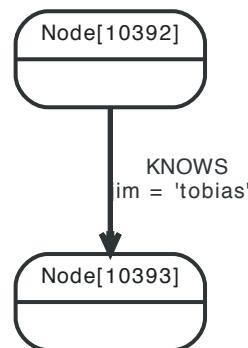


Figure 21.46. Final Graph



Example request

- PUT <http://localhost:7474/db/data/relationship/183/properties>
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{  
  "jim" : "tobias"  
}
```

Example response

- 204: No Content

Remove properties from a relationship

Figure 21.47. Starting Graph

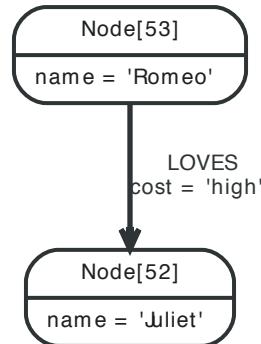
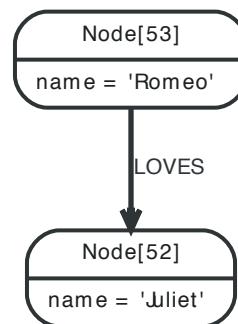


Figure 21.48. Final Graph



Example request

- `DELETE http://localhost:7474/db/data/relationship/37/properties`
- `Accept: application/json; charset=UTF-8`

Example response

- `204: No Content`

Remove property from a relationship

See the example request below.

Figure 21.49. Starting Graph

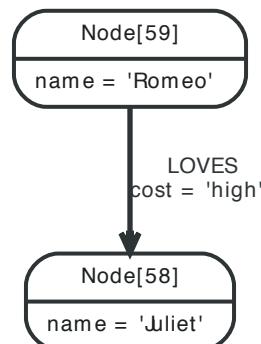
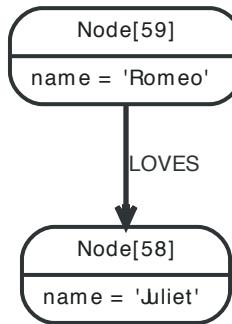


Figure 21.50. Final Graph*Example request*

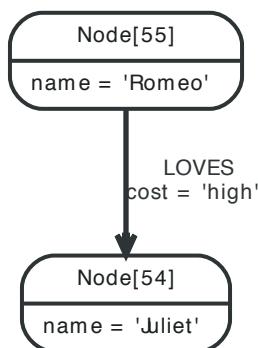
- DELETE <http://localhost:7474/db/data/relationship/40/properties/cost>
- Accept: application/json; charset=UTF-8

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 21.51. Starting Graph*Example request*

- DELETE <http://localhost:7474/db/data/relationship/38/properties/non-existent>
- Accept: application/json; charset=UTF-8

Example response

- 404: Not Found
- Content-Type: application/json; charset=UTF-8

```
{
  "message": "Relationship[38] 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:670)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteRelationshipProperty(RestfulGraphDatabase.java:812)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.translational.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)"
  ]
}
```

```

"java.lang.Thread.run(Thread.java:745)"
],
"errors": [
{
  "message": "Relationship[38] 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.

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:188)",
    "org.neo4j.server.rest.web.DatabaseActions.removeAllRelationshipProperties(DatabaseActions.java:660)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteAllRelationshipProperties(RestfulGraphDatabase.java:792)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.translational.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
    "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.impl.factory.GraphDatabaseFacade.getRelationshipById(GraphDatabaseFacade.java:248)",
        "org.neo4j.server.rest.web.DatabaseActions.relationship(DatabaseActions.java:184)",
        "org.neo4j.server.rest.web.DatabaseActions.removeAllRelationshipProperties(DatabaseActions.java:660)",
        "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteAllRelationshipProperties(RestfulGraphDatabase.java:792)",
        "java.lang.reflect.Method.invoke(Method.java:606)"
      ]
    }
  }
}

"org.neo4j.server.rest.translational.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
"org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
```

```

    "java.lang.Thread.run(Thread.java:745)"
]
},
"exception": "NotFoundException",
"fullname": "org.neo4j.graphdb.NotFoundException",
"stackTrace": [
    "org.neo4j.kernel.impl.factory.GraphDatabaseFacade.getRelationshipById(GraphDatabaseFacade.java:248)",
    "org.neo4j.server.rest.web.DatabaseActions.relationship(DatabaseActions.java:184)",
    "org.neo4j.server.rest.web.DatabaseActions.removeAllRelationshipProperties(DatabaseActions.java:660)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteAllRelationshipProperties(RestfulGraphDatabase.java:792)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
    "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.

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:188)",
        "org.neo4j.server.rest.web.DatabaseActions.removeRelationshipProperty(DatabaseActions.java:666)",
        "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteRelationshipProperty(RestfulGraphDatabase.java:812)",
        "java.lang.reflect.Method.invoke(Method.java:606)",
        "org.neo4j.server.rest.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
        "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
        "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.impl.factory.GraphDatabaseFacade.getRelationshipById(GraphDatabaseFacade.java:248)",
    "org.neo4j.server.rest.web.DatabaseActions.relationship(DatabaseActions.java:184)",
    "org.neo4j.server.rest.web.DatabaseActions.removeRelationshipProperty(DatabaseActions.java:666)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteRelationshipProperty(RestfulGraphDatabase.java:812)",
    "java.lang.reflect.Method.invoke(Method.java:606)",

"org.neo4j.server.rest.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
    "java.lang.Thread.run(Thread.java:745)"
]
},
"exception": "NotFoundException",
"fullname": "org.neo4j.graphdb.NotFoundException",
"stackTrace": [
    "org.neo4j.kernel.impl.factory.GraphDatabaseFacade.getRelationshipById(GraphDatabaseFacade.java:248)",
    "org.neo4j.server.rest.web.DatabaseActions.relationship(DatabaseActions.java:184)",
    "org.neo4j.server.rest.web.DatabaseActions.removeRelationshipProperty(DatabaseActions.java:666)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.deleteRelationshipProperty(RestfulGraphDatabase.java:812)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
    "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 21.52. Starting Graph

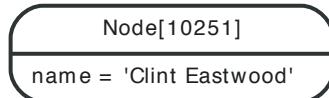
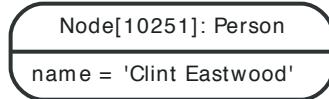


Figure 21.53. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/10251/labels`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
"Person"
```

Example response

- 204: No Content

Adding multiple labels to a node

Figure 21.54. Starting Graph

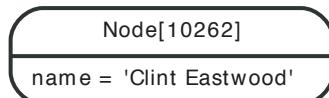
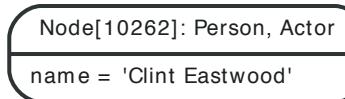


Figure 21.55. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/10262/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.

Example request

- POST http://localhost:7474/db/data/node/10269/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:319)",
    "org.neo4j.server.rest.web.RestfulGraphDatabase.addNodeLabel(RestfulGraphDatabase.java:446)",
    "java.lang.reflect.Method.invoke(Method.java:606)",
    "org.neo4j.server.rest.translational.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
    "org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
    "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.java:783)",
        "org.neo4j.kernel.impl.api.DataIntegrityValidatingStatementOperations.labelGetOrCreateForName(DataIntegrityValidatingStatementOperations.java:139)",
        "org.neo4j.kernel.impl.api.OperationsFacade.labelGetOrCreateForName(OperationsFacade.java:783)",
        "org.neo4j.kernel.impl.core.NodeProxy.addLabel(NodeProxy.java:620)",
        "org.neo4j.server.rest.web.DatabaseActions.addLabelToNode(DatabaseActions.java:314)",
        "org.neo4j.server.rest.web.RestfulGraphDatabase.addNodeLabel(RestfulGraphDatabase.java:446)",
        "java.lang.reflect.Method.invoke(Method.java:606)"
      ],
      "exception": "ConstraintViolationException",
      "fullname": "org.neo4j.graphdb.ConstraintViolationException",
      "stackTrace": [
        "org.neo4j.kernel.impl.core.NodeProxy.addLabel(NodeProxy.java:631)",
        "org.neo4j.kernel.impl.core.NodeProxy.addLabel(NodeProxy.java:631)"
      ]
    }
  }
}
```

```

"org.neo4j.server.rest.web.DatabaseActions.addLabelToNode(DatabaseActions.java:314)",
"org.neo4j.server.rest.web.RestfulGraphDatabase.addNodeLabel(RestfulGraphDatabase.java:446)",
"java.lang.reflect.Method.invoke(Method.java:606)",
"org.neo4j.server.rest.TransactionalRequestDispatcher.dispatch(TransactionalRequestDispatcher.java:139)",
"org.neo4j.server.rest.web.CollectUserAgentFilter.doFilter(CollectUserAgentFilter.java:69)",
"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 21.56. Starting Graph

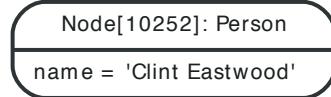
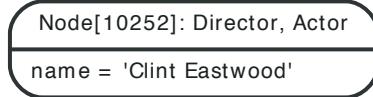


Figure 21.57. Final Graph



Example request

- PUT <http://localhost:7474/db/data/node/10252/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 21.58. Starting Graph

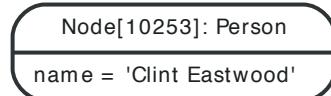
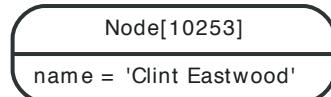


Figure 21.59. Final Graph



Example request

- DELETE `http://localhost:7474/db/data/node/10253/labels/Person`
- Accept: `application/json; charset=UTF-8`

Example response

- 204: No Content

Removing a non-existent label from a node

Figure 21.60. Starting Graph

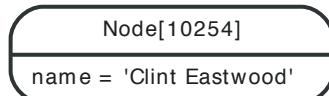
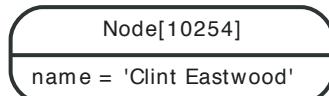


Figure 21.61. Final Graph



Example request

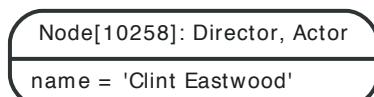
- DELETE `http://localhost:7474/db/data/node/10254/labels/Person`
- Accept: `application/json; charset=UTF-8`

Example response

- 204: No Content

Listing labels for a node

Figure 21.62. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/10258/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 21.63. 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`

```
[ {
  "labels" : "http://localhost:7474/db/data/node/10264/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/10264/relationships/out",
  "data" : {
    "name" : "Clint Eastwood"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/10264/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/10264/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/10264",
  "property" : "http://localhost:7474/db/data/node/10264/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/10264/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10264/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/10264/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/10264/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/10264/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/10264/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10264/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 10264,
    "labels" : [ "Director", "Actor" ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/10265/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/10265/relationships/out",
  "data" : {
    "name" : "Donald Sutherland"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/10265/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/10265/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/10265",
  "property" : "http://localhost:7474/db/data/node/10265/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/10265/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10265/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/10265/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/10265/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/10265/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/10265/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10265/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 10265,
    "labels" : [ "Actor" ]
  }
} ]
```

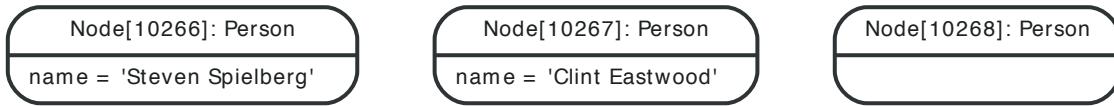
Get nodes by label and property

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 21.64. 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

```
[
  {
    "labels" : "http://localhost:7474/db/data/node/10267/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10267/relationships/out",
    "data" : {
      "name" : "Clint Eastwood"
    },
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10267/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/10267/traverse/{returnType}",
    "self" : "http://localhost:7474/db/data/node/10267",
    "property" : "http://localhost:7474/db/data/node/10267/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10267/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10267/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10267/relationships/in",
    "extensions" : { },
    "create_relationship" : "http://localhost:7474/db/data/node/10267/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/10267/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10267/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10267/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 10267,
      "labels" : [ "Person" ]
    }
  }
]
```

List all labels

By default, the server will return labels in use only. If you also want to return labels not in use, append the "in_use=0" query parameter.

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

```
[ "Actor", "Director", "Person" ]
```

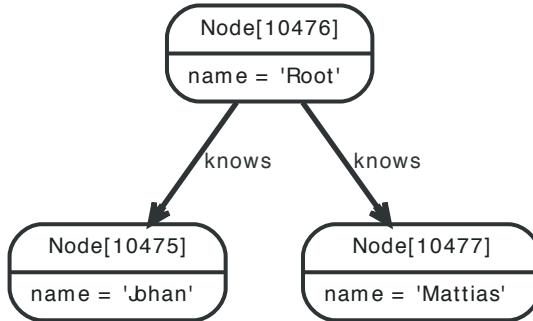
21.14. Node degree

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 21.65. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/10476/degree/all`
- Accept: `application/json; charset=UTF-8`

Example response

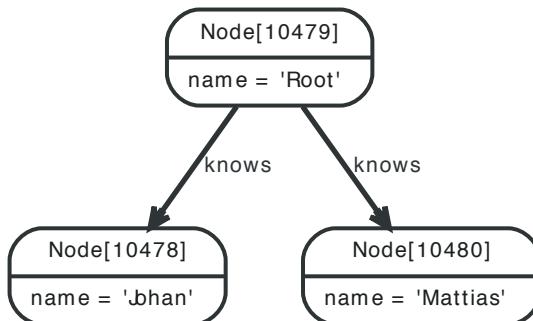
- 200: OK
- Content-Type: `application/json; charset=UTF-8`

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 21.66. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/10479/degree/out`
- Accept: `application/json; charset=UTF-8`

Example response

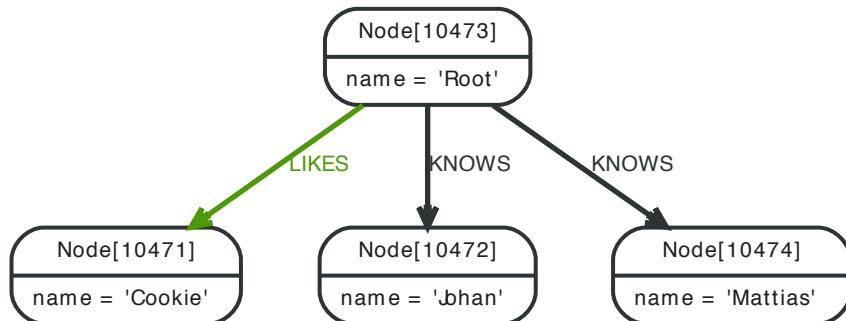
- 200: OK
- Content-Type: `application/json; charset=UTF-8`

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 21.67. Final Graph



Example request

- GET `http://localhost:7474/db/data/node/10473/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” \[417\]](#) 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.

Example request

- POST `http://localhost:7474/db/data/schema/index/label_1447161319402_1`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{  
  "property_keys" : [ "property_1447161319402_1" ]  
}
```

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
{  
  "label" : "label_1447161319402_1",  
  "property_keys" : [ "property_1447161319402_1" ]  
}
```

List indexes for a label

Example request

- GET `http://localhost:7474/db/data/schema/index/label_1447161319142_1`
- Accept: `application/json; charset=UTF-8`

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
[ {  
  "property_keys" : [ "property_1447161319142_1" ],  
  "label" : "label_1447161319142_1"  
} ]
```

Drop index

Drop index

Example request

- DELETE `http://localhost:7474/db/data/schema/index/label_1447161319302_1/property_1447161319302_1`
- 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/label_1447161295070_1/uniqueness/`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{  
  "property_keys" : [ "property_1447161295070_1" ]  
}
```

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
{  
  "label" : "label_1447161295070_1",  
  "type" : "UNIQUENESS",  
  "property_keys" : [ "property_1447161295070_1" ]  
}
```

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/label_1447161295429_1/uniqueness/property_1447161295429_1`
- Accept: `application/json; charset=UTF-8`

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
[ {  
  "property_keys" : [ "property_1447161295429_1" ],  
  "label" : "label_1447161295429_1",  
  "type" : "UNIQUENESS"  
} ]
```

Get all uniqueness constraints for a label

Example request

- GET `http://localhost:7474/db/data/schema/constraint/label_1447161295373_1/uniqueness/`
- Accept: `application/json; charset=UTF-8`

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
[ {
```

```
[{"property_keys": ["property_1447161295373_1"], "label": "label_1447161295373_1", "type": "UNIQUENESS"}, {"property_keys": ["property_1447161295373_2"], "label": "label_1447161295373_1", "type": "UNIQUENESS"}]
```

Drop uniqueness constraint

Drop uniqueness constraint for a label and a property.

Example request

- DELETE `http://localhost:7474/db/data/schema/constraint/label_1447161294978_1/uniqueness/property_1447161294978_1`
- Accept: `application/json; charset=UTF-8`

Example response

- 204: No Content

Get a specific node property existence constraint

Get a specific node property existence constraint for a label and a property.

Example request

- GET `http://localhost:7474/db/data/schema/constraint/label_1447163589965_1/existence/property_1447163589965_1`
- Accept: `application/json; charset=UTF-8`

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
[{"property_keys": ["property_1447163589965_1"], "label": "label_1447163589965_1", "type": "NODE_PROPERTY_EXISTENCE"}]
```

Get all node property existence constraints for a label

Example request

- GET `http://localhost:7474/db/data/schema/constraint/label_1447163590086_1/existence`
- Accept: `application/json; charset=UTF-8`

Example response

- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
[{"property_keys": ["property_1447163590086_1"], "label": "label_1447163590086_1", "type": "NODE_PROPERTY_EXISTENCE"}, {"property_keys": [...]}]
```

```
[{"property_keys": ["property_1447163590086_2"], "label": "label_1447163590086_1", "type": "NODE_PROPERTY_EXISTENCE"}]
```

Get all constraints for a label

Example request

- GET http://localhost:7474/db/data/schema/constraint/label_1447161295286_1
- Accept: application/json; charset=UTF-8

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

```
[{"property_keys": ["property_1447161295286_1"], "label": "label_1447161295286_1", "type": "UNIQUENESS"}]
```

Get a specific relationship property existence constraint

Get a specific relationship property existence constraint for a label and a property.

Example request

- GET http://localhost:7474/db/data/schema/relationship/constraint/relationshipType_1447163587779_1/existence/property_1447163587779_1
- Accept: application/json; charset=UTF-8

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

```
[{"relationshipType": "relationshipType_1447163587779_1", "property_keys": ["property_1447163587779_1"], "type": "RELATIONSHIP_PROPERTY_EXISTENCE"}]
```

Get all relationship property existence constraints for a type

Example request

- GET http://localhost:7474/db/data/schema/relationship/constraint/relationshipType_1447163589735_1/existence/
- Accept: application/json; charset=UTF-8

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

```
[{"relationshipType": "relationshipType_1447163589735_1", "property_keys": ["property_1447163589735_2"], "type": "RELATIONSHIP_PROPERTY_EXISTENCE"}]
```

```
}, {  
  "relationshipType" : "relationshipType_1447163589735_1",  
  "property_keys" : [ "property_1447163589735_1" ],  
  "type" : "RELATIONSHIP_PROPERTY_EXISTENCE"  
} ]
```

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" : [ "property_1447161295113_1" ],  
    "label" : "label_1447161295113_1",  
    "type" : "UNIQUENESS"  
  }, {  
    "property_keys" : [ "property_1447161295070_1" ],  
    "label" : "label_1447161295070_1",  
    "type" : "UNIQUENESS"  
  } ]
```

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” \[102\]](#) or write your own server side plugin executing the interesting traversals with the Java API (see [Section 32.1, “Server Plugins” \[563\]](#)) or secure your server, see [Chapter 27, Security \[502\]](#).

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” \[376\]](#) 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” \[376\]](#)), or use one of the built-in filters:

- `all`
- `all_but_start_node`

max_depth Is a short-hand way of specifying a prune evaluator which prunes after a certain depth. If not specified a max depth of 1 is used and if a `prune_evaluator` is specified instead of a `max_depth`, no max depth limit is set.

³ http://en.wikipedia.org/wiki/Breadth-first_search

⁴ http://en.wikipedia.org/wiki/Depth-first_search

⁵ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/Uniqueness.html>

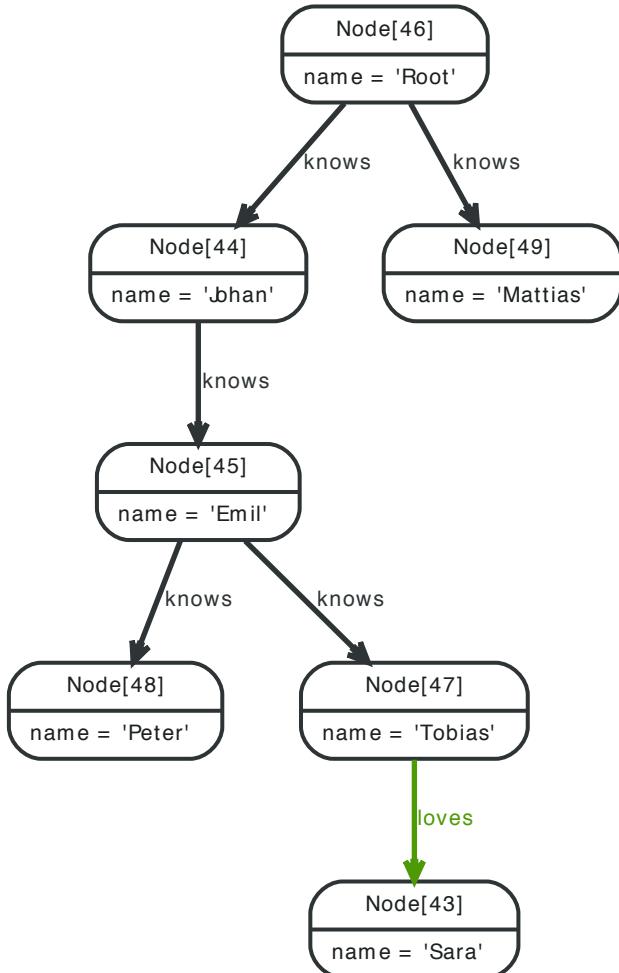
The position object in the body of the `return_filter` and `prune_evaluator` is a [Path⁶](#) object representing the path from the start node to the current traversal position.

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 API⁷](#). 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 21.68. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/46/traverse/node`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
{
  "order" : "breadth_first",
  "return_filter" : {
    "body" : "position.endNode().getProperty('name').toLowerCase().contains('t')",
    "language" : "javascript"
  },
  "prune_evaluator" : {
    "body" : "none"
  }
}
```

⁶ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/Path.html>

⁷ <http://neo4j.com/docs/2.3.1/javadocs/>

```

    "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

```

[ {
  "labels" : "http://localhost:7474/db/data/node/46/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/46/relationships/out",
  "data" : {
    "name" : "Root"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/46/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/46/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/46",
  "property" : "http://localhost:7474/db/data/node/46/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/46/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/46/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/46/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/46/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/46/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/46/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/46/relationships/in/-list|&|types",
  "metadata" : {
    "id" : 46,
    "labels" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/49/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/49/relationships/out",
  "data" : {
    "name" : "Mattias"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/49/relationships/all/-list|&|types",
  "traverse" : "http://localhost:7474/db/data/node/49/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/49",
  "property" : "http://localhost:7474/db/data/node/49/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/49/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/49/relationships/out/-list|&|types",
  "incoming_relationships" : "http://localhost:7474/db/data/node/49/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/49/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/49/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/49/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/49/relationships/in/-list|&|types",
  "metadata" : {
    "id" : 49,
    "labels" : [ ]
  }
}
]

```

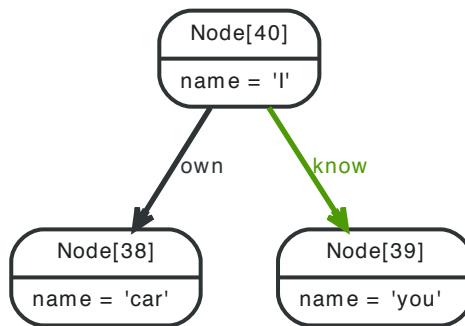
```

"labels" : "http://localhost:7474/db/data/node/48/labels",
"outgoing_relationships" : "http://localhost:7474/db/data/node/48/relationships/out",
"data" : {
  "name" : "Peter"
},
"all_typed_relationships" : "http://localhost:7474/db/data/node/48/relationships/all/{-list|&|types}",
"traverse" : "http://localhost:7474/db/data/node/48/traverse/{returnType}",
"self" : "http://localhost:7474/db/data/node/48",
"property" : "http://localhost:7474/db/data/node/48/properties/{key}",
"properties" : "http://localhost:7474/db/data/node/48/properties",
"outgoing_typed_relationships" : "http://localhost:7474/db/data/node/48/relationships/out/{-list|&|types}",
"incoming_relationships" : "http://localhost:7474/db/data/node/48/relationships/in",
"extensions" : { },
"create_relationship" : "http://localhost:7474/db/data/node/48/relationships",
"paged_traverse" : "http://localhost:7474/db/data/node/48/paged/traverse/{returnType}{?pageSize,leaseTime}",
"all_relationships" : "http://localhost:7474/db/data/node/48/relationships/all",
"incoming_typed_relationships" : "http://localhost:7474/db/data/node/48/relationships/in/{-list|&|types}",
"metadata" : {
  "id" : 48,
  "labels" : [ ]
}
},
{
"labels" : "http://localhost:7474/db/data/node/47/labels",
"outgoing_relationships" : "http://localhost:7474/db/data/node/47/relationships/out",
"data" : {
  "name" : "Tobias"
},
"all_typed_relationships" : "http://localhost:7474/db/data/node/47/relationships/all/{-list|&|types}",
"traverse" : "http://localhost:7474/db/data/node/47/traverse/{returnType}",
"self" : "http://localhost:7474/db/data/node/47",
"property" : "http://localhost:7474/db/data/node/47/properties/{key}",
"properties" : "http://localhost:7474/db/data/node/47/properties",
"outgoing_typed_relationships" : "http://localhost:7474/db/data/node/47/relationships/out/{-list|&|types}",
"incoming_relationships" : "http://localhost:7474/db/data/node/47/relationships/in",
"extensions" : { },
"create_relationship" : "http://localhost:7474/db/data/node/47/relationships",
"paged_traverse" : "http://localhost:7474/db/data/node/47/paged/traverse/{returnType}{?pageSize,leaseTime}",
"all_relationships" : "http://localhost:7474/db/data/node/47/relationships/all",
"incoming_typed_relationships" : "http://localhost:7474/db/data/node/47/relationships/in/{-list|&|types}",
"metadata" : {
  "id" : 47,
  "labels" : [ ]
}
}
]

```

Return relationships from a traversal

Figure 21.69. Final Graph



Example request

- POST <http://localhost:7474/db/data/node/40/traverse/relationship>

- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "order" : "breadth_first",
  "uniqueness" : "none",
  "return_filter" : {
    "language" : "builtin",
    "name" : "all"
  }
}
```

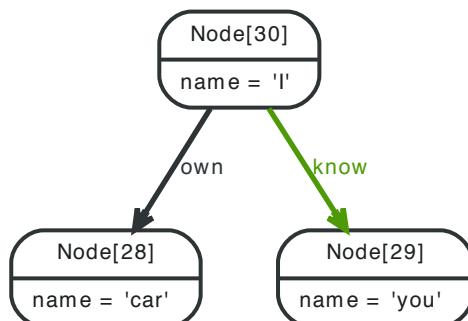
Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

```
[ {
  "start" : "http://localhost:7474/db/data/node/40",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/28",
  "property" : "http://localhost:7474/db/data/relationship/28/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/28/properties",
  "type" : "own",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/38",
  "metadata" : {
    "id" : 28,
    "type" : "own"
  }
}, {
  "start" : "http://localhost:7474/db/data/node/40",
  "data" : { },
  "self" : "http://localhost:7474/db/data/relationship/27",
  "property" : "http://localhost:7474/db/data/relationship/27/properties/{key}",
  "properties" : "http://localhost:7474/db/data/relationship/27/properties",
  "type" : "know",
  "extensions" : { },
  "end" : "http://localhost:7474/db/data/node/39",
  "metadata" : {
    "id" : 27,
    "type" : "know"
  }
} ]
```

Return paths from a traversal

Figure 21.70. Final Graph



Example request

- POST `http://localhost:7474/db/data/node/30/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

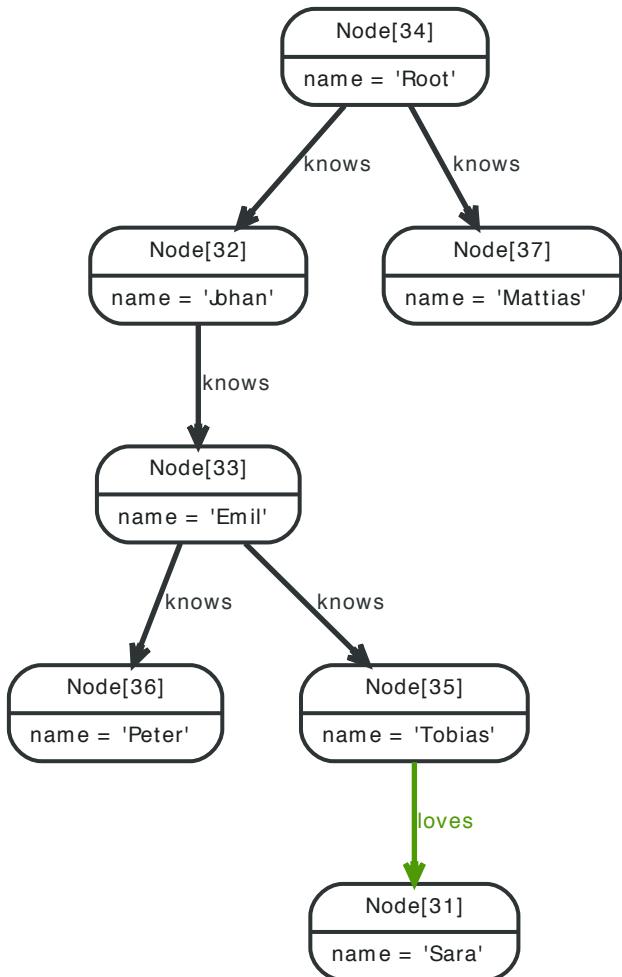
- 200: OK
- Content-Type: `application/json; charset=UTF-8`

```
[ {
  "directions" : [ ],
  "start" : "http://localhost:7474/db/data/node/30",
  "nodes" : [ "http://localhost:7474/db/data/node/30" ],
  "length" : 0,
  "relationships" : [ ],
  "end" : "http://localhost:7474/db/data/node/30"
}, {
  "directions" : [ "->" ],
  "start" : "http://localhost:7474/db/data/node/30",
  "nodes" : [ "http://localhost:7474/db/data/node/30", "http://localhost:7474/db/data/node/28" ],
  "length" : 1,
  "relationships" : [ "http://localhost:7474/db/data/relationship/20" ],
  "end" : "http://localhost:7474/db/data/node/28"
}, {
  "directions" : [ "->" ],
  "start" : "http://localhost:7474/db/data/node/30",
  "nodes" : [ "http://localhost:7474/db/data/node/30", "http://localhost:7474/db/data/node/29" ],
  "length" : 1,
  "relationships" : [ "http://localhost:7474/db/data/relationship/19" ],
  "end" : "http://localhost:7474/db/data/node/29"
} ]
```

Traversal returning nodes below a certain depth

Here, all nodes at a traversal depth below 3 are returned.

Figure 21.71. Final Graph

*Example request*

- POST <http://localhost:7474/db/data/node/34/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

```
[ {
  "labels" : "http://localhost:7474/db/data/node/34/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/34/relationships/out",
  "data" : {
    "name" : "Root"
  }
},
```

```

"all_typed_relationships" : "http://localhost:7474/db/data/node/34/relationships/all/{-list|&|types}",
"traverse" : "http://localhost:7474/db/data/node/34/traverse/{returnType}",
"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" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/32/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/32/relationships/out",
  "data" : {
    "name" : "Johan"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/32/relationships/all/-list|&|types",
  "traverse" : "http://localhost:7474/db/data/node/32/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/32",
  "property" : "http://localhost:7474/db/data/node/32/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/32/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/32/relationships/out/-list|&|types",
  "incoming_relationships" : "http://localhost:7474/db/data/node/32/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/32/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/32/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/32/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/32/relationships/in/-list|&|types",
  "metadata" : {
    "id" : 32,
    "labels" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/33/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/33/relationships/out",
  "data" : {
    "name" : "Emil"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/33/relationships/all/-list|&|types",
  "traverse" : "http://localhost:7474/db/data/node/33/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/33",
  "property" : "http://localhost:7474/db/data/node/33/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/33/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/33/relationships/out/-list|&|types",
  "incoming_relationships" : "http://localhost:7474/db/data/node/33/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/33/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/33/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/33/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/33/relationships/in/-list|&|types",
  "metadata" : {
    "id" : 33,
    "labels" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/37/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/37/relationships/out",
  "data" : {
    "name" : "Mattias"
  }
}

```

```
{
  "all_typed_relationships" : "http://localhost:7474/db/data/node/37/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/37/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/37",
  "property" : "http://localhost:7474/db/data/node/37/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/37/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/37/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/37/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/37/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/37/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/37/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/37/relationships/in/-list|&|types",
  "metadata" : {
    "id" : 37,
    "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/f6afad2bfad94478a86beab0f4d404a8>

```
[
  {
    "labels" : "http://localhost:7474/db/data/node/298/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/298/relationships/out",
    "data" : {
      "name" : "1"
    },
    "all_typed_relationships" : "http://localhost:7474/db/data/node/298/relationships/all/-list|&|types",
    "traverse" : "http://localhost:7474/db/data/node/298/traverse/{returnType}",
    "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" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/307/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/307/relationships/out",
  "data" : {
    "name" : "10"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/307/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/307/traverse/{returnType}",
  "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" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/308/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/308/relationships/out",
  "data" : {
    "name" : "11"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/308/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/308/traverse/{returnType}",
  "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" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/309/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/309/relationships/out",
  "data" : {
    "name" : "12"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/309/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/309/traverse/{returnType}",
  "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" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/310/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/310/relationships/out",
  "data" : {
    "name" : "13"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/310/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/310/traverse/{returnType}",
  "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" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/311/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/311/relationships/out",
  "data" : {
    "name" : "14"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/311/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/311/traverse/{returnType}",
  "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" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/312/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/312/relationships/out",
  "data" : {
    "name" : "15"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/312/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/312/traverse/{returnType}"
}

```

```

"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" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/313/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/313/relationships/out",
  "data" : {
    "name" : "16"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/313/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/313/traverse/{returnType}",
  "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" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/314/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/314/relationships/out",
  "data" : {
    "name" : "17"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/314/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/314/traverse/{returnType}",
  "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" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/315/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/315/relationships/out",
  "data" : {
    "name" : "18"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/315/relationships/all/{-list|&|types}"
}

```

```

"traverse" : "http://localhost:7474/db/data/node/315/traverse/{returnType}",
"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" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/316/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/316/relationships/out",
  "data" : {
    "name" : "19"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/316/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/316/traverse/{returnType}",
  "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" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/318/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/318/relationships/out",
  "data" : {
    "name" : "21"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/318/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/318/traverse/{returnType}",
  "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" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/328/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/328/relationships/out",
  "data" : {
    "name" : "31"
  }
},

```

```

"all_typed_relationships" : "http://localhost:7474/db/data/node/328/relationships/all/{-list|&|types}",
"traverse" : "http://localhost:7474/db/data/node/328/traverse/{returnType}",
"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/4bb1bdedc1604316b2132a8778fb7859`
- Accept: application/json

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

```

[ {
  "labels" : "http://localhost:7474/db/data/node/661/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/661/relationships/out",
  "data" : {
    "name" : "331"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/661/relationships/all/-list|&|types",
  "traverse" : "http://localhost:7474/db/data/node/661/traverse/{returnType}",
  "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" : [ ]
  }
}, {

```

```

"labels" : "http://localhost:7474/db/data/node/671/labels",
"outgoing_relationships" : "http://localhost:7474/db/data/node/671/relationships/out",
"data" : {
  "name" : "341"
},
"all_typed_relationships" : "http://localhost:7474/db/data/node/671/relationships/all/{-list|&|types}",
"traverse" : "http://localhost:7474/db/data/node/671/traverse/{returnType}",
"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" : [ ]
}
},
{
"labels" : "http://localhost:7474/db/data/node/681/labels",
"outgoing_relationships" : "http://localhost:7474/db/data/node/681/relationships/out",
"data" : {
  "name" : "351"
},
"all_typed_relationships" : "http://localhost:7474/db/data/node/681/relationships/all/-list|&|types",
"traverse" : "http://localhost:7474/db/data/node/681/traverse/{returnType}",
"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" : [ ]
}
},
{
"labels" : "http://localhost:7474/db/data/node/691/labels",
"outgoing_relationships" : "http://localhost:7474/db/data/node/691/relationships/out",
"data" : {
  "name" : "361"
},
"all_typed_relationships" : "http://localhost:7474/db/data/node/691/relationships/all/-list|&|types",
"traverse" : "http://localhost:7474/db/data/node/691/traverse/{returnType}",
"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" : [ ]
}
}

```

```

}, {
  "labels" : "http://localhost:7474/db/data/node/701/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/701/relationships/out",
  "data" : {
    "name" : "371"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/701/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/701/traverse/{returnType}",
  "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" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/711/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/711/relationships/out",
  "data" : {
    "name" : "381"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/711/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/711/traverse/{returnType}",
  "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" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/721/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/721/relationships/out",
  "data" : {
    "name" : "391"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/721/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/721/traverse/{returnType}",
  "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" : [ ]
  }
}

```

```

    },
    {
      "labels" : "http://localhost:7474/db/data/node/731/labels",
      "outgoing_relationships" : "http://localhost:7474/db/data/node/731/relationships/out",
      "data" : {
        "name" : "401"
      },
      "all_typed_relationships" : "http://localhost:7474/db/data/node/731/relationships/all/{-list|&|types}",
      "traverse" : "http://localhost:7474/db/data/node/731/traverse/{returnType}",
      "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" : [ ]
      }
    },
    {
      "labels" : "http://localhost:7474/db/data/node/740/labels",
      "outgoing_relationships" : "http://localhost:7474/db/data/node/740/relationships/out",
      "data" : {
        "name" : "410"
      },
      "all_typed_relationships" : "http://localhost:7474/db/data/node/740/relationships/all/{-list|&|types}",
      "traverse" : "http://localhost:7474/db/data/node/740/traverse/{returnType}",
      "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" : [ ]
      }
    },
    {
      "labels" : "http://localhost:7474/db/data/node/741/labels",
      "outgoing_relationships" : "http://localhost:7474/db/data/node/741/relationships/out",
      "data" : {
        "name" : "411"
      },
      "all_typed_relationships" : "http://localhost:7474/db/data/node/741/relationships/all/{-list|&|types}",
      "traverse" : "http://localhost:7474/db/data/node/741/traverse/{returnType}",
      "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" : [ ]
      }
    }
  ]
}

```

```

    "labels" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/742/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/742/relationships/out",
  "data" : {
    "name" : "412"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/742/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/742/traverse/{returnType}",
  "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" : [ ]
  }
},
{
  "labels" : "http://localhost:7474/db/data/node/743/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/743/relationships/out",
  "data" : {
    "name" : "413"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/743/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/743/traverse/{returnType}",
  "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" : [ ]
  }
},
{
  "labels" : "http://localhost:7474/db/data/node/744/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/744/relationships/out",
  "data" : {
    "name" : "414"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/744/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/744/traverse/{returnType}",
  "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" : [ ]
  },
}, {
  "labels" : "http://localhost:7474/db/data/node/745/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/745/relationships/out",
  "data" : {
    "name" : "415"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/745/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/745/traverse/{returnType}",
  "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" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/746/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/746/relationships/out",
  "data" : {
    "name" : "416"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/746/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/746/traverse/{returnType}",
  "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" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/747/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/747/relationships/out",
  "data" : {
    "name" : "417"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/747/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/747/traverse/{returnType}",
  "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" : [ ]
}
}, {
"labels" : "http://localhost:7474/db/data/node/748/labels",
"outgoing_relationships" : "http://localhost:7474/db/data/node/748/relationships/out",
"data" : {
    "name" : "418"
},
"All_typed_relationships" : "http://localhost:7474/db/data/node/748/relationships/all/{-list|&|types}",
"traverse" : "http://localhost:7474/db/data/node/748/traverse/{returnType}",
"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" : [ ]
}
}, {
"labels" : "http://localhost:7474/db/data/node/749/labels",
"outgoing_relationships" : "http://localhost:7474/db/data/node/749/relationships/out",
"data" : {
    "name" : "419"
},
"All_typed_relationships" : "http://localhost:7474/db/data/node/749/relationships/all/-list|&|types",
"traverse" : "http://localhost:7474/db/data/node/749/traverse/{returnType}",
"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" : [ ]
}
}, {
"labels" : "http://localhost:7474/db/data/node/751/labels",
"outgoing_relationships" : "http://localhost:7474/db/data/node/751/relationships/out",
"data" : {
    "name" : "421"
},
"All_typed_relationships" : "http://localhost:7474/db/data/node/751/relationships/all/-list|&|types",
"traverse" : "http://localhost:7474/db/data/node/751/traverse/{returnType}",
"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" : [ ]
}
},
{
  "labels" : "http://localhost:7474/db/data/node/761/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/761/relationships/out",
  "data" : {
    "name" : "431"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/761/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/761/traverse/{returnType}",
  "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" : [ ]
  }
},
{
  "labels" : "http://localhost:7474/db/data/node/771/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/771/relationships/out",
  "data" : {
    "name" : "441"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/771/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/771/traverse/{returnType}",
  "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/8bd345d095464be0bd60a738e946664e

```
[
{
    "labels" : "http://localhost:7474/db/data/node/34/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/34/relationships/out",
    "data" : {
        "name" : "1"
    },
    "all_typed_relationships" : "http://localhost:7474/db/data/node/34/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/34/traverse/{returnType}",
    "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');"
    }
}
```

```

    "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/b6bcc444f17a43509a42c7184a36a332>

```

[ {
    "labels" : "http://localhost:7474/db/data/node/808/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/808/relationships/out",
    "data" : {
        "name" : "1"
    },
    "all_typed_relationships" : "http://localhost:7474/db/data/node/808/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/808/traverse/{returnType}",
    "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" : [ ]
    }
}, {
    "labels" : "http://localhost:7474/db/data/node/817/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/817/relationships/out",
    "data" : {
        "name" : "10"
    },
    "all_typed_relationships" : "http://localhost:7474/db/data/node/817/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/817/traverse/{returnType}",
    "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" : [ ]
    }
}, {
    "labels" : "http://localhost:7474/db/data/node/818/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/818/relationships/out",
    "data" : {

```

```

    "name" : "11"
},
"all_typed_relationships" : "http://localhost:7474/db/data/node/818/relationships/all/{-list|&|types}",
"traverse" : "http://localhost:7474/db/data/node/818/traverse/{returnType}",
"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" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/819/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/819/relationships/out",
  "data" : {
    "name" : "12"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/819/relationships/all/-list|&|types",
  "traverse" : "http://localhost:7474/db/data/node/819/traverse/{returnType}",
  "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" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/820/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/820/relationships/out",
  "data" : {
    "name" : "13"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/820/relationships/all/-list|&|types",
  "traverse" : "http://localhost:7474/db/data/node/820/traverse/{returnType}",
  "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" : [ ]
  }
}, {
  "labels" : "http://localhost:7474/db/data/node/821/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/821/relationships/out",

```

```

"data" : {
    "name" : "14"
},
"all_typed_relationships" : "http://localhost:7474/db/data/node/821/relationships/all/{-list|&|types}",
"traverse" : "http://localhost:7474/db/data/node/821/traverse/{returnType}",
"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" : [ ]
}
}, {
"labels" : "http://localhost:7474/db/data/node/822/labels",
"outgoing_relationships" : "http://localhost:7474/db/data/node/822/relationships/out",
"data" : {
    "name" : "15"
},
"all_typed_relationships" : "http://localhost:7474/db/data/node/822/relationships/all/-list|&|types",
"traverse" : "http://localhost:7474/db/data/node/822/traverse/{returnType}",
"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" : [ ]
}
}, {
"labels" : "http://localhost:7474/db/data/node/823/labels",
"outgoing_relationships" : "http://localhost:7474/db/data/node/823/relationships/out",
"data" : {
    "name" : "16"
},
"all_typed_relationships" : "http://localhost:7474/db/data/node/823/relationships/all/-list|&|types",
"traverse" : "http://localhost:7474/db/data/node/823/traverse/{returnType}",
"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" : [ ]
}
}, {
"labels" : "http://localhost:7474/db/data/node/824/labels",

```

```

"outgoing_relationships" : "http://localhost:7474/db/data/node/824/relationships/out",
"data" : {
  "name" : "17"
},
"all_typed_relationships" : "http://localhost:7474/db/data/node/824/relationships/all/{-list|&|types}",
"traverse" : "http://localhost:7474/db/data/node/824/traverse/{returnType}",
"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" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/825/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/825/relationships/out",
  "data" : {
    "name" : "18"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/825/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/825/traverse/{returnType}",
  "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" : [ ]
}
}, {
  "labels" : "http://localhost:7474/db/data/node/826/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/826/relationships/out",
  "data" : {
    "name" : "19"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/826/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/826/traverse/{returnType}",
  "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" : [ ]
}
}

```

```
"labels" : "http://localhost:7474/db/data/node/828/labels",
"outgoing_relationships" : "http://localhost:7474/db/data/node/828/relationships/out",
"data" : {
  "name" : "21"
},
"all_typed_relationships" : "http://localhost:7474/db/data/node/828/relationships/all/{-list|&|types}",
"traverse" : "http://localhost:7474/db/data/node/828/traverse/{returnType}",
"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" : [ ]
}
},
{
"labels" : "http://localhost:7474/db/data/node/838/labels",
"outgoing_relationships" : "http://localhost:7474/db/data/node/838/relationships/out",
"data" : {
  "name" : "31"
},
"all_typed_relationships" : "http://localhost:7474/db/data/node/838/relationships/all/-list|&|types",
"traverse" : "http://localhost:7474/db/data/node/838/traverse/{returnType}",
"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

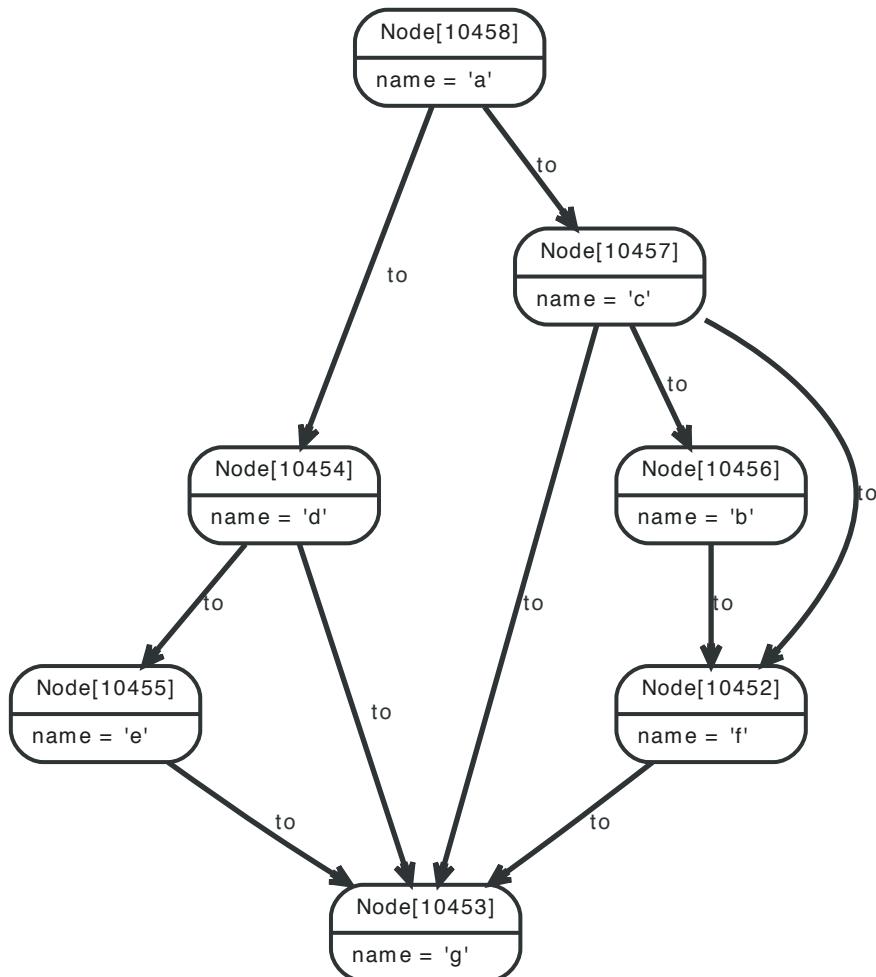
Neo4j comes with a number of built-in graph algorithms. They are performed from a start node. The traversal is controlled by the URI and the body sent with the request. These are the parameters that can be used:

algorithm	The algorithm to choose. If not set, default is shortestPath. algorithm can have one of these values:
	<ul style="list-style-type: none"> • 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 21.72. Final Graph



Example request

- POST <http://localhost:7474/db/data/node/10458/paths>
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{  
  "to" : "http://localhost:7474/db/data/node/10453",  
  "max_depth" : 3,  
  "relationships" : {  
    "type" : "to",  
    "direction" : "out"  
  },  
  "algorithm" : "shortestPath"  
}
```

Example response

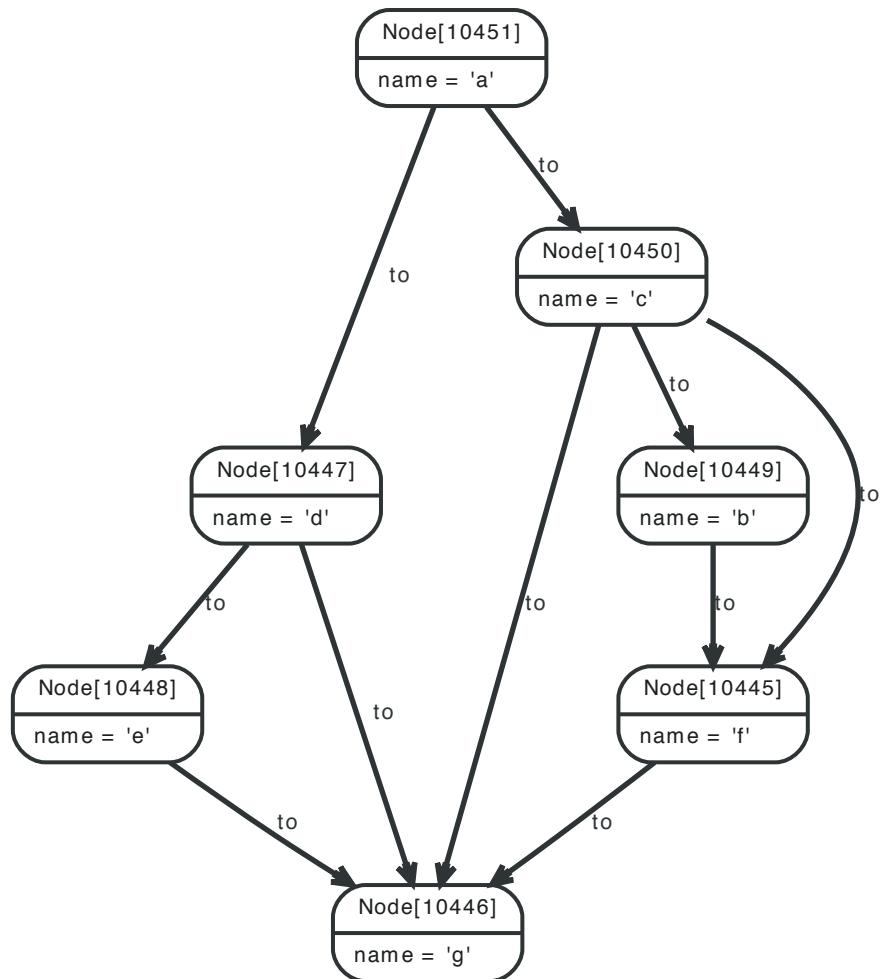
- 200: OK
- Content-Type: application/json; charset=UTF-8

```
[ {  
  "directions" : [ "->", "->" ],  
  "start" : "http://localhost:7474/db/data/node/10458",  
  "nodes" : [ "http://localhost:7474/db/data/node/10458", "http://localhost:7474/db/data/node/10457", "http://localhost:7474/db/data/node/10453" ],  
  "length" : 2,  
  "relationships" : [ "http://localhost:7474/db/data/relationship/228", "http://localhost:7474/db/data/relationship/237" ],  
  "end" : "http://localhost:7474/db/data/node/10453"  
}, {  
  "directions" : [ "->", "->" ],  
  "start" : "http://localhost:7474/db/data/node/10458",  
  "nodes" : [ "http://localhost:7474/db/data/node/10458", "http://localhost:7474/db/data/node/10454", "http://localhost:7474/db/data/node/10453" ],  
  "length" : 2,  
  "relationships" : [ "http://localhost:7474/db/data/relationship/229", "http://localhost:7474/db/data/relationship/235" ],  
  "end" : "http://localhost:7474/db/data/node/10453"  
} ]
```

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 21.73. Final Graph

*Example request*

- POST <http://localhost:7474/db/data/node/10451/path>
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "to" : "http://localhost:7474/db/data/node/10446",
  "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/10451",
  "nodes" : [ "http://localhost:7474/db/data/node/10451", "http://localhost:7474/db/data/node/10447", "http://localhost:7474/db/data/node/10446" ],
  "length" : 2
}
```

```

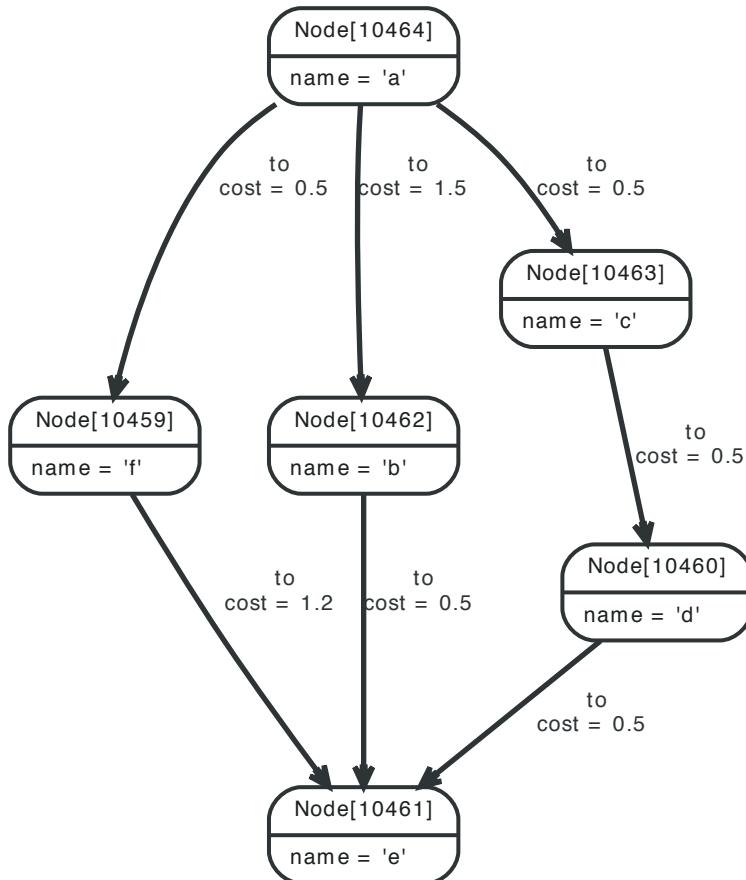
"relationships" : [ "http://localhost:7474/db/data/relationship/219", "http://localhost:7474/db/data/relationship/225" ],
"end" : "http://localhost:7474/db/data/node/10446"
}

```

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 21.74. Final Graph



Example request

- POST <http://localhost:7474/db/data/node/10464/path>
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "to" : "http://localhost:7474/db/data/node/10461",
  "cost_property" : "cost",
  "relationships" : {
    "type" : "to",
    "direction" : "out"
  },
  "algorithm" : "dijkstra"
}
```

Example response

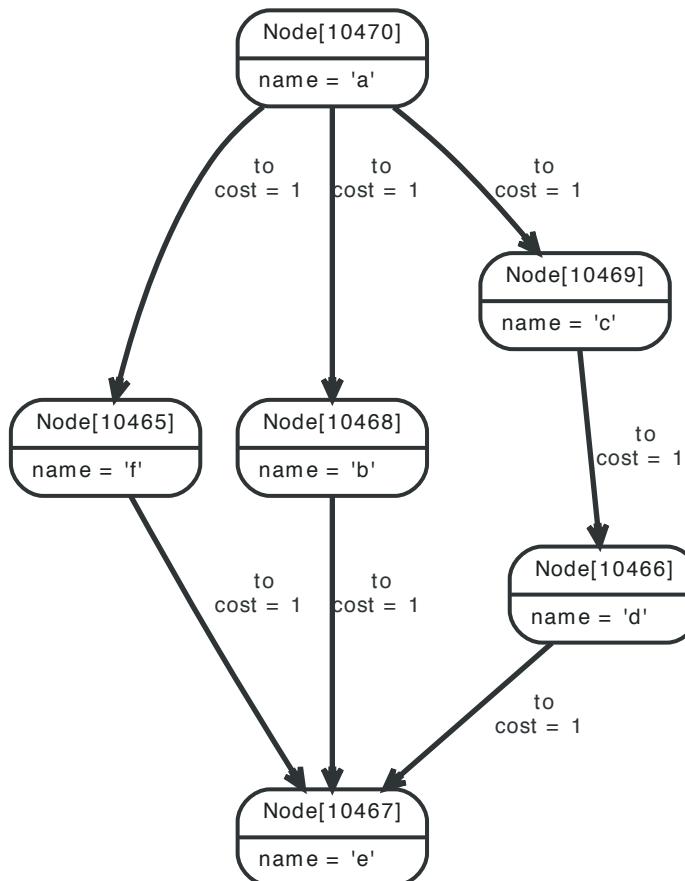
- 200: OK
- Content-Type: application/json; charset=UTF-8

```
{
  "directions" : [ "->", "->", "->" ],
  "weight" : 1.5,
  "start" : "http://localhost:7474/db/data/node/10464",
  "nodes" : [ "http://localhost:7474/db/data/node/10464", "http://localhost:7474/db/data/node/10463", "http://localhost:7474/db/data/node/10460", "http://localhost:7474/db/data/node/10461" ],
  "length" : 3,
  "relationships" : [ "http://localhost:7474/db/data/relationship/239", "http://localhost:7474/db/data/relationship/241",
  "http://localhost:7474/db/data/relationship/242" ],
  "end" : "http://localhost:7474/db/data/node/10461"
}
```

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 21.75. Final Graph



Example request

- POST <http://localhost:7474/db/data/node/10470/path>
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "to" : "http://localhost:7474/db/data/node/10467",
  "cost_property" : "cost",
  "relationships" : {
    "type" : "to",
    "direction" : "out"
  },
}
```

```
"algorithm" : "dijkstra"
}
```

Example response

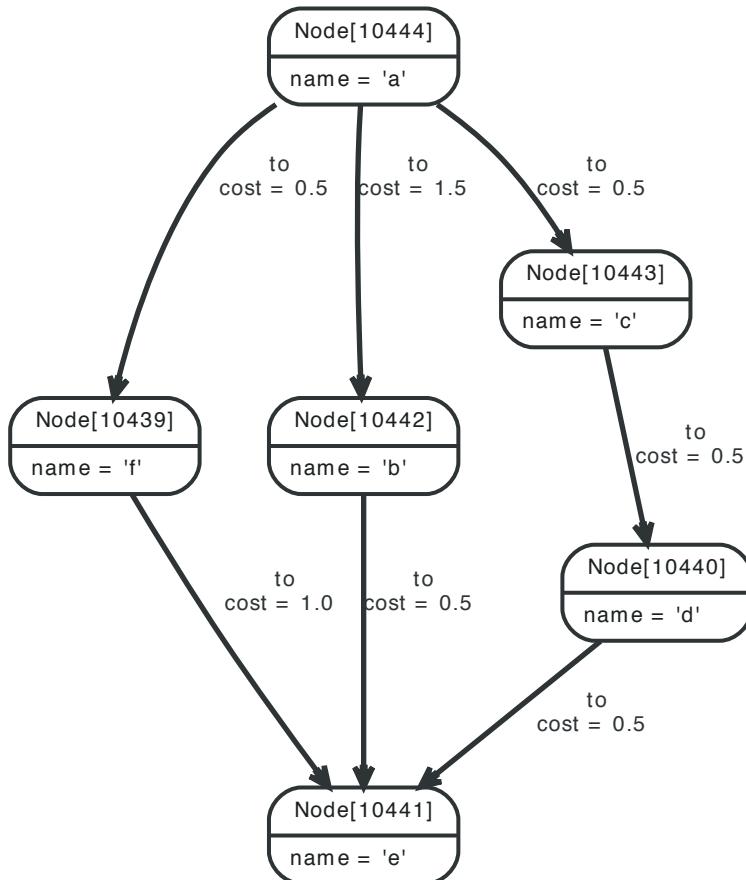
- 200: OK
- Content-Type: application/json; charset=UTF-8

```
{
  "directions" : [ "->", "->" ],
  "weight" : 2.0,
  "start" : "http://localhost:7474/db/data/node/10470",
  "nodes" : [ "http://localhost:7474/db/data/node/10470", "http://localhost:7474/db/data/node/10468", "http://localhost:7474/db/data/node/10467" ],
  "length" : 2,
  "relationships" : [ "http://localhost:7474/db/data/relationship/245", "http://localhost:7474/db/data/relationship/250" ],
  "end" : "http://localhost:7474/db/data/node/10467"
}
```

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 21.76. Final Graph



Example request

- POST <http://localhost:7474/db/data/node/10444/paths>
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{  
  "to" : "http://localhost:7474/db/data/node/10441",  
  "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/10444",  
  "nodes" : [ "http://localhost:7474/db/data/node/10444", "http://localhost:7474/db/data/node/10443", "http://localhost:7474/  
db/data/node/10440", "http://localhost:7474/db/data/node/10441" ],  
  "length" : 3,  
  "relationships" : [ "http://localhost:7474/db/data/relationship/212", "http://localhost:7474/db/data/relationship/214",  
  "http://localhost:7474/db/data/relationship/215" ],  
  "end" : "http://localhost:7474/db/data/node/10441"  
}, {  
  "directions" : [ "->", "->" ],  
  "weight" : 1.5,  
  "start" : "http://localhost:7474/db/data/node/10444",  
  "nodes" : [ "http://localhost:7474/db/data/node/10444", "http://localhost:7474/db/data/node/10439", "http://localhost:7474/  
db/data/node/10441" ],  
  "length" : 2,  
  "relationships" : [ "http://localhost:7474/db/data/relationship/213", "http://localhost:7474/db/data/relationship/217" ],  
  "end" : "http://localhost:7474/db/data/node/10441"  
} ]
```

21.19. Batch operations

Batch operations lets you execute multiple API calls through a single HTTP call. This improves performance for large insert and update operations significantly.

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 no changes will be applied.



Important

You cannot use this resource to execute Cypher queries with USING PERIODIC COMMIT.

Execute multiple operations in batch

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.

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:

```
[{"method": "PUT", "to": "/node/0/properties", "body": {"age": 1}, "id": 0},
 {"method": "GET", "to": "/node/0", "id": 1},
 {"method": "POST", "to": "/node", "body": {"age": 1}, "id": 2},
 {"method": "POST", "to": "/node", "body": {"age": 1}, "id": 3}]
```

Figure 21.77. Starting Graph

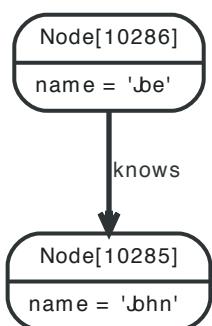
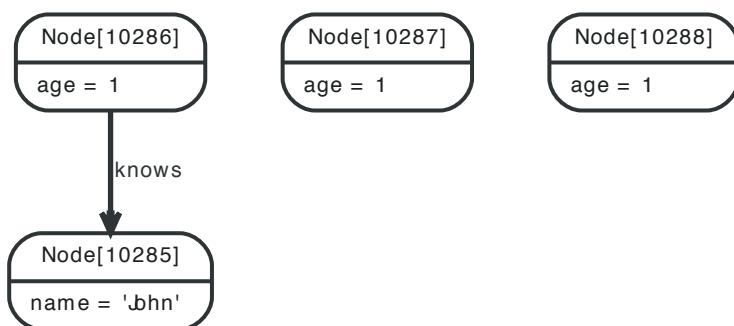


Figure 21.78. Final Graph



Example request

- POST `http://localhost:7474/db/data/batch`
- Accept: `application/json; charset=UTF-8`
- Content-Type: `application/json`

```
[ {
  "method" : "PUT",
  "to" : "/node/10286/properties",
  "body" : {
    "age" : 1
  },
  "id" : 0
}, {
  "method" : "GET",
  "to" : "/node/10286",
  "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/10286/properties"
}, {
  "id" : 1,
  "body" : {
    "extensions" : { },
    "labels" : "http://localhost:7474/db/data/node/10286/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10286/relationships/out",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10286/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/10286/traverse/{returnType}",
    "self" : "http://localhost:7474/db/data/node/10286",
    "property" : "http://localhost:7474/db/data/node/10286/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10286/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10286/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10286/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/10286/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/10286/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10286/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10286/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 10286,
      "labels" : [ ]
    },
    "data" : {
      "age" : 1
    }
} ]
```

```

},
  "from" : "/node/10286"
}, {
  "id" : 2,
  "location" : "http://localhost:7474/db/data/node/10287",
  "body" : {
    "extensions" : { },
    "labels" : "http://localhost:7474/db/data/node/10287/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10287/relationships/out",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10287/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/10287/traverse/{returnType}",
    "self" : "http://localhost:7474/db/data/node/10287",
    "property" : "http://localhost:7474/db/data/node/10287/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10287/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10287/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10287/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/10287/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/10287/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10287/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10287/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 10287,
      "labels" : [ ]
    },
    "data" : {
      "age" : 1
    }
  },
  "from" : "/node"
}, {
  "id" : 3,
  "location" : "http://localhost:7474/db/data/node/10288",
  "body" : {
    "extensions" : { },
    "labels" : "http://localhost:7474/db/data/node/10288/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10288/relationships/out",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10288/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/10288/traverse/{returnType}",
    "self" : "http://localhost:7474/db/data/node/10288",
    "property" : "http://localhost:7474/db/data/node/10288/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10288/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10288/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10288/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/10288/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/10288/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10288/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10288/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 10288,
      "labels" : [ ]
    },
    "data" : {
      "age" : 1
    }
  },
  "from" : "/node"
} ]

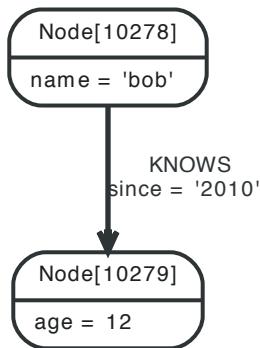
```

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 21.79. 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,
    "label" : "Person"
  }
]
```

```

"location" : "http://localhost:7474/db/data/node/10278",
"body" : {
  "extensions" : { },
  "labels" : "http://localhost:7474/db/data/node/10278/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/10278/relationships/out",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/10278/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/10278/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/10278",
  "property" : "http://localhost:7474/db/data/node/10278/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/10278/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10278/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/10278/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/10278/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/10278/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/10278/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10278/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 10278,
    "labels" : [ ]
  },
  "data" : {
    "name" : "bob"
  }
},
"from" : "/node"
}, {
  "id" : 1,
  "location" : "http://localhost:7474/db/data/node/10279",
  "body" : {
    "extensions" : { },
    "labels" : "http://localhost:7474/db/data/node/10279/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/10279/relationships/out",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/10279/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/10279/traverse/{returnType}",
    "self" : "http://localhost:7474/db/data/node/10279",
    "property" : "http://localhost:7474/db/data/node/10279/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/10279/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/10279/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/10279/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/10279/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/10279/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/10279/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/10279/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 10279,
      "labels" : [ ]
    },
    "data" : {
      "age" : 12
    }
},
"from" : "/node"
}, {
  "id" : 3,
  "location" : "http://localhost:7474/db/data/relationship/119",
  "body" : {
    "extensions" : { },
    "start" : "http://localhost:7474/db/data/node/10278",
    "property" : "http://localhost:7474/db/data/relationship/119/properties/{key}",
    "self" : "http://localhost:7474/db/data/relationship/119",
    "properties" : "http://localhost:7474/db/data/relationship/119/properties",
    "type" : "KNOWS",
    "end" : "http://localhost:7474/db/data/node/10279",
    "metadata" : {
      "id" : 119,
      "labels" : [ ]
    }
  }
}

```

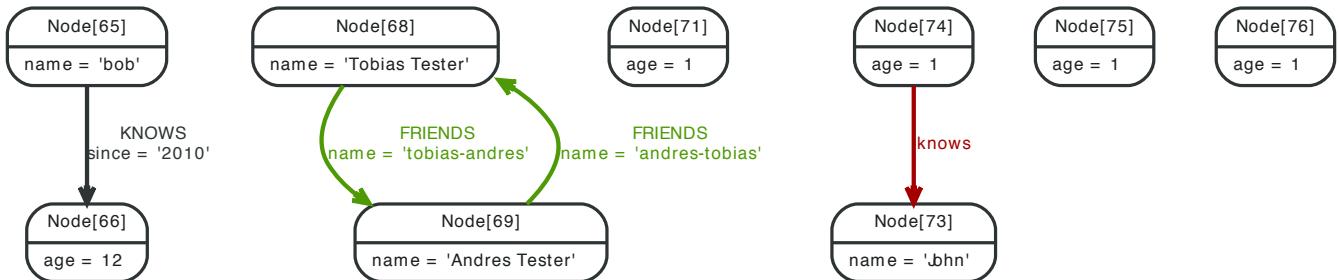
```

    "type" : "KNOWS"
},
"data" : {
  "since" : "2010"
}
},
"from" : "http://localhost:7474/db/data/node/10278/relationships"
}, {
  "id" : 4,
  "location" : "http://localhost:7474/db/data/index/relationship/my_rels/since/2010/119",
  "body" : {
    "extensions" : { },
    "start" : "http://localhost:7474/db/data/node/10278",
    "property" : "http://localhost:7474/db/data/relationship/119/properties/{key}",
    "self" : "http://localhost:7474/db/data/relationship/119",
    "properties" : "http://localhost:7474/db/data/relationship/119/properties",
    "type" : "KNOWS",
    "end" : "http://localhost:7474/db/data/node/10279",
    "metadata" : {
      "id" : 119,
      "type" : "KNOWS"
    },
    "data" : {
      "since" : "2010"
    },
    "indexed" : "http://localhost:7474/db/data/index/relationship/my_rels/since/2010/119"
},
"from" : "/index/relationship/my_rels"
} ]

```

Execute multiple operations in batch streaming

Figure 21.80. 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/74/properties",
    "body" : {
      "age" : 1
    },
    "id" : 0
  }, {
    "method" : "GET",
    "to" : "/node/74",
    "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/74/properties",
  "body" : null,
  "status" : 204
}, {
  "id" : 1,
  "from" : "/node/74",
  "body" : {
    "extensions" : { },
    "labels" : "http://localhost:7474/db/data/node/74/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/74/relationships/out",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/74/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/74/traverse/{returnType}",
    "self" : "http://localhost:7474/db/data/node/74",
    "property" : "http://localhost:7474/db/data/node/74/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/74/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/74/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/74/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/74/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/74/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/74/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/74/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 74,
      "labels" : [ ]
    },
    "data" : {
      "age" : 1
    }
  },
  "status" : 200
}, {
  "id" : 2,
  "from" : "/node",
  "body" : {
    "extensions" : { },
    "labels" : "http://localhost:7474/db/data/node/75/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/75/relationships/out",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/75/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/75/traverse/{returnType}",
    "self" : "http://localhost:7474/db/data/node/75",
    "property" : "http://localhost:7474/db/data/node/75/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/75/properties"
}

```

```
"outgoing_typed_relationships" : "http://localhost:7474/db/data/node/75/relationships/out/{-list|&|types}",
"incoming_relationships" : "http://localhost:7474/db/data/node/75/relationships/in",
"create_relationship" : "http://localhost:7474/db/data/node/75/relationships",
"paged_traverse" : "http://localhost:7474/db/data/node/75/paged/traverse/{returnType}{?pageSize,leaseTime}",
"all_relationships" : "http://localhost:7474/db/data/node/75/relationships/all",
"incoming_typed_relationships" : "http://localhost:7474/db/data/node/75/relationships/in/{-list|&|types}",
"metadata" : {
  "id" : 75,
  "labels" : [ ]
},
"data" : {
  "age" : 1
}
},
"location" : "http://localhost:7474/db/data/node/75",
"status" : 201
}, {
  "id" : 3,
  "from" : "/node",
  "body" : {
    "extensions" : { },
    "labels" : "http://localhost:7474/db/data/node/76/labels",
    "outgoing_relationships" : "http://localhost:7474/db/data/node/76/relationships/out",
    "all_typed_relationships" : "http://localhost:7474/db/data/node/76/relationships/all/{-list|&|types}",
    "traverse" : "http://localhost:7474/db/data/node/76/traverse/{returnType}",
    "self" : "http://localhost:7474/db/data/node/76",
    "property" : "http://localhost:7474/db/data/node/76/properties/{key}",
    "properties" : "http://localhost:7474/db/data/node/76/properties",
    "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/76/relationships/out/{-list|&|types}",
    "incoming_relationships" : "http://localhost:7474/db/data/node/76/relationships/in",
    "create_relationship" : "http://localhost:7474/db/data/node/76/relationships",
    "paged_traverse" : "http://localhost:7474/db/data/node/76/paged/traverse/{returnType}{?pageSize,leaseTime}",
    "all_relationships" : "http://localhost:7474/db/data/node/76/relationships/all",
    "incoming_typed_relationships" : "http://localhost:7474/db/data/node/76/relationships/in/{-list|&|types}",
    "metadata" : {
      "id" : 76,
      "labels" : [ ]
},
"data" : {
  "age" : 1
}
},
"location" : "http://localhost:7474/db/data/node/76",
"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” \[369\]](#).

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” \[417\]](#).

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" : "index_1447161296834_1"  
}
```

Example response

- 201: Created
- Content-Type: `application/json; charset=UTF-8`
- Location: `http://localhost:7474/db/data/index/node/index_1447161296834_1/`

```
{  
  "template" : "http://localhost:7474/db/data/index/node/index_1447161296834_1/{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 35.10, “Configuration and fulltext indexes” \[631\]](#) 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/index_1447161296073_1
- 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

```
{  
  "index_1447161295949_1" : {  
    "template" : "http://localhost:7474/db/data/index/node/index_1447161295949_1/{key}/{value}",  
    "provider" : "lucene",  
    "type" : "exact"  
  },  
  "index_1447161295774_1" : {  
    "template" : "http://localhost:7474/db/data/index/node/index_1447161295774_1/{key}/{value}",  
    "provider" : "lucene",  
    "type" : "exact"  
  },  
  "index_1447161295533_1" : {  
    "template" : "http://localhost:7474/db/data/index/node/index_1447161295533_1/{key}/{value}",  
    "provider" : "lucene",  
    "type" : "exact"  
  },  
}
```

```

"index_1447161295780_1" : {
  "template" : "http://localhost:7474/db/data/index/node/index_1447161295780_1/{key}/{value}",
  "provider" : "lucene",
  "type" : "exact"
},
"index_1447161296110_1" : {
  "template" : "http://localhost:7474/db/data/index/node/index_1447161296110_1/{key}/{value}",
  "provider" : "lucene",
  "type" : "exact"
},
"index_1447161295513_1" : {
  "template" : "http://localhost:7474/db/data/index/node/index_1447161295513_1/{key}/{value}",
  "provider" : "lucene",
  "type" : "exact"
},
"index_1447161295900_1" : {
  "template" : "http://localhost:7474/db/data/index/node/index_1447161295900_1/{key}/{value}",
  "provider" : "lucene",
  "type" : "exact"
},
"index_1447161295830_1" : {
  "template" : "http://localhost:7474/db/data/index/node/index_1447161295830_1/{key}/{value}",
  "provider" : "lucene",
  "type" : "exact"
},
"index_1447161296193_1" : {
  "template" : "http://localhost:7474/db/data/index/node/index_1447161296193_1/{key}/{value}",
  "provider" : "lucene",
  "type" : "exact"
},
"index_1447161295542_1" : {
  "template" : "http://localhost:7474/db/data/index/node/index_1447161295542_1/{key}/{value}",
  "provider" : "lucene",
  "type" : "exact"
},
"index_1447161295565_1" : {
  "template" : "http://localhost:7474/db/data/index/node/index_1447161295565_1/{key}/{value}",
  "provider" : "lucene",
  "type" : "exact"
},
"index_1447161296150_1" : {
  "template" : "http://localhost:7474/db/data/index/node/index_1447161296150_1/{key}/{value}",
  "provider" : "lucene",
  "type" : "exact"
},
"index_1447161296197_1" : {
  "template" : "http://localhost:7474/db/data/index/node/index_1447161296197_1/{key}/{value}",
  "provider" : "lucene",
  "type" : "exact"
},
"index_1447161295730_1" : {
  "template" : "http://localhost:7474/db/data/index/node/index_1447161295730_1/{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/index_1447161295780_1
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "value" : "some value",
  "uri" : "http://localhost:7474/db/data/node/11",
  "key" : "some-key"
}
```

Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8
- Location: http://localhost:7474/db/data/index/node/index_1447161295780_1/some-key/some%20value/11

```
{
  "extensions" : { },
  "labels" : "http://localhost:7474/db/data/node/11/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/11/relationships/out",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/11/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/11/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/11",
  "property" : "http://localhost:7474/db/data/node/11/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/11/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/11/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/11/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/11/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/11/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/11/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/11/relationships/in/-list|&|types",
  "metadata" : {
    "id" : 11,
    "labels" : [ ]
  },
  "data" : { },
  "indexed" : "http://localhost:7474/db/data/index/node/index_1447161295780_1/some-key/some%20value/11"
}
```

Remove all entries with a given node from an index

Example request

- DELETE http://localhost:7474/db/data/index/node/index_1447161296150_1/16
- 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/index_1447161296393_1/kvkey2/19`
- 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/index_1447161295830_1/kvkey1/value1/12`
- 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/index_1447161296775_1/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/index_1447161296775_1/key/the%2520value/27",  
  "labels" : "http://localhost:7474/db/data/node/27/labels",  
  "outgoing_relationships" : "http://localhost:7474/db/data/node/27/relationships/out",  
  "data" : { },  
  "all_typed_relationships" : "http://localhost:7474/db/data/node/27/relationships/all/{-list|&|types}",  
  "traverse" : "http://localhost:7474/db/data/node/27/traverse/{returnType}",  
  "self" : "http://localhost:7474/db/data/node/27",  
  "property" : "http://localhost:7474/db/data/node/27/properties/{key}",  
  "properties" : "http://localhost:7474/db/data/node/27/properties",  
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/27/relationships/out/{-list|&|types}",  
  "incoming_relationships" : "http://localhost:7474/db/data/node/27/relationships/in",  
  "extensions" : { },  
  "create_relationship" : "http://localhost:7474/db/data/node/27/relationships",  
  "paged_traverse" : "http://localhost:7474/db/data/node/27/paged/traverse/{returnType}{?pageSize,leaseTime}",  
  "all_relationships" : "http://localhost:7474/db/data/node/27/relationships/all",  
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/27/relationships/in/-list|&|types",  
  "metadata" : {  
    "id" : 27,  
    "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/index_1447161296716_1?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`

```
[ {
  "labels" : "http://localhost:7474/db/data/node/26/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/26/relationships/out",
  "data" : {
    "Name" : "Builder"
  },
  "all_typed_relationships" : "http://localhost:7474/db/data/node/26/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/26/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/26",
  "property" : "http://localhost:7474/db/data/node/26/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/26/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/26/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/26/relationships/in",
  "extensions" : { },
  "create_relationship" : "http://localhost:7474/db/data/node/26/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/26/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/26/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/26/relationships/in/-list|&|types",
  "metadata" : {
    "id" : 26,
    "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" \[295\]](#).

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/index_1447161296653_1?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/index_1447161296653_1/name/Tobias/25`

```
{
  "extensions" : { },
  "labels" : "http://localhost:7474/db/data/node/25/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/25/relationships/out",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/25/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/25/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/25",
  "property" : "http://localhost:7474/db/data/node/25/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/25/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/25/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/25/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/25/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/25/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/25/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/25/relationships/in/-list|&|types",
  "metadata" : {
    "id" : 25,
    "labels" : [ ]
  }
}
```

```
{
  "data" : {
    "sequence" : 1,
    "name" : "Tobias"
  },
  "indexed" : "http://localhost:7474/db/data/index/node/index_1447161296653_1/name/Tobias/25"
}
```

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/index_1447161296110_1?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/index_1447161296110_1/name/Peter/15

```
{
  "extensions" : { },
  "labels" : "http://localhost:7474/db/data/node/15/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/15/relationships/out",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/15/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/15/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/15",
  "property" : "http://localhost:7474/db/data/node/15/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/15/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/15/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/15/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/15/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/15/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/15/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/15/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 15,
    "labels" : [ ]
  },
  "data" : {
    "sequence" : 1,
    "name" : "Peter"
  },
  "indexed" : "http://localhost:7474/db/data/index/node/index_1447161296110_1/name/Peter/15"
}
```

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/index_1447161296596_1?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/index_1447161296596_1/name/Tobias/24

```
{
  "extensions" : { },
  "labels" : "http://localhost:7474/db/data/node/24/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/24/relationships/out",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/24/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/24/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/24",
  "property" : "http://localhost:7474/db/data/node/24/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/24/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/24/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/24/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/24/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/24/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/24/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/24/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 24,
    "labels" : [ ]
  },
  "data" : {
    "sequence" : 1,
    "name" : "Tobias"
  },
  "indexed" : "http://localhost:7474/db/data/index/node/index_1447161296596_1/name/Tobias/24"
}
```

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/index_1447161295730_1?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" : { },
  "labels" : "http://localhost:7474/db/data/node/9/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/9/relationships/out",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/9/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/9/traverse/{returnType}",
  "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",
  "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" : [ ]
  },
  "data" : {
    "sequence" : 1,
    "name" : "Peter"
  },
  "indexed" : "http://localhost:7474/db/data/index/node/index_1447161295730_1/name/Peter/9"
}
```

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/index_1447161296444_1?uniqueness=create_or_fail`
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "value" : "some value",
  "uri" : "http://localhost:7474/db/data/node/20",
  "key" : "some-key"
}
```

Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8
- Location: `http://localhost:7474/db/data/index/node/index_1447161296444_1/some-key/some%20value/20`

```
{
  "extensions" : { },
  "labels" : "http://localhost:7474/db/data/node/20/labels",
```

```

"outgoing_relationships" : "http://localhost:7474/db/data/node/20/relationships/out",
"all_typed_relationships" : "http://localhost:7474/db/data/node/20/relationships/all/{-list|&|types}",
"traverse" : "http://localhost:7474/db/data/node/20/traverse/{returnType}",
"self" : "http://localhost:7474/db/data/node/20",
"property" : "http://localhost:7474/db/data/node/20/properties/{key}",
"properties" : "http://localhost:7474/db/data/node/20/properties",
"outgoing_typed_relationships" : "http://localhost:7474/db/data/node/20/relationships/out/{-list|&|types}",
"incoming_relationships" : "http://localhost:7474/db/data/node/20/relationships/in",
"create_relationship" : "http://localhost:7474/db/data/node/20/relationships",
"paged_traverse" : "http://localhost:7474/db/data/node/20/paged/traverse/{returnType}{?pageSize,leaseTime}",
"all_relationships" : "http://localhost:7474/db/data/node/20/relationships/all",
"incoming_typed_relationships" : "http://localhost:7474/db/data/node/20/relationships/in/{-list|&|types}",
"metadata" : {
  "id" : 20,
  "labels" : [ ]
},
"data" : { },
"indexed" : "http://localhost:7474/db/data/index/node/index_1447161296444_1/some-key/some%20value/20"
}

```

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/index_1447161296529_1?uniqueness=create_or_fail`
- Accept: application/json; charset=UTF-8
- Content-Type: application/json

```
{
  "value" : "some value",
  "uri" : "http://localhost:7474/db/data/node/23",
  "key" : "some-key"
}
```

Example response

- 409: Conflict
- Content-Type: application/json; charset=UTF-8

```
{
  "extensions" : { },
  "labels" : "http://localhost:7474/db/data/node/22/labels",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/22/relationships/out",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/22/relationships/all/{-list|&|types}",
  "traverse" : "http://localhost:7474/db/data/node/22/traverse/{returnType}",
  "self" : "http://localhost:7474/db/data/node/22",
  "property" : "http://localhost:7474/db/data/node/22/properties/{key}",
  "properties" : "http://localhost:7474/db/data/node/22/properties",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/22/relationships/out/{-list|&|types}",
  "incoming_relationships" : "http://localhost:7474/db/data/node/22/relationships/in",
  "create_relationship" : "http://localhost:7474/db/data/node/22/relationships",
  "paged_traverse" : "http://localhost:7474/db/data/node/22/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/22/relationships/all",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/22/relationships/in/{-list|&|types}",
  "metadata" : {
    "id" : 22,
    "labels" : [ ]
  },
  "data" : {
    "some-key" : "some value"
  }
}
```

```
{
  "indexed" : "http://localhost:7474/db/data/index/node/index_1447161296529_1/some-key/some%20value/22"
}
```

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/index_1447161288652_1/?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/134",
  "end" : "http://localhost:7474/db/data/node/135",
  "type" : "knowledge"
}
```

Example response

- 201: Created
- Content-Type: `application/json; charset=UTF-8`
- Location: `http://localhost:7474/db/data/index/relationship/index_1447161288652_1/name/Tobias/89`

```
{
  "extensions" : { },
  "start" : "http://localhost:7474/db/data/node/134",
  "property" : "http://localhost:7474/db/data/relationship/89/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/89",
  "properties" : "http://localhost:7474/db/data/relationship/89/properties",
  "type" : "knowledge",
  "end" : "http://localhost:7474/db/data/node/135",
  "metadata" : {
    "id" : 89,
    "type" : "knowledge"
  },
  "data" : {
    "name" : "Tobias"
  },
  "indexed" : "http://localhost:7474/db/data/index/relationship/index_1447161288652_1/name/Tobias/89"
}
```

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/index_1447161288747_1?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/138",
  "end" : "http://localhost:7474/db/data/node/139",
  "type" : "KNOWS"
}
```

Example response

- 200: OK
- Content-Type: application/json; charset=UTF-8

```
{
  "extensions" : { },
  "start" : "http://localhost:7474/db/data/node/136",
  "property" : "http://localhost:7474/db/data/relationship/90/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/90",
  "properties" : "http://localhost:7474/db/data/relationship/90/properties",
  "type" : "KNOWS",
  "end" : "http://localhost:7474/db/data/node/137",
  "metadata" : {
    "id" : 90,
    "type" : "KNOWS"
  },
  "data" : { },
  "indexed" : "http://localhost:7474/db/data/index/relationship/index_1447161288747_1/name/Peter/90"
}
```

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/index_1447161288924_1?
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/146",
  "end" : "http://localhost:7474/db/data/node/147",
  "type" : "KNOWS"
}
```

Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8
- Location: http://localhost:7474/db/data/index/relationship/index_1447161288924_1/name/Tobias/93

```
{
  "extensions" : { },
```

```

"start" : "http://localhost:7474/db/data/node/146",
"property" : "http://localhost:7474/db/data/relationship/93/properties/{key}",
"self" : "http://localhost:7474/db/data/relationship/93",
"properties" : "http://localhost:7474/db/data/relationship/93/properties",
"type" : "KNOWS",
"end" : "http://localhost:7474/db/data/node/147",
"metadata" : {
  "id" : 93,
  "type" : "KNOWS"
},
"data" : {
  "name" : "Tobias"
},
"indexed" : "http://localhost:7474/db/data/index/relationship/index_1447161288924_1/name/Tobias/93"
}

```

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/index_1447161288363_1?`
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/126",
  "end" : "http://localhost:7474/db/data/node/127",
  "type" : "KNOWS"
}
```

Example response

- 409: Conflict
- Content-Type: application/json; charset=UTF-8

```
{
  "extensions" : { },
  "start" : "http://localhost:7474/db/data/node/124",
  "property" : "http://localhost:7474/db/data/relationship/85/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/85",
  "properties" : "http://localhost:7474/db/data/relationship/85/properties",
  "type" : "KNOWS",
  "end" : "http://localhost:7474/db/data/node/125",
  "metadata" : {
    "id" : 85,
    "type" : "KNOWS"
  },
  "data" : { },
  "indexed" : "http://localhost:7474/db/data/index/relationship/index_1447161288363_1/name/Peter/85"
}
```

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/index_1447161288308_1?`
`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/84"
}
```

Example response

- 201: Created
- Content-Type: application/json; charset=UTF-8
- Location: `http://localhost:7474/db/data/index/relationship/index_1447161288308_1/name/Peter/84`

```
{
  "extensions" : { },
  "start" : "http://localhost:7474/db/data/node/122",
  "property" : "http://localhost:7474/db/data/relationship/84/properties/{key}",
  "self" : "http://localhost:7474/db/data/relationship/84",
  "properties" : "http://localhost:7474/db/data/relationship/84/properties",
  "type" : "KNOWS",
  "end" : "http://localhost:7474/db/data/node/123",
  "metadata" : {
    "id" : 84,
    "type" : "KNOWS"
  },
  "data" : { },
  "indexed" : "http://localhost:7474/db/data/index/relationship/index_1447161288308_1/name/Peter/84"
}
```

Add an existing relationship to a unique index (already indexed)*Example request*

- POST `http://localhost:7474/db/data/index/relationship/index_1447161288460_1?`
`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/87"
}
```

Example response

- 409: Conflict
- Content-Type: application/json; charset=UTF-8

```
{
```

```
"extensions" : { },
"start" : "http://localhost:7474/db/data/node/128",
"property" : "http://localhost:7474/db/data/relationship/86/properties/{key}",
"self" : "http://localhost:7474/db/data/relationship/86",
"properties" : "http://localhost:7474/db/data/relationship/86/properties",
"type" : "KNOWS",
"end" : "http://localhost:7474/db/data/node/129",
"metadata" : {
  "id" : 86,
  "type" : "KNOWS"
},
"data" : { },
"indexed" : "http://localhost:7474/db/data/index/relationship/index_1447161288460_1/name/Peter/86"
}
```

21.22. 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.

21.23. 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 2.3.1 or earlier in order to help you find a smoother transition path to future releases. All features listed below may be removed in a future major release.

Cypher ExecutionEngine	There's no need to use ExecutionEngine anymore. 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.
Windows scripts	The .bat files used to operate the database and tools on Windows are being phased out and will be removed in future releases, in favor of modern, equivalent PowerShell scripts. For more information, see Section 23.3, “Windows PowerShell module” [443] .
STR() function	The STR() function is deprecated from Neo4j version 2.3 and onwards.
HAS() function	The HAS() function is deprecated from Neo4j version 2.3 and onwards. Please use EXISTS() instead.
Bare node patterns	The usage of node identifiers without enclosing them in parentheses, such as in MATCH n-->m RETURN n.prop, is deprecated from Neo4j version 2.3 and onwards. Please use MATCH (n)-->(m) RETURN n.prop instead.
Create nodes from a parameter list of maps	The syntax CREATE (:X {param}), where param is a list of maps, is deprecated in Neo4j version 2.3 and onwards. To achieve the same functionality, use UNWIND instead: UNWIND {param} as props CREATE (n:X) SET n = props

¹ [http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/GraphDatabaseService.html#execute\(java.lang.String\)](http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/GraphDatabaseService.html#execute(java.lang.String))

² <http://neo4j.com/docs/2.3.1/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.

23. Installation & Deployment	438
23.1. System Requirements	439
23.2. Server Installation	440
23.3. Windows PowerShell module	443
23.4. Upgrading	446
23.5. Setup for remote debugging	448
23.6. Usage Data Collector	449
24. Configuration & Performance	450
24.1. Introduction	451
24.2. Server Configuration	452
24.3. Server Performance Tuning	456
24.4. Performance Guide	457
24.5. Logical logs	461
24.6. Compressed storage of property values	462
24.7. Memory mapped IO settings	464
24.8. Configuration Settings Reference	466
25. High Availability	474
25.1. Architecture	475
25.2. HA Setup and configuration	476
25.3. How Neo4j HA operates	482
25.4. Arbiter Instances	483
25.5. Upgrade of a Neo4j HA Cluster	484
25.6. High Availability setup tutorial	486
25.7. REST endpoint for HA status information	492
25.8. Setting up HAProxy as a load balancer	494
26. Backup	497
26.1. Introducing Backup	498
26.2. Performing Backups	500
26.3. Restoring Your Data	501
27. Security	502
27.1. Securing Neo4j Server	503
28. Monitoring	508
28.1. Adjusting remote JMX access to the Neo4j Server	509
28.2. How to connect to a Neo4j instance using JMX and JConsole	510
28.3. How to connect to the JMX monitoring programmatically	513
28.4. Reference of supported JMX MBeans	514
28.5. Metrics Reporting	524

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” \[440\]](#).

For running Neo4j in high availability mode, see [Chapter 25, *High Availability* \[474\]](#).

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 8 (preferred) or 7¹ or Oracle Java 8 (preferred) or 7²
Operating Systems	Linux, HP UX, Windows 2012 for production; additionally Windows XP, Mac OS X for development.
Architectures	x86, Power8 (OpenJDK 8 only)



Important

You must use an up-to-date Java release as there are bugs in earlier releases which may affect Neo4j.

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" \[440\]](#).
- For Unix/Linux, see [the section called "Linux" \[441\]](#).
- For OSX, see [the section called "Mac OSX" \[441\]](#).

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" \[441\]](#) for more information on this.

For information on High Availability, please refer to [Chapter 25, High Availability \[474\]](#).

Prerequisites

With the exception of our Windows and Mac Installers, you'll need a Java Virtual Machine installed on your computer. We recommend that you install [OpenJDK 8 \(preferred\) or 7³](#) or [Oracle Java 8 \(preferred\) or 7⁴](#).

Setting Proper File Permissions

When installing Neo4j Server, keep in mind that the *bin/neo4j* executable will need to be run by some OS system user, and that user will need write permissions to some files/directories. This goes specifically for the *data/graph.db* directory. That user will also need execute permissions on other files, such as those in *bin/*.

It is recommended to either choose or create a user who will own and manage the Neo4j Server. This user should own the entire Neo4j directory, so make sure to untar/unzip it as this user and not with sudo (UNIX/Linux/OSx) etc.

If *data/graph.db* is not writable by the user Neo4j won't be able to write anything either to the store or its log files. As a result any logs would be appended to *console.log*. The following error message would indicate a possible permissions issue: Write transactions to database disabled.

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 instance of Neo4j the installer will select a new install directory by default. If you specify the same directory it will ask if you want to upgrade. This should proceed without issue although

³ <http://openjdk.java.net/>

⁴ <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

some users have reported a JRE is damaged error. 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: NE04J_HOME
3. Consult [Section 23.3, "Windows PowerShell module" \[443\]](#) for how to start or install Neo4j.



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 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 24.4, "Performance Guide" \[457\]](#).

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: NE04J_HOME
3. Change directory to: \$NE04J_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

Mac OSX Installer

1. Download the .dmg installer that you want from <http://neo4j.com/download/>.

⁵ http://refspecs.linuxfoundation.org/LSB_3.1.0/LSB-Core-generic/LSB-Core-generic/tocsyinit.html

⁶ <https://gist.github.com/chrisvest/7673244>

⁷ <http://www.linux.com/learn/tutorials/442412-managing-linux-daemons-with-init-scripts>

2. Click the downloaded installer file.
3. Drag the Neo4j icon into the Applications folder.

**Tip**

If you install Neo4j using the Mac installer and already have an existing instance of Neo4j the installer will ensure that both the old and new versions can co-exist on your system.

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 its 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.

Java SE 6 is incompatible with Neo4j 2.3.1, so we strongly advise you to skip installing Java SE 6 if you have no other uses for it. Instead, for Neo4j 2.3.1 we recommend you install Java SE 8 (preferred) or 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” \[489\]](#). 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. Windows PowerShell module

The Neo4j PowerShell module allows administrators to:

- audit and set Neo4j configuration settings,
- install, start and stop Neo4j Windows® Services
- and start tools, such as Neo4j Shell and Neo4j Import.

The PowerShell module is installed as part of the [ZIP file](#)⁸ distributions of Neo4j.

System Requirements

- Requires PowerShell v2.0 or above.
- Supported on either 32 or 64 bit operating systems.

How do I import the module?

The module file is located in the `bin` directory of your Neo4j installation, i.e. where you unzipped the downloaded file. For example, if Neo4j was installed in `C:\Neo4j` then the module would be imported like this:

```
Import-Module C:\Neo4j\bin\Neo4j-Management.psd1
```

This will add the module to the current session.



Note

On Windows it is sometimes necessary to *Unblock* a downloaded zip file before you can import its contents as a module. If you right-click on the zip file and choose "Properties" you will get a dialog. Bottom-right on that dialog you will find an "Unblock" button. Click that. Then you should be able to import the module.



Note

Running scripts has to be enabled on the system. This can for example be achieved by executing the following from an elevated PowerShell prompt:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

For more information see [About execution policies](#)⁹.

Once the module has been imported you can start an interactive console version of a Neo4j Server like this:

```
'C:\Neo4j' | Start-Neo4jServer -Console -Wait
```

To stop the server, issue `ctrl-c` in the console window that was created by the command.

How do I get help about the module?

Once the module is imported you can query the available commands like this:

```
Get-Command -Module Neo4j-Management
```

The output should be similar to the following:

CommandType	Name	Version	Source
Function	Get-Neo4jHome	2.3.0	Neo4j-Management
Function	Get-Neo4jServer	2.3.0	Neo4j-Management
Function	Get-Neo4jSetting	2.3.0	Neo4j-Management

⁸ <http://neo4j.com/download/>

⁹ <http://go.microsoft.com/fwlink/?LinkID=135>

Function	Initialize-Neo4jHACluster	2.3.0	Neo4j-Management
Function	Initialize-Neo4jServer	2.3.0	Neo4j-Management
Function	Install-Neo4jArbiter	2.3.0	Neo4j-Management
Function	Install-Neo4jServer	2.3.0	Neo4j-Management
Function	Remove-Neo4jSetting	2.3.0	Neo4j-Management
Function	Restart-Neo4jArbiter	2.3.0	Neo4j-Management
Function	Restart-Neo4jServer	2.3.0	Neo4j-Management
Function	Set-Neo4jSetting	2.3.0	Neo4j-Management
Function	Start-Neo4jArbiter	2.3.0	Neo4j-Management
Function	Start-Neo4jBackup	2.3.0	Neo4j-Management
Function	Start-Neo4jImport	2.3.0	Neo4j-Management
Function	Start-Neo4jServer	2.3.0	Neo4j-Management
Function	Start-Neo4jShell	2.3.0	Neo4j-Management
Function	Stop-Neo4jArbiter	2.3.0	Neo4j-Management
Function	Stop-Neo4jServer	2.3.0	Neo4j-Management
Function	Uninstall-Neo4jArbiter	2.3.0	Neo4j-Management
Function	Uninstall-Neo4jServer	2.3.0	Neo4j-Management

The module also supports the standard PowerShell help commands.

```
Get-Help Initialize-Neo4jServer
```

To see examples for a command, do like this:

```
Get-Help Initialize-Neo4jServer -examples
```

Basic Examples

- Retrieve basic information about the Neo4j Server e.g. Version and Edition.

```
Get-Neo4jServer C:\Neo4j
```

- Retrieve all of the settings of a Neo4j Server and display in a nice table

```
'C:\Neo4j' | Get-Neo4jSetting | `  
Select ConfigurationFile, Name, Value | `  
Format-Table
```

- The module uses the pipeline so you can export the settings, modify or filter them. For example, only show settings with the value of True:

```
'C:\Neo4j' | Get-Neo4jSetting | `  
Where { $_.Value -eq 'True' } | `  
Select ConfigurationFile, Name, Value | `  
Format-Table
```

- Quickly configure a Neo4j Server from saved settings in a CSV file.

```
Import-Csv -Path 'C:\Neo4jSettings.CSV' | Set-Neo4jSetting -Force
```

Advanced examples

- You can quickly configure and start an interactive console version of a Neo4j Server like this:

```
'C:\Neo4j' | `  
Initialize-Neo4jServer -ListenOnIPAddress 127.0.0.1 -PassThru | `  
Start-Neo4jServer -Console -Wait
```

To stop the server, issue **ctrl-c** in the console window that was created by the command.

- You can quickly configure and start a Service version of a Neo4j Server.



Note

The following must be executed from an elevated PowerShell prompt, where the Neo4j module has been imported into the session.

```
'C:\Neo4j' | `  
Initialize-Neo4jServer -ListenOnIPAddress 127.0.0.1 -PassThru | `  
Install-Neo4jServer -PassThru | `  
Start-Neo4jServer
```

To stop the server do this:

```
'C:\Neo4j' | Stop-Neo4jServer
```

- Create a three node cluster on the local computer. This example assumes three installations of the Enterprise version of Neo4j installed at C:\Neo4j-1,C:\Neo4j-2 and C:\Neo4j-3.

```
'C:\Neo4j-1' |  
Initialize-Neo4jServer `  
-ListenOnIPAddress 127.0.0.1 `  
-HTTPPort 7474 `  
-OnlineBackupServer '127.0.0.1:6362' `  
-PassThru |  
Initialize-Neo4jHACluster `  
-ServerID 1 `  
-InitialHosts '127.0.0.1:5001' `  
-ClusterServer '127.0.0.1:5001' `  
-HAServer '127.0.0.1:6001' `  
-PassThru |  
Start-Neo4jServer -Console  
  
'C:\Neo4j-2' |  
Initialize-Neo4jServer `  
-ListenOnIPAddress 127.0.0.1 `  
-HTTPPort 7475 `  
-ClearExistingDatabase `  
-OnlineBackupServer '127.0.0.1:6363' `  
-PassThru |  
Initialize-Neo4jHACluster `  
-ServerID 2 `  
-InitialHosts '127.0.0.1:5001' `  
-ClusterServer '127.0.0.1:5002' `  
-HAServer '127.0.0.1:6002' `  
-DisallowClusterInit `  
-PassThru |  
Start-Neo4jServer -Console  
  
'C:\Neo4j-3' |  
Initialize-Neo4jServer `  
-ListenOnIPAddress 127.0.0.1 `  
-HTTPPort 7476 `  
-ClearExistingDatabase `  
-OnlineBackupServer '127.0.0.1:6364' `  
-PassThru |  
Initialize-Neo4jHACluster `  
-ServerID 3 `  
-InitialHosts '127.0.0.1:5001' `  
-ClusterServer '127.0.0.1:5003' `  
-HAServer '127.0.0.1:6003' `  
-DisallowClusterInit `  
-PassThru |  
Start-Neo4jServer -Console
```

Common PowerShell parameters

The module commands support the common PowerShell parameters of Verbose, Debug, WhatIf etc.

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 see [Section 25.5, "Upgrade of a Neo4j HA Cluster" \[484\]](#).

A database created by an older version of Neo4j will be upgraded during startup when opened by Neo4j 2.3.1. While 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 → 2.3.1	Explicit configuration is required
2.0.x → 2.3.1	Explicit configuration is required
2.1.x → 2.3.1	Explicit configuration is required
2.2.x → 2.3.1	Explicit configuration is required



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 2.3.1, 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 2.3.1, 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:

```
wrapper.java.additional=-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005
```

This configuration will start a Neo4j server ready for remote debugging attachment 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

UDC is easily turned off by disabling it in the database configuration. In `conf/neo4j.properties` for Neo4j server or in the configuration passed to the database in embedded mode.

Usage Data Collector configuration settings

- `neo4j.ext.udc.enabled`: Enable the UDC extension.

`neo4j.ext.udc.enabled`

Description	Enable the UDC extension.
Valid values	<code>neo4j.ext.udc.enabled</code> is a boolean.
Default value	<code>true</code>

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. 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. Having too large heap may also hurt performance so you may have to try some different heap sizes.
- Use a concurrent garbage collector. We find that -XX:+UseG1GC works well in most use-cases.
- Give the Neo4j page cache generous amounts of memory. On a dedicated server, there are four big memory allocations you need to balance: the operating system, the Neo4j JVM, the Neo4j page cache and the paging memory for the Lucene indexes.
 - The operating system on a dedicated server can usually make do with 1 to 2 GBs of memory, but the more physical memory the machine has, the more memory the operating system will need.
 - The Neo4j JVM needs enough heap memory for the transaction state and query processing, plus some head-room for the garbage collector. Because the heap memory needs are so workload dependent, it is common to see configurations from 1 GB, up to 32 GBs of heap memory.
 - The Neo4j page cache should preferably have enough memory to keep the entire data set in memory, which is to say, the page cache should be big enough to fit all of the `neostore.*` files that are not `neostore.transaction.db.*` files.
 - Lastly, leave enough memory for the operating system page cache, to fit the contents of the `index` and `schema` directories, since it will impact index lookup performance if the indexes cannot fit in memory.

See the [Section 24.4, “Performance Guide” \[457\]](#) chapter for more information on how to tune Neo4j.

How to add configuration settings

When the database is given no configuration, it will try to use reasonable defaults. This is seldom optimal, however, because the database and the JVM have no knowledge about your workload or your performance requirements.

The way you configure Neo4j depend on your mode of deployment; whether you are using the database as an embedded library, or as a server.

Embedded	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” [578] for an example.
Server	When using the Neo4j REST server, see Section 24.2, “Server Configuration” [452] 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 \[502\]](#)

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 parameter:

```
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" \[456\]](#). 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.

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](#)¹ meaning that most Web server tooling can automatically 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.

¹ http://en.wikipedia.org/wiki/Common_Log_Format

**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.

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/neo4j-gc.log* or whatever directory you set the option to.

Web Interface configuration settings

Whitelist for remote guides in Browser

The Browser can :play guides from remote locations. You can specify a whitelist of hosts from where the Browser will be allowed to fetch content from.

In the *conf/neo4j-server.properties* file:

```
# To allow default hosts
dbms.browser.remote_content_hostname_whitelist="http://guides.neo4j.com,https://guides.neo4j.com,http://localhost,https://localhost"

# To allow all hosts (enterprise edition only)
dbms.browser.remote_content_hostname_whitelist="*"
```

Outgoing connections for Browser

The Browser can load services and data from remote locations and with this configuration this can be restricted.

In the *conf/neo4j-server.properties* file:

```
dbms.security.allow_outgoing_browser_connections=true
```

Disabling console types in Webadmin

You may, for security reasons, want to disable 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

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
<code>wrapper.java.initmemory</code>	initial heap size (in MB)
<code>wrapper.java.maxmemory</code>	maximum heap size (in MB)
<code>wrapper.java.additional</code>	additional literal JVM parameter

For more information on the tuning properties, see [Section 24.4, “Performance Guide” \[457\]](#).

24.4. Performance Guide

This is the Neo4j performance guide. It will attempt to give you guidance on how to tune Neo4j to achieve maximum performance.

Try this first

The first thing to look at, when Neo4j is not performing as expected, is to make sure that your Cypher queries do not do more work than they have to. For instance, a query might, unbeknownst to the author, mandate the production of a large cartesian product; or it might perform an expensive label-scan, because a certain label/property combination isn't indexed. The [Chapter 15, Query Tuning \[255\]](#) chapter has more information on how to investigate Cypher performance issues.

The second thing to look at, is to make sure that the Neo4j Java process has enough memory to do its work. If there is not enough memory to keep the JVM heap resident, then the OS will swap it out to storage. When a garbage collection happens, the swapped out heap memory has to be swapped in again, and something else will have to be swapped out. This swap-thrashing effect has a dramatic impact on the performance of the database, rendering it practically unusable. A well-tuned Neo4j database should not have any swap activity in its steady-state.

Next, make sure the JVM has enough memory, and isn't spending too much time in garbage collection. 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 GC-trashing happens.

Start the JVM with `-server` flag and `-Xmx<good sized heap>`, f.ex. `-Xmx512m` for 512 MiB memory or `-Xmx3g` for 3GiB memory. Having too large heap may also hurt performance so you may have to try out some different heap sizes. Make sure you are using a concurrent garbage collector. We find that `-XX:+UseG1GC` works well in most use-cases.

The next thing to look at, is the file caching memory. Neo4j uses its own page cache for the store files, and relies on the operating system for caching the index files. Make sure that the `dbms.pagecache.memory` setting (in `neo4j.properties`) is large enough to fit the entire store, if possible. But also make sure that you are not allocating so much memory to the JVM and the Neo4j page cache, that there is no memory left for the operating system to cache the Lucene index files. For more information on configuration see [Chapter 24, Configuration & Performance \[450\]](#).

Configuring heap size and GC

The size of the JVM heap is an important aspect of the performance of any Java application. The heap is separated into an old generation and a young generation. New objects are allocated in the young generation, and then later moved to the old generation if they stay live (in use) for long enough. When a generation fills up, the garbage collector performs a collection, during which all other threads in the process are paused. The young generation is quick to collect since the pause time correlates with the *live set* of objects, and is independent of the size of the young generation. In the old generation, pause times roughly correlates with the size of the heap. For this reason, the heap should ideally be sized and tuned such that transaction and query state never makes it to the old generation.



Note

When using Neo4j Server, JVM configuration goes into the `conf/neo4j-wrapper.conf` file, see [Section 24.2, "Server Configuration" \[452\]](#).

In server deployments, the heap size is configured with the `wrapper.java.maxmemory` (in MBs) setting in the `neo4j-wrapper.conf` file. For embedded, you specify the heap size by giving the `-Xmx??m` command line flag to the java process, where the `???` is the maximum heap size in MBs. The initial size of the heap is specified by the `wrapper.java.initmemory` setting, or with the `-Xms??m` flag, or chosen heuristically by the JVM itself if left unspecified. The JVM will automatically grow the heap as needed, up to the maximum size. The growing of the heap requires a full GC cycle, so if you know that you will need all the heap memory, you can set the initial heap size and the maximum heap size to the same value, and avoid the GC pauses that would otherwise be required to grow the heap.

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

The ratio of the size between the old generation and the new generation of the heap, is controlled by the `-XX:NewRatio=N` flag, where N is typically between 2 and 8 by default. A ratio of 2 means that the old generation size, divided by the new generation size, is equal to 2. In other words, two thirds of the heap memory will be dedicated to the old generation. A ratio of 3 will dedicate three quarters of the heap to the old generation, and a ratio of 1 will keep the two generations about the same size. A ratio of 1 is quite aggressive, but may be necessary if your transactions changes a lot of data. Having a large new generation can also be important if you run Cypher queries that needs to keep a lot of data resident, e.g. for sorting big result sets.

If the new generation is too small, short-lived objects might be moved to the old generation too soon. This is called premature promotion, and will slow the database down by increasing the frequency of old generation GC cycles. If the new generation is too big, the GC might decide that the old generation does not have enough space to fit all the objects it expects to promote from the new to the old generation. This turns new generation GC cycles into old generation GC cycles, again slowing the database down. Running more concurrent threads means that more allocations can take place in a given span of time, in turn increasing the pressure on the new generation in particular.

Be aware that configuring a heap size larger than 32 GiBs will disable a feature in the JVM called Compressed OOPs. When the heap size is less than 32 GiBs, the JVM can compress object references to only use 32 bits. This saves a lot of heap memory, and means that the gains from a larger heap are small or even negative, if you cannot give it at least 64 GiBs.

Neo4j has a number of long-lived objects, that stay around in the old generation, effectively for the lifetime of the Java process. To process them efficiently, and without adversely affecting the GC pause time, we recommend using a concurrent garbage collector.



Tip

The recommended garbage collector to use when running Neo4j in production is the G1 garbage collector. G1 is turned on by default in server deployments. For embedded deployments, it can be turned on by supplying `-XX:+UseG1GC` as a JVM parameter.

Tuning the specific GC algorithm depends on both the JVM version and the workload. It is recommended that you test your GC settings under realistic load for days or weeks. Problems like heap fragmentation can take a long time to surface.

Disk, 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 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 amount of RAM assigned to the page cache is too small. A new good SATA enabled SSD has an average seek time of less than 100 microseconds, meaning those scenarios will execute at least 50 times faster. However, this is still tens or hundreds of times slower than accessing RAM.

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 (nodes, relationships and properties) with 2-3 GBs of RAM. A server with 8-16 GBs of RAM can handle graphs with hundreds of millions of primitives, and a good server with 16-64 GBs can 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 `dstat` or `vmstat` to gather information when your application is running. If the swap or paging numbers are high, then it's a sign that the Lucene indexes don't quite fit in memory. In this case, queries that do index lookups will have high latencies.

When Neo4j starts up, its page cache is empty and needs to warm up. This can take a while, especially for large stores, so it's not uncommon to see a long period with many blocks being read from the drive, and high IO wait times.

Neo4j also flushes its page cache in the background, so it is not uncommon to see a steady trickle of blocks being written to the drive, during steady-state. This background flushing only produces a small amount of IO wait, however. If the IO wait times are high during steady-state, then it might be a sign that Neo4j is bottle-necked on the random IO performance of the drive. The best drives for running Neo4j are fast SSDs that can take lots of random IOPS.

Linux file system tuning

Databases often produce many small and random reads when querying data, and few sequential writes when committing changes. Neo4j is no different in this regard.

By default, most Linux distributions schedule IO requests using the Completely Fair Queuing (CFQ) algorithm, which provides a good balance between throughput and latency. The particular IO workload of a database, however, is better served by the Deadline scheduler. The Deadline scheduler gives preference to *read* requests, and processes them as soon as possible. This tends to decrease the latency of reads, while the latency of writes goes up. Since the writes are usually sequential, their lingering in the IO queue increases the chance of overlapping or adjacent write requests being merged together. This effectively reduces the number of writes that are sent to the drive.

On Linux, the IO scheduler for a drive, in this case `sda`, can be changed at runtime like this:

```
$ echo 'deadline' > /sys/block/sda/queue/scheduler
$ cat /sys/block/sda/queue/scheduler
noop [deadline] cfq
```

Another recommended practice is to disable file and directory access time updates. This way, the file system won't have to issue writes that update this meta-data, thus improving write performance. You do this by setting the `noatime,nodiratime` mount options in your `fstab`, or when you issue your disk mount command.

There may be other tuning options relevant to your specific file system of choice, but make sure that barriers are enabled. Barriers prevent certain reorderings of writes. They are important for maintaining the integrity of the transaction log, in case a power failure happens.

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 `ulimit -n` 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.

24.5. 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	"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"

24.6. Compressed storage of property values

Neo4j can in many cases compress and inline the storage of property values, such as short arrays and strings.

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 original 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 $4 \times 3 = 12$ bits. The array `long[] {-1L, 1L, 2L, 4L}` however will require the whole 64 bits for the -1 entry so it needs $64 \times 4 = 32$ bytes and it will end up in the dynamic store.

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, Alphanumeric 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.

24.7. Memory mapped IO settings

Introduction

Quick info

- The page cache is sometimes called *low level cache*, *file system cache* or *file buffer 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.

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 manner², 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` or `4g`. The default page memory setting is 50% 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. For instance, on a posix system you can look at the total of running `$ du -hc *store.db*` in your `data/graph.db` directory. 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.

Configuration

Parameter	Possible values	Effect
<code>dbms.pagecache.memory</code>	The maximum amount of memory to use for the page cache, either in bytes, or greater byte-like units, such as <code>150m</code> or <code>4g</code> .	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

²This 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.

Parameter	Possible values	Effect
	as 100m for 100 mega-bytes, or 4g for 4 giga-bytes.	be zero. For extremely small and memory constrained deployments, it is recommended to still reserve at least a couple of megabytes for the page cache.
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 page cache 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 50% of what is left for the page cache. This is the default configuration when nothing else is specified.

Batch insert example

Read general information on batch insertion in [Chapter 36, Batch Insertion \[634\]](#).

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.

24.8. Configuration Settings Reference

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" \[452\]](#)). 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" \[578\]](#)).

For additional configuration settings, see:

- [Section 24.2, "Server Configuration" \[452\]](#)
- [Settings for the remote shell extension \[545\]](#)
- [High Availability configuration settings \[477\]](#)
- [Cluster configuration settings \[479\]](#)
- [Online backup configuration settings \[498\]](#)
- [Consistency check configuration settings \[498\]](#)
- [Usage Data Collector configuration settings \[449\]](#)
- [Metrics settings \[525\]](#)

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.
- `cypher_parser_version`: Set this to specify the default parser (language version).
- `dbms.checkpoint.interval.time`: Configures the time interval between check-points.
- `dbms.checkpoint.interval.tx`: Configures the transaction interval between check-points.
- `dbms.cypher.hints.error`: Set this to specify the behavior when Cypher planner or runtime hints cannot be fulfilled.
- `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 for the default language version.
- `dbms.cypher.statistics_divergence_threshold`: The threshold when a plan is considered stale.
- `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.pagecache.swapper`: Specify which page swapper to use for doing paged IO.
- `dbms.querylog.enabled`: Log executed queries that takes longer than the configured threshold.
- `dbms.querylog.filename`: Log executed queries that take longer than the configured threshold.
- `dbms.querylog.max_archives`: Maximum number of history files for the query log.
- `dbms.querylog.rotation.threshold`: Specifies at which file size the query log will auto-rotate.
- `dbms.querylog.threshold`: If the execution of query takes more time than this threshold, the query is logged - provided query logging is enabled.
- `dbms.security.load_csv_file_url_root`: Sets the root directory for file URLs used with the Cypher `LOAD CSV` clause.
- `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.
- `query_cache_size`: The number of Cypher query execution plans that are cached.
- `read_only`: Only allow read operations from this Neo4j instance.
- `relationship_grab_size`: How many relationships to read at a time during iteration.
- `store.internal_log.level`: Log level threshold.
- `store.internal_log.max_archives`: Maximum number of history files for the internal log.
- `store.internal_log.rotation_threshold`: Threshold for rotation of the internal log.

Deprecated settings

- `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.
- `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.
- `store_dir`: The directory where the database files are located.
- `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	allow_file_urls 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	allow_store_upgrade is a boolean.
Default value	false

batched_writes

Description	Whether or not transactions are appended to the log in batches.
Valid values	batched_writes is a boolean.
Default value	true
Deprecated	Write batching can no longer be turned off.

cache_type

Description	The type of cache to use for nodes and relationships. This configuration setting is no longer applicable from Neo4j 2.3. Configuration has been simplified to only require tuning of the page cache.
Valid values	cache_type is a string.
Default value	deprecated
Deprecated	The cache_type configuration setting has been deprecated.

cypher_parser_version

Description	Set this to specify the default parser (language version).
Valid values	cypher_parser_version is one of 1.9, 2.2, 2.3, default.
Default value	default

dbms.checkpoint.interval.time

Description	Configures the time interval between check-points. The database will not check-point more often than this (unless check pointing is triggered by a different event), but might check-point less often than this interval, if performing a check-point takes longer time than the configured interval. A check-point is a point in the transaction logs, from which recovery would start from. Longer check-point intervals typically means that recovery will take longer to complete in case of a crash. On the other hand, a longer check-point interval can also reduce the I/O load that the database places on the system, as each check-point implies a flushing and forcing of all the store files. The default is 5m for a check-point every 5 minutes. Other supported units are s for seconds, and ms for milliseconds.
Valid values	dbms.checkpoint.interval.time is a duration (valid units are ms, s, m).
Default value	5m

dbms.checkpoint.interval.tx

Description	Configures the transaction interval between check-points. The database will not check-point more often than this (unless check pointing is triggered by a different event), but might check-point less often than this interval, if performing a check-point takes longer time than the configured interval. A check-point is a point in the transaction logs, from which recovery would start from. Longer check-point intervals typically means that recovery will take longer to complete in case of a crash. On the other hand, a longer check-point interval can also reduce the I/O load that the database places on the system, as each check-point implies a flushing and forcing of all the store files. The default is 100000 for a check-point every 100000 transactions.
Valid values	dbms.checkpoint.interval.tx is an integer which is minimum 1.
Default value	100000

dbms.cypher.hints.error

Description	Set this to specify the behavior when Cypher planner or runtime hints cannot be fulfilled. If true, then non-conformance will result in an error, otherwise only a warning is generated.
Valid values	dbms.cypher.hints.error is a boolean.
Default value	false

dbms.cypher.min_replan_interval

Description	The minimum lifetime of a query plan before a query is considered for replanning.
--------------------	---

Valid values	dbms.cypher.min_replan_interval is a duration (valid units are ms, s, m).
Default value	1s
dbms.cypher.planner	
Description	Set this to specify the default planner for the default language version.
Valid values	dbms.cypher.planner is one of COST, RULE, default.
Default value	default
dbms.cypher.statistics_divergence_threshold	
Description	The threshold when a plan is considered stale. If any of the underlying statistics used to create the plan has changed more than this value, the plan is considered stale and will be replanned. A value of 0 means always replan, and 1 means never replan.
Valid values	dbms.cypher.statistics_divergence_threshold is a double which is minimum 0.0, and is maximum 1.0.
Default value	0.5
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 50% of RAM minus the max Java heap size.
Valid values	dbms.pagecache.memory is a byte size (valid multipliers are k, m, g, K, M, G) which is minimum 245760.
Default value	3443740672
dbms.pagecache.swapper	
Description	Specify which page swapper to use for doing paged IO. This is only used when integrating with proprietary storage technology.
Valid values	dbms.pagecache.swapper is a string.
dbms.querylog.enabled	
Description	Log executed queries that takes longer than the configured threshold. <i>NOTE: This feature is only available in the Neo4j Enterprise Edition.</i>
Valid values	dbms.querylog.enabled is a boolean.
Default value	false
dbms.querylog.filename	
Description	Log executed queries that take longer than the configured threshold.
Valid values	dbms.querylog.filename is a path.
dbms.querylog.max_archives	
Description	Maximum number of history files for the query log.
Valid values	dbms.querylog.max_archives is an integer which is minimum 1.
Default value	7

dbms.querylog.rotation.threshold

Description	Specifies at which file size the query log will auto-rotate. 0 means that no rotation will automatically occur based on file size.
Valid values	dbms.querylog.rotation.threshold is a byte size (valid multipliers are k, m, g, K, M, G) which is minimum 0, and is maximum 9223372036854775807.
Default value	20m

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	dbms.querylog.threshold is a duration (valid units are ms, s, m).
Default value	0s

dbms.security.load_csv_file_url_root

Description	Sets the root directory for file URLs used with the Cypher LOAD CSV clause. This must be set to a single directory, restricting access to only those files within that directory and its subdirectories.
Valid values	dbms.security.load_csv_file_url_root is a path.

dense_node_threshold

Description	Relationship count threshold for considering a node to be dense.
Valid values	dense_node_threshold is an integer which is minimum 1.
Default value	50

dump_configuration

Description	Print out the effective Neo4j configuration after startup.
Valid values	dump_configuration is a boolean.
Default value	false

index_background_sampling_enabled

Description	Enable or disable background index sampling.
Valid values	index_background_sampling_enabled is a boolean.
Default value	true

index_sampling_buffer_size

Description	Size of buffer used by index sampling.
Valid values	index_sampling_buffer_size 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	index_sampling_update_percentage 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	keep_logical_logs 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	log_mapped_memory_stats 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	log_mapped_memory_stats_filename is a path which is relative to store_dir.
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	log_mapped_memory_stats_interval 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	logical_log_rotation_threshold 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	lucene_searcher_cache_size 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	neostore.nodestore.db.mapped_memory is a byte size (valid multipliers are k, m, g, K, M, G).
Deprecated	Replaced by the dbms.pagecache.memory setting.
neostore.propertystore.db.arrays.mapped_memory	
Description	The size to allocate for memory mapping the array property store.

Valid values	neostore.propertystore.db.arrays.mapped_memory is a byte size (valid multipliers are k, m, g, K, M, G).
Deprecated	Replaced by the dbms.pagecache.memory setting.
neostore.propertystore.db.index.keys.mapped_memory	
Description	The size to allocate for memory mapping the store for property key strings.
Valid values	neostore.propertystore.db.index.keys.mapped_memory is a byte size (valid multipliers are k, m, g, K, M, G).
Deprecated	Replaced by the dbms.pagecache.memory setting.
neostore.propertystore.db.index.mapped_memory	
Description	The size to allocate for memory mapping the store for property key indexes.
Valid values	neostore.propertystore.db.index.mapped_memory is a byte size (valid multipliers are k, m, g, K, M, G).
Deprecated	Replaced by the dbms.pagecache.memory setting.
neostore.propertystore.db.mapped_memory	
Description	The size to allocate for memory mapping the property value store.
Valid values	neostore.propertystore.db.mapped_memory is a byte size (valid multipliers are k, m, g, K, M, G).
Deprecated	Replaced by the dbms.pagecache.memory setting.
neostore.propertystore.db.strings.mapped_memory	
Description	The size to allocate for memory mapping the string property store.
Valid values	neostore.propertystore.db.strings.mapped_memory is a byte size (valid multipliers are k, m, g, K, M, G).
Deprecated	Replaced by the dbms.pagecache.memory setting.
neostore.relationshipstore.db.mapped_memory	
Description	The size to allocate for memory mapping the relationship store.
Valid values	neostore.relationshipstore.db.mapped_memory is a byte size (valid multipliers are k, m, g, K, M, G).
Deprecated	Replaced by the dbms.pagecache.memory setting.
query_cache_size	
Description	The number of Cypher query execution plans that are cached.
Valid values	query_cache_size 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	read_only is a boolean.
Default value	false
relationship_grab_size	
Description	How many relationships to read at a time during iteration.
Valid values	relationship_grab_size is an integer which is minimum 1.

Default value	100
----------------------	-----

store.internal_log.level

Description	Log level threshold.
Valid values	store. internal_log. level is one of DEBUG, INFO, WARN, ERROR, NONE.
Default value	INFO

store.internal_log.max_archives

Description	Maximum number of history files for the internal log.
Valid values	store. internal_log. max_archives is an integer which is minimum 1.
Default value	7

store.internal_log.rotation_threshold

Description	Threshold for rotation of the internal log.
Valid values	store. internal_log. rotation_threshold is a byte size (valid multipliers are k, m, g, K, M, G) which is minimum 0, and is maximum 9223372036854775807.
Default value	20m

store_dir

Description	The directory where the database files are located.
Valid values	store_dir is a path.
Deprecated	This is no longer used.

use_memory_mapped_buffers

Description	Use memory mapped buffers for accessing the native storage layer.
Valid values	use_memory_mapped_buffers 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.

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:

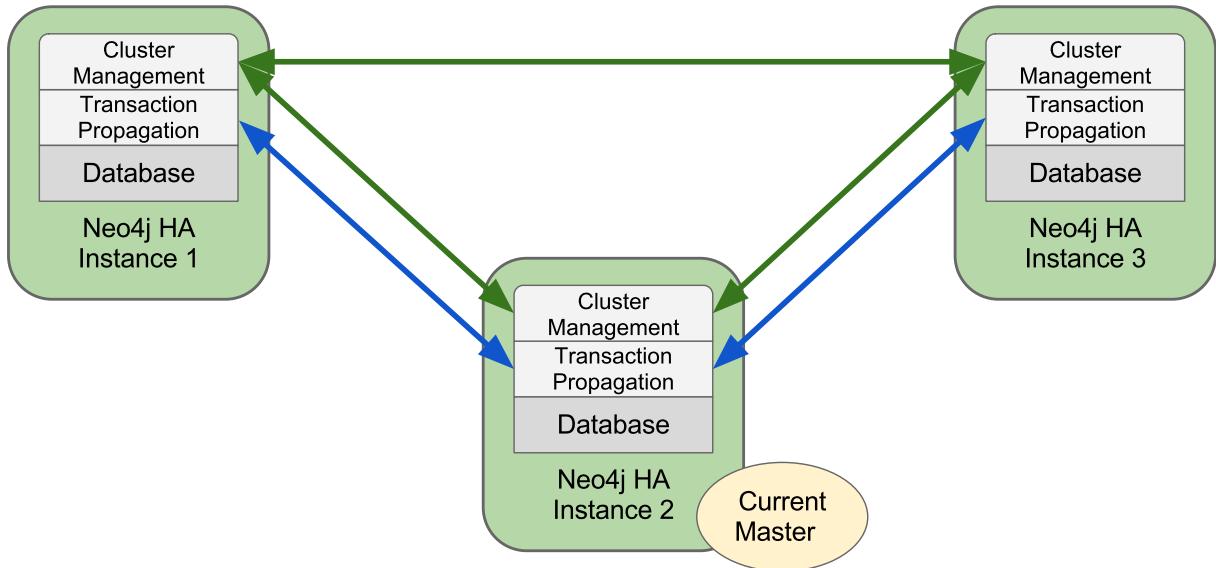
1. It enables a *fault-tolerant database architecture*, where several Neo4j slave databases can be configured to be exact replicas of a single Neo4j master database. This allows the end-user system to be fully functional and both read and write to the database in the event of hardware failure.
2. It 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. HA 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” \[486\]](#).

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`.

Settings available in the Enterprise server

- `dbms.browser.credential_timeout`: Configure the Neo4j Browser to time out logged in users after this idle period.
- `dbms.browser.remote_content_hostname_whitelist`: Whitelist of hosts for the Neo4j Browser to be allowed to fetch content from.
- `dbms.browser.store_credentials`: Configure the Neo4j Browser to store or not store user credentials.
- `dbms.security.allow_outgoing_browser_connections`: Configure the policy for outgoing Neo4j Browser connections.
- `org.neo4j.server.database.mode`: Configure the operating mode of the database — *SINGLE* for stand-alone operation or *HA* for operating as a member in a cluster.

`dbms.browser.credential_timeout`

Description	Configure the Neo4j Browser to time out logged in users after this idle period. Setting this to 0 indicates no limit.
Valid values	<code>dbms.browser.credential_timeout</code> is a duration (valid units are <code>ms</code> , <code>s</code> , <code>m</code>).
Default value	0

`dbms.browser.remote_content_hostname_whitelist`

Description	Whitelist of hosts for the Neo4j Browser to be allowed to fetch content from.
Valid values	<code>dbms.browser.remote_content_hostname_whitelist</code> is a string.
Default value	

`dbms.browser.store_credentials`

Description	Configure the Neo4j Browser to store or not store user credentials.
Valid values	<code>dbms.browser.store_credentials</code> is a boolean.
Default value	true

`dbms.security.allow_outgoing_browser_connections`

Description	Configure the policy for outgoing Neo4j Browser connections.
--------------------	--

Valid values	dbms.security.allow_outgoing_browser_connections is a boolean.
Default value	true
org.neo4j.server.database.mode	
Description	Configure the operating mode of the database — <i>SINGLE</i> for stand-alone operation or <i>HA</i> for operating as a member in a cluster.
Valid values	org.neo4j.server.database.mode is a string.
Default value	SINGLE

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

- [dbms.security.ha_status_auth_enabled](#): Require authorization for access to the HA status endpoints.
- [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.internal_state_switch_timeout](#): Timeout for waiting for internal conditions during state switch, like for transactions to complete, before switching to master or slave.
- [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_apply_batch_size](#): Size of batches of transactions applied on slaves when pulling from 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 request threads 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.

dbms.security.ha_status_auth_enabled

Description	Require authorization for access to the HA status endpoints.
Valid values	dbms.security.ha_status_auth_enabled is a boolean.
Default value	true

ha.branched_data_policy

Description	Policy for how to handle branched data.
Valid values	ha.branched_data_policy is one of <i>keep_all</i> , <i>keep_last</i> , <i>keep_none</i> .
Default value	<i>keep_all</i>

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	ha.com_chunk_size is a byte size (valid multipliers are k, m, g, K, M, G) which is minimum 1024.

Default value	2M
ha.internal_state_switch_timeout	
Description	Timeout for waiting for internal conditions during state switch, like for transactions to complete, before switching to master or slave.
Valid values	ha. internal_state_switch_timeout is a duration (valid units are ms, s, m).
Default value	10s
ha.lock_read_timeout	
Description	Timeout for taking remote (write) locks on slaves. Defaults to .
Valid values	ha. lock_read_timeout 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	ha. max_concurrent_channels_per_slave is an integer which is minimum 1.
Default value	20
ha.pull_apply_batch_size	
Description	Size of batches of transactions applied on slaves when pulling from master.
Valid values	ha. pull_apply_batch_size is an integer.
Default value	100
ha.pull_interval	
Description	Interval of pulling updates from master.
Valid values	ha. pull_interval 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	ha. read_timeout 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	ha. server 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	ha. slave_only is a boolean.
Default value	false
ha.state_switch_timeout	
Description	Timeout for request threads waiting for instance to become master or slave.
Valid values	ha. state_switch_timeout 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	ha.tx_push_factor 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	ha.tx_push_strategy is one of 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	ha.allow_init_cluster 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 <code>and</code> settings.
Valid values	ha.broadcast_timeout is a duration (valid units are ms, s, m).
Default value	30s

ha.cluster_name

Description	The name of a cluster.
Valid values	ha.cluster_name 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	ha.cluster_server 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	ha.configuration_timeout 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	ha.default_timeout 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	ha.election_timeout 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	ha.heartbeat_interval 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	ha.heartbeat_timeout 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	ha.initial_hosts 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	ha.join_timeout is a duration (valid units are ms, s, m).
Default value	30s
ha.learn_timeout	
Description	Timeout for learning values. Defaults to .
Valid values	ha.learn_timeout 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	ha.leave_timeout 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	ha.paxos_timeout 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	ha.phase1_timeout 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	ha.phase2_timeout 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	ha.server_id 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 \[478\]](#) 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"](#) [486].

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 its status in exactly the same way as for Neo4j instances. See also [Section 23.2, "Server Installation"](#) [440].

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 2.3.1 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 2.3.1 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 2.3.1, 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 2.3.1 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 2.3.1 on all previous slaves **and remove their database store** (typically `data/graph.db`). *Slaves should not be started with a previous store present.*
- Note: The security configuration of the master is *not* propagated to the slaves. See the section called "[Copying security configuration from one instance to another](#)" [316] for more information.

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 between which automatic upgrades are possible. This typically means only within patch releases of the same Neo4j version. See [Section 23.4, "Upgrading" \[446\]](#) for more information.

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 three instances to be deployed on three 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 three 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 heartbeat 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 start, to allow them 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 configure

- Download Neo4j Enterprise from [the Neo4j download site](#)¹, and unpack on three separate machines.
- Configure the HA related settings for each installation as outlined below. Note that all three 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

```
# HA - High Availability
# SINGLE - Single mode, default.
org.neo4j.server.database.mode=HA

# Let the webserver only listen on the specified IP.
org.neo4j.server.webserver.address=0.0.0.0
```

Neo4j instance

#2 — neo4j-02.local

conf/neo4j.properties

¹ <http://neo4j.com/download/>

```
# 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

```
# HA - High Availability
# SINGLE - Single mode, default.
org.neo4j.server.database.mode=HA

# Let the webserver only listen on the specified IP.
org.neo4j.server.webserver.address=0.0.0.0
```

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

```
# HA - High Availability
# SINGLE - Single mode, default.
org.neo4j.server.database.mode=HA

# Let the webserver only listen on the specified IP.
org.neo4j.server.webserver.address=0.0.0.0
```

Neo4j instance
#3 — neo4j-03.local

Start the Neo4j Servers

Start the Neo4j servers as usual. Note that 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 start the second instance. To keep track of the startup state you can follow the messages in *console.log* — the path is printed before the startup script returns.

Now, you should be able to access the three servers and check their HA status. Open the locations below in a web browser and issue the following command in the editor after having set a password for the database: `:play sysinfo`

- <http://neo4j-01.local:7474/>
- <http://neo4j-02.local:7474/>
- <http://neo4j-03.local:7474/>

**Tip**

You can replace database #3 with an *arbiter* instance, see [Section 25.4, “Arbiter Instances” \[483\]](#).

That's it! You now have a Neo4j HA cluster of three 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 [Section 25.2, “HA Setup and configuration” \[476\]](#).

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 configure

1. Download Neo4j Enterprise from [the Neo4j download site](#)², and unpack into three separate directories on your test machine.
2. Configure the HA related settings for each installation as outlined below.

Neo4j instance #1 — ~/
neo4j-01

conf/neo4j.properties

```
# Reduce the default page cache memory allocation
dbms.pagecache.memory=500m

# Port to listen to for incoming backup requests.
online_backup_server = 127.0.0.1:6366

# 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 = 127.0.0.1:5001,127.0.0.1:5002,127.0.0.1:5003

# 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

# 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
```

conf/neo4j-server.properties

```
# HA - High Availability
# SINGLE - Single mode, default.
org.neo4j.server.database.mode=HA

# 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
```

² <http://neo4j.com/download/>

Neo4j instance #2 — ~/neo4j-02

conf/neo4j.properties

```
# Reduce the default page cache memory allocation
dbms.pagecache.memory=500m

# Port to listen to for incoming backup requests.
online_backup_server = 127.0.0.1:6367

# 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 = 127.0.0.1:5001,127.0.0.1:5002,127.0.0.1:5003

# 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

# 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
```

conf/neo4j-server.properties

```
# HA - High Availability
# SINGLE - Single mode, default.
org.neo4j.server.database.mode=HA

# 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
```

Neo4j instance #3 — ~/neo4j-03

conf/neo4j.properties

```
# Reduce the default page cache memory allocation
dbms.pagecache.memory=500m

# Port to listen to for incoming backup requests.
online_backup_server = 127.0.0.1:6368

# 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 = 127.0.0.1:5001,127.0.0.1:5002,127.0.0.1:5003

# 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

# 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
```

conf/neo4j-server.properties

```
# HA - High Availability
# SINGLE - Single mode, default.
org.neo4j.server.database.mode=HA

# 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
```

Start the Neo4j Servers

Start the Neo4j servers as usual. Note that 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 three servers and check their HA status. Open the locations below in a web browser and issue the following command in the editor after having set a password for the database: `:play sysinfo`

- <http://127.0.0.1:7474/>
- <http://127.0.0.1:7475/>
- <http://127.0.0.1:7476/>

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 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” \[492\]](#)) 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 backends.

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	<code>online_backup_enabled</code> is a boolean.
Default value	<code>true</code>

`online_backup_server`

Description	Listening server for online backups.
Valid values	<code>online_backup_server</code> is a hostname and port.
Default value	<code>0.0.0.0:6362-6372</code>

Consistency check configuration settings

- `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_graph`

Description	Perform checks between nodes, relationships, properties, types and tokens.
Valid values	<code>consistency_check_graph</code> is a boolean.
Default value	<code>true</code>

`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	<code>consistency_check_indexes</code> 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	consistency_check_label_scan_store 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	consistency_check_property_owners 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	consistency_check_report_file is a path.

26.2. Performing Backups



Tip

When using Neo4j in embedded mode, the way to perform backup is still the same.

Backup Commands

```
# 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.

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 the following:

```
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/2.3.1/javadocs/org/neo4j/backup/OnlineBackup.html>

26.3. Restoring Your Data

The Neo4j backups are fully functional databases. To use a backup, all you need to do replace your database folder with the backup. 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 Neo4j Server” \[503\]](#) for details.

27.1. Securing 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 want the server to listen to 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` which will cause the server to bind to all available network interfaces. Note that firewalls et cetera have to be configured accordingly as well.

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” \[316\]](#)).

Please refer to [Section 21.3, “REST API Authentication and Authorization” \[313\]](#) for additional details. When accessing Neo4j over unsecured networks, make sure HTTPS is configured and used for access (see [the section called “HTTPS support” \[503\]](#)).

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 web 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)
dbms.security.tls_certificate_file=ssl/snakeoil.cert

# Private key location (auto generated if the file does not exist)
dbms.security.tls_key_file=ssl/snakeoil.key
```

¹ http://en.wikipedia.org/wiki/Cross-site_scripting

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



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 presents a substantial security vulnerability.

By default, the Neo4j Server comes with some places where arbitrary code execution can happen. These are the [Section 21.17, “Traversals” \[375\]](#) 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 authentication and authorization (see the section called “[Server authentication and authorization](#)” [503]), if enabled.

Enforcing Server Authorization Rules

In this example, a (dummy) failing security rule is registered to deny access to all URLs 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 :-)
```

² <http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j.app%22%20AND%20a%3A%22neo4j-server%22>

```

@Override
public boolean isAuthorized( HttpServletRequest request )
{
    return false; // always fails - a production implementation performs
                  // 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 URLs 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"
-

Using a proxy

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_proxy³.

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.

³http://httpd.apache.org/docs/2.2/mod/mod_proxy.html

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

While it is possible to develop plugins using Neo4j's SecurityRule (see above), operations professionals would often prefer to configure proxy servers such as Apache. However, it should be noted that in cases where both approaches are being used, they will work harmoniously provided that 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_balancer⁴ plugin

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

LOAD CSV

The Cypher LOAD CSV clause can load files from the filesystem, and its default configuration allows any file on the system to be read using a file:/// URL. This presents a security vulnerability in production environments where database users should not otherwise have access to files on the system. For production deployments, configure the [dbms.security.load_csv_file_url_root \[470\]](#) setting, which will make all files identified in a file:/// URL relative to the specified directory, similarly to how a unix chroot works. Alternatively, set the [allow_file_urls \[467\]](#) setting to false, which disables the use of file:/// URLs entirely. Further information can be found in [Section 11.6, "Load CSV" \[184\]](#).

Neo4j Web Interface Security

For configuration settings to consider in order to get the level of security you want to achieve, see [the section called "Web Interface configuration settings" \[454\]](#).

⁴http://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. Neo4j Enterprise also has the ability to automatically report *metrics* to commonly used monitoring systems, like [Graphite](#)² and [Ganglia](#)³.

¹ <http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html>

² <http://graphite.wikidot.com>

³ <http://ganglia.sourceforge.net>

28.1. Adjusting remote JMX access to the Neo4j Server

Per default, the Neo4j Enterprise Server edition does 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

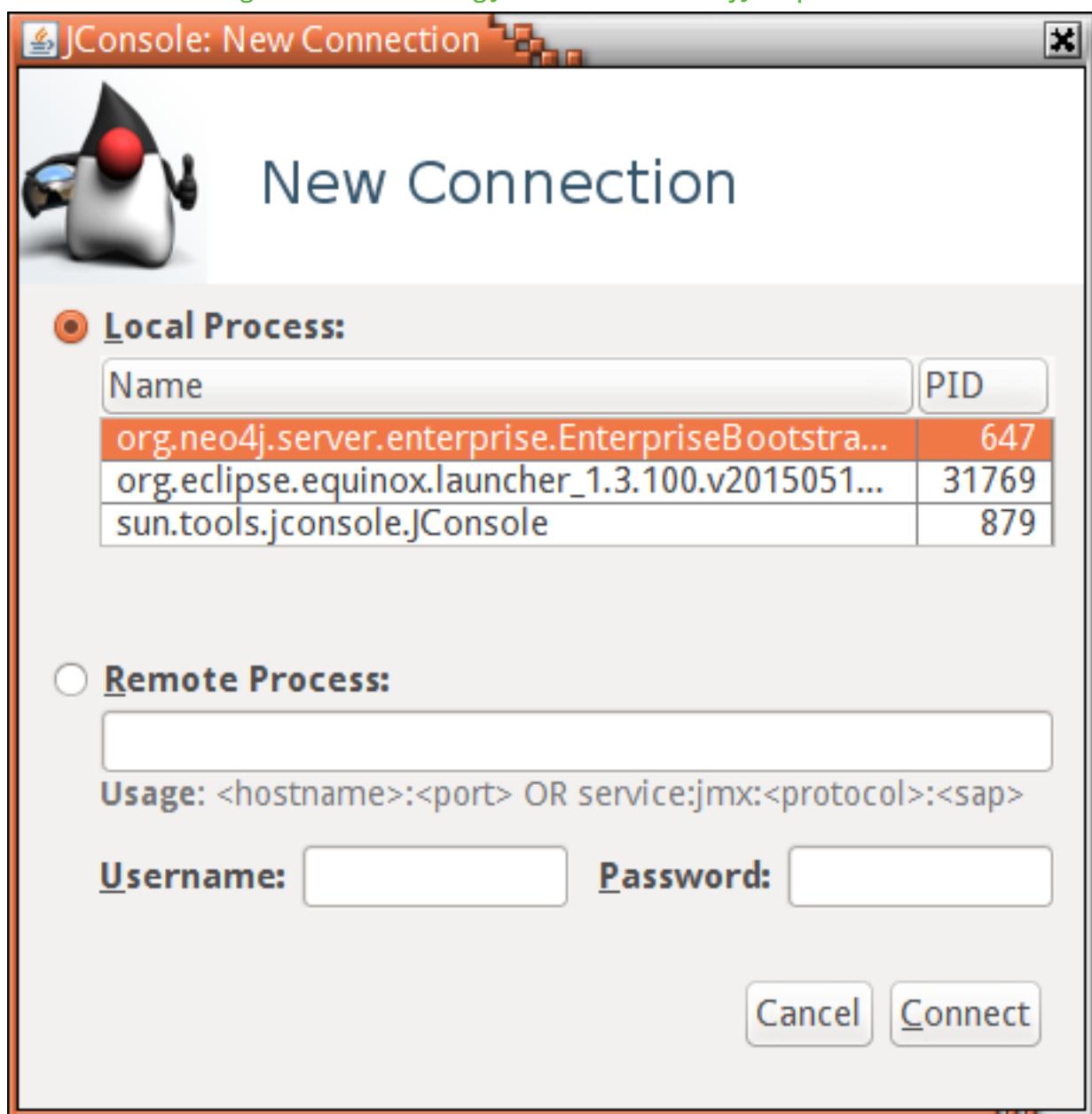
```
$NEO4j_HOME/bin/neo4j start
```

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” \[509\]](#) and [the JMX documentation](#)⁴. 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.

⁴ <http://docs.oracle.com/javase/7/docs/technotes/guides/management/agent.html>

Figure 28.2. Neo4j MBeans View

Monitoring & Management Console

Window Help

neo4j.server.enterprise.EnterpriseBootstrapper

Memory Threads Classes VM Summary MBeans

Attribute value

Name	Value
KernelStartTime	Tue Oct 06 12:46:37 CEST 2010

[Refresh](#)

MBeanAttributeInfo

Name	Value
Attribute:	
Name	KernelStartTime
Description	The time from which this Neo4j instance was in operation.
Readable	true
Writable	false
Is	false
Type	java.util.Date

Descriptor

Name	Value

neo4j.server.enterprise.EnterpriseBootstr...

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 33.11, “Reading a management attribute” \[602\]](#) for an example.

Once you have access to this information, you can use it to for instance expose the values to [SNMP](#)⁵ or other monitoring systems.

⁵ 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.
- **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 \[584\]](#) in the section called “Properties” [7].

MBean Branched Store (org.neo4j.management.BranchedStore) Attributes

Name	Description	Type	Read	Write
<i>Information about the branched stores present in this HA cluster member</i>				
BranchedStores	A list of the branched stores	org.neo4j.management.BranchedStoreInfo[] as CompositeData[]	yes	no

MBean Configuration (org.neo4j.jmx.impl.ConfigurationBean) Attributes

Name	Description	Type	Read	Write
<i>The configuration parameters used to configure Neo4j</i>				
allow_store_upgrade	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.	String	yes	no
array_block_size	Specifies the block size for storing arrays. This parameter is only honored when the store is created, otherwise it is ignored. The default block size is 120 bytes, and the overhead of each block is the same as for string blocks, i.e., 8 bytes.	String	yes	no
batch_inserter_batch_size	Specifies number of operations that batch inserter will try to group into one batch before flushing data into underlying storage.	String	yes	no
batched_writes	Whether or not transactions are appended to the log in batches	String	yes	no
cache_type	The type of cache to use for nodes and relationships. This configuration setting is no longer applicable from Neo4j 2.3.	String	yes	no

Name	Description	Type	Read	Write
	Configuration has been simplified to only require tuning of the page cache.			
cypher_parser_version	Set this to specify the default parser (language version).	String	yes	no
dbms.checkpoint.interval.time	Configures the time interval between check-points. The database will not check-point more often than this (unless check pointing is triggered by a different event), but might check-point less often than this interval, if performing a check-point takes longer time than the configured interval. A check-point is a point in the transaction logs, from which recovery would start from. Longer check-point intervals typically means that recovery will take longer to complete in case of a crash. On the other hand, a longer check-point interval can also reduce the I/O load that the database places on the system, as each check-point implies a flushing and forcing of all the store files. The default is <i>5m</i> for a check-point every 5 minutes. Other supported units are <i>s</i> for seconds, and <i>ms</i> for milliseconds.	String	yes	no
dbms.checkpoint.interval.tx	Configures the transaction interval between check-points. The database will not check-point more often than this (unless check pointing is triggered by a different event), but might check-point less often than this interval, if performing a check-point takes longer time than the configured interval. A check-point is a point in the transaction logs, from which recovery would start from. Longer check-point intervals typically means that recovery will take longer to complete in case of a crash. On the other hand, a longer check-point interval can also reduce the I/O load that the database places on the system, as each check-point implies a flushing and forcing of all the store files. The default is <i>100000</i> for a check-point every 100000 transactions.	String	yes	no
dbms.cypher.compiler_tracing	Enable tracing of compilation in cypher.	String	yes	no
dbms.cypher.hints.error	Set this to specify the behavior when Cypher planner or runtime hints cannot be fulfilled. If true, then non-conformance will result in an error, otherwise only a warning is generated.	String	yes	no

Name	Description	Type	Read	Write
dbms.cypher.planner	Set this to specify the default planner for the default language version.	String	yes	no
dbms.cypher.runtime	Set this to specify the default runtime for the default language version.	String	yes	no
dbms.pagecache.memory	The amount of memory to use for mapping the store files, in bytes (or kilobytes with the <i>k</i> suffix, megabytes with <i>m</i> and gigabytes with <i>g</i>). 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 50% of RAM minus the max Java heap size.	String	yes	no
dbms.pagecache.pagesize	Target size for pages of mapped memory. If set to 0, then a reasonable default is chosen, depending on the storage device used.	String	yes	no
dbms.pagecache.swapper	Specify which page swapper to use for doing paged IO. This is only used when integrating with proprietary storage technology.	String	yes	no
dbms.querylog.enabled	Log executed queries that takes longer than the configured threshold. <i>NOTE: This feature is only available in the Neo4j Enterprise Edition.</i>	String	yes	no
dbms.querylog.filename	Log executed queries that take longer than the configured threshold	String	yes	no
dbms.querylog.max_archives	Maximum number of history files for the query log.	String	yes	no
dbms.querylog.rotation.threshold	Specifies at which file size the query log will auto-rotate. 0 means that no rotation will automatically occur based on file size.	String	yes	no
dbms.querylog.threshold	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.	String	yes	no
dense_node_threshold	Relationship count threshold for considering a node to be dense	String	yes	no
dump_configuration	Print out the effective Neo4j configuration after startup.	String	yes	no
edition	Configuration attribute	String	yes	no
ephemeral	Configuration attribute	String	yes	no

Name	Description	Type	Read	Write
forced_kernel_id	An identifier that uniquely identifies this graph database instance within this JVM. Defaults to an auto-generated number depending on how many instance are started in this JVM.	String	yes	no
gc_monitor_threshold	The amount of time in ms the monitor thread has to be blocked before logging a message it was blocked.	String	yes	no
gc_monitor_wait_time	Amount of time in ms the GC monitor thread will wait before taking another measurement.	String	yes	no
ha.initial_hosts	Configuration attribute	String	yes	no
ha.server_id	Configuration attribute	String	yes	no
index_background_sampling_enabled	Enable or disable background index sampling	String	yes	no
index_sampling_buffer_size	Size of buffer used by index sampling	String	yes	no
index_sampling_update_percentage	Percentage of index updates of total index size required before sampling of a given index is triggered	String	yes	no
intercept_deserialized_transactions	Determines whether any TransactionInterceptors loaded will intercept externally received transactions (for example in HA) before they reach the logical log and are applied to the store.	String	yes	no
jmx.port	Configuration attribute	String	yes	no
keep_logical_logs	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.	String	yes	no
label_block_size	Specifies the block size for storing labels exceeding in-lined space in node record. This parameter is only honored when the store is created, otherwise it is ignored. The default block size is 60 bytes, and the overhead of each block is the same as for string blocks, i.e., 8 bytes.	String	yes	no
log_mapped_memory_stats_filename	The file where memory mapping statistics will be recorded.	String	yes	no
log_mapped_memory_stats_interval	The number of records to be loaded between regular logging of memory mapping statistics.	String	yes	no

Name	Description	Type	Read	Write
log_mapped_memory_stats	Log memory mapping statistics regularly.	String	yes	no
logical_log_rotation_threshold	Specifies at which file size the logical log will auto-rotate. 0 means that no rotation will automatically occur based on file size.	String	yes	no
neostore.nodestore.db.mapped_memory	The size to allocate for memory mapping the node store.	String	yes	no
neostore.propertystore.db.arrays.mapped_memory	The size to allocate for memory mapping the array property store.	String	yes	no
neostore.propertystore.db.index.keys.mapped_memory	The size to allocate for memory mapping the store for property key strings.	String	yes	no
neostore.propertystore.db.index.mapped_memory	The size to allocate for memory mapping the store for property key indexes.	String	yes	no
neostore.propertystore.db.mapped_memory	The size to allocate for memory mapping the property value store.	String	yes	no
neostore.propertystore.db.strings.mapped_memory	The size to allocate for memory mapping the string property store.	String	yes	no
neostore.relationshipstore.db.mapped_memory	The size to allocate for memory mapping the relationship store.	String	yes	no
node_auto_indexing	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 node_keys_indexable setting.	String	yes	no
node_keys_indexable	A list of property names (comma separated) that will be indexed by default. This applies to <i>nodes</i> only.	String	yes	no
online_backup_enabled	Enable support for running online backups	String	yes	no
online_backup_server	Listening server for online backups	String	yes	no
read_only	Only allow read operations from this Neo4j instance. This mode still requires write access to the directory for lock purposes.	String	yes	no
rebuild_idgenerators_fast	Use a quick approach for rebuilding the ID generators. This give quicker recovery time, but will limit the ability to reuse the space of deleted entities.	String	yes	no
relationship_auto_indexing	Controls the auto indexing feature for relationships. Setting it to false shuts it down, while true enables it	String	yes	no

Name	Description	Type	Read	Write
	by default for properties listed in the relationship_keys_indexable setting.			
relationship_grab_size	How many relationships to read at a time during iteration	String	yes	no
relationship_keys_indexable	A list of property names (comma separated) that will be indexed by default. This applies to <i>relationships</i> only.	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.internal_log.debug_contexts	Internal log contexts that should output debug level logging	String	yes	no
store.internal_log.level	Log level threshold.	String	yes	no
store.internal_log.location	The location of the internal diagnostics log.	String	yes	no
store.internal_log.max_archives	Maximum number of history files for the internal log.	String	yes	no
store.internal_log.rotation_threshold	Minimum time (in seconds) after last rotation of the internal log before it may be rotated again.	String	yes	no
store.interval.log.rotation	Maximum time interval for log rotation to wait for active transaction completion	String	yes	no
store_dir	The directory where the database files are located.	String	yes	no
string_block_size	Specifies the block size for storing strings. This parameter is only honored when the store is created, otherwise it is ignored. Note that each character in a string occupies two bytes, meaning that a block size of 120 (the default size) will hold a 60 character long string before overflowing into a second block. Also note that each block carries an overhead of 8 bytes. This means that if the block size is 120, the size of the stored records will be 128 bytes.	String	yes	no
transaction_start_timeout	The maximum amount of time to wait for the database to become available, when starting a new transaction.	String	yes	no

Name	Description	Type	Read	Write
use_memory_mapped_buffers	Use memory mapped buffers for accessing the native storage layer.	String	yes	no

MBean Diagnostics (org.neo4j.management.Diagnostics) Attributes

Name	Description	Type	Read	Write
<i>Diagnostics provided by Neo4j</i>				
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
<i>Information about an instance participating in a HA cluster</i>				
Alive	Whether this instance is alive or not	boolean	yes	no
Available	Whether this instance is available or not	boolean	yes	no
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
<i>Information about the Neo4j kernel</i>				
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

Name	Description	Type	Read	Write
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
<i>Information about the Neo4j lock status</i>				
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 that Neo4j has detected and averted (by throwing DeadlockDetectedException).	long	yes	no

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
<i>The status of Neo4j memory mapping</i>				
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
<i>Information about the Neo4j page cache</i>				
BytesRead	Number of bytes read from durable storage	long	yes	no
BytesWritten	Number of bytes written to durable storage	long	yes	no

Name	Description	Type	Read	Write
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
<i>Estimates of the numbers of different kinds of Neo4j primitives</i>				
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
NumberOfRelationshipTypeIdsInUse	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
<i>Information about the sizes of the different parts of the Neo4j graph store</i>				
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
<i>Information about the Neo4j transaction manager</i>				

Name	Description	Type	Read	Write
LastCommittedTxId	The id of the latest committed transaction	long	yes	no
NumberOfCommittedTransactions	The total number of committed transactions	long	yes	no
NumberOfOpenedTransactions	The total number started transactions	long	yes	no
NumberOfOpenTransactions	The number of currently open transactions	long	yes	no
NumberOfRolledBackTransactions	The total number of rolled back transactions	long	yes	no
PeakNumberOfConcurrentTransactions	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

28.5. Metrics Reporting



Note

Metrics reporting is only available in the Neo4j Enterprise Edition.

Introducing Metrics

Neo4j Enterprise can be configured to continuously export Neo4j-specific metrics to Graphite, Ganglia, or CSV files. This makes it easy to monitor the health of running Neo4j instances.

Neo4j Enterprise can expose metrics for the following parts of the database, and does so by default:

```
// default setting for enabling all supported metrics
metrics.enabled=true

// default setting for enabling all Neo4j specific metrics
metrics.neo4j.enabled=true

// setting for exposing metrics about transactions; number of transactions started, committed, etc.
metrics.neo4j.tx.enabled=true

// setting for exposing metrics about the Neo4j page cache; page faults, evictions, flushes and exceptions, etc.
metrics.neo4j.pagecache.enabled=true

// setting for exposing metrics about approximately entities are in the database; nodes, relationships, properties, etc.
metrics.neo4j.counts.enabled=true

// setting for exposing metrics about the network usage of the HA cluster component
metrics.neo4j.network.enabled=true
```

Graphite Configuration

For Graphite integration add the following settings to *neo4j.properties*:

```
metrics.graphite.enabled=true // default is 'false'
metrics.graphite.server=<ip>:2003
metrics.graphite.interval=<how often to send data, defaults to 3s>
metrics.prefix=<Neo4j instance name, e.g. wwwneo1>
```

Start the Neo4j Server and connect to Graphite via a web browser in order to monitor your Neo4j metrics.

Ganglia Configuration

For Ganglia integration add the following settings to *neo4j.properties*:

```
metrics.ganglia.enabled=true // default is 'false'
metrics.ganglia.server=<ip>:8469
metrics.ganglia.interval=<how often to send data, defaults to 3s>
metrics.prefix=<Neo4j instance name, e.g. wwwneo1>
```

Export to CSV Configuration

For storing metrics in local CSV files add the following settings to *neo4j.properties*:

```
metrics.csv.enabled=true // default is 'false'
metrics.csv.path=<file or directory path, defaults to "metrics/" in the store directory>
metrics.csv.file=<single/split, if split then each metric gets its own file in given directory>
metrics.csv.interval=<how often to store data, defaults to 3s>
```



Note

The CSV exporter does not automatically rotate the output files, so it is recommended to also set up a CRON job to periodically archive the files.

Configuration Settings Reference for Metrics

Metrics settings

- `metrics.csv.enabled`: Set to `true` to enable exporting metrics to CSV files.
- `metrics.csv.file`: Write to a single CSV file or to multiple files.
- `metrics.csv.interval`: The reporting interval for the CSV files.
- `metrics.csv.path`: The target location of the CSV files.
- `metrics.enabled`: The default enablement value for all the supported metrics.
- `metrics.ganglia.enabled`: Set to `true` to enable exporting metrics to Ganglia.
- `metrics.ganglia.interval`: The reporting interval for Ganglia.
- `metrics.ganglia.server`: The hostname or IP address of the Ganglia server.
- `metrics.graphite.enabled`: Set to `true` to enable exporting metrics to Graphite.
- `metrics.graphite.interval`: The reporting interval for Graphite.
- `metrics.graphite.server`: The hostname or IP address of the Graphite server.
- `metrics.jvm.buffers.enabled`: Enable reporting metrics about the buffer pools.
- `metrics.jvm.gc.enabled`: Enable reporting metrics about the duration of garbage collections.
- `metrics.jvm.memory.enabled`: Enable reporting metrics about the memory usage.
- `metrics.jvm.threads.enabled`: Enable reporting metrics about the current number of threads running.
- `metrics.neo4j.checkpointing.enabled`: Enable reporting metrics about Neo4j check pointing.
- `metrics.neo4j.cluster.enabled`: Enable reporting metrics about HA cluster info.
- `metrics.neo4j.counts.enabled`: Enable reporting metrics about approximately how many entities are in the database.
- `metrics.neo4j.enabled`: The default enablement value for all Neo4j specific support metrics.
- `metrics.neo4j.logrotation.enabled`: Enable reporting metrics about the Neo4j log rotation.
- `metrics.neo4j.network.enabled`: Enable reporting metrics about the network usage.
- `metrics.neo4j.pagecache.enabled`: Enable reporting metrics about the Neo4j page cache.
- `metrics.neo4j.tx.enabled`: Enable reporting metrics about transactions.
- `metrics.prefix`: A common prefix for the reported metrics field names.

`metrics.csv.enabled`

Description	Set to <code>true</code> to enable exporting metrics to CSV files.
Valid values	<code>metrics.csv.enabled</code> is a boolean.
Default value	<code>false</code>

`metrics.csv.file`

Description	Write to a single CSV file or to multiple files. Set to <code>single</code> (the default) for reporting the metrics in a single CSV file (given by), with a column per metrics field. Or set to <code>split</code> to produce a CSV file for each metrics field, in a directory given by .
Valid values	<code>metrics.csv.file</code> is one of <code>single</code> , <code>split</code> .
Default value	<code>single</code>

`metrics.csv.interval`

Description	The reporting interval for the CSV files. That is, how often new rows with numbers are appended to the CSV files.
Valid values	<code>metrics.csv.interval</code> is a duration (valid units are <code>ms</code> , <code>s</code> , <code>m</code>).
Default value	<code>3s</code>

`metrics.csv.path`

Description	The target location of the CSV files. Depending on the setting, this is either the path to an individual CSV file, that have each of the reported metrics fields as
--------------------	---

	columns, or it is a path to a directory wherein a CSV file per reported field will be written. Relative paths will be interpreted relative to the configured Neo4j store directory.
Valid values	metrics.csv.path is a path which is relative to store_dir.
Default value	metrics.csv
metrics.enabled	
Description	The default enablement value for all the supported metrics. Set this to false to turn off all metrics by default. The individual settings can then be used to selectively re-enable specific metrics.
Valid values	metrics.enabled is a boolean.
Default value	true
metrics.ganglia.enabled	
Description	Set to true to enable exporting metrics to Ganglia.
Valid values	metrics.ganglia.enabled is a boolean.
Default value	false
metrics.ganglia.interval	
Description	The reporting interval for Ganglia. That is, how often to send updated metrics to Ganglia.
Valid values	metrics.ganglia.interval is a duration (valid units are ms, s, m).
Default value	3s
metrics.ganglia.server	
Description	The hostname or IP address of the Ganglia server.
Valid values	metrics.ganglia.server is a hostname and port.
Default value	:8469
metrics.graphite.enabled	
Description	Set to true to enable exporting metrics to Graphite.
Valid values	metrics.graphite.enabled is a boolean.
Default value	false
metrics.graphite.interval	
Description	The reporting interval for Graphite. That is, how often to send updated metrics to Graphite.
Valid values	metrics.graphite.interval is a duration (valid units are ms, s, m).
Default value	3s
metrics.graphite.server	
Description	The hostname or IP address of the Graphite server.
Valid values	metrics.graphite.server is a hostname and port.
Default value	:2003
metrics.jvm.buffers.enabled	
Description	Enable reporting metrics about the buffer pools.
Valid values	metrics.jvm.buffers.enabled is a boolean.

Default value	true
metrics.jvm.gc.enabled	
Description	Enable reporting metrics about the duration of garbage collections.
Valid values	metrics.jvm.gc.enabled is a boolean.
Default value	true
metrics.jvm.memory.enabled	
Description	Enable reporting metrics about the memory usage.
Valid values	metrics.jvm.memory.enabled is a boolean.
Default value	true
metrics.jvm.threads.enabled	
Description	Enable reporting metrics about the current number of threads running.
Valid values	metrics.jvm.threads.enabled is a boolean.
Default value	true
metrics.neo4j.checkpointing.enabled	
Description	Enable reporting metrics about Neo4j check pointing; when it occurs and how much time it takes to complete.
Valid values	metrics.neo4j.checkpointing.enabled is a boolean.
Default value	true
metrics.neo4j.cluster.enabled	
Description	Enable reporting metrics about HA cluster info.
Valid values	metrics.neo4j.cluster.enabled is a boolean.
Default value	true
metrics.neo4j.counts.enabled	
Description	Enable reporting metrics about approximately how many entities are in the database; nodes, relationships, properties, etc.
Valid values	metrics.neo4j.counts.enabled is a boolean.
Default value	true
metrics.neo4j.enabled	
Description	The default enablement value for all Neo4j specific support metrics. Set this to false to turn off all Neo4j specific metrics by default. The individual metrics.neo4j.* metrics can then be turned on selectively.
Valid values	metrics.neo4j.enabled is a boolean.
Default value	true
metrics.neo4j.logrotation.enabled	
Description	Enable reporting metrics about the Neo4j log rotation; when it occurs and how much time it takes to complete.
Valid values	metrics.neo4j.logrotation.enabled is a boolean.
Default value	true
metrics.neo4j.network.enabled	
Description	Enable reporting metrics about the network usage.

Valid values	metrics.neo4j.network.enabled is a boolean.
Default value	true
metrics.neo4j.pagecache.enabled	
Description	Enable reporting metrics about the Neo4j page cache; page faults, evictions, flushes, exceptions, etc.
Valid values	metrics.neo4j.pagecache.enabled is a boolean.
Default value	true
metrics.neo4j.tx.enabled	
Description	Enable reporting metrics about transactions; number of transactions started, committed, etc.
Valid values	metrics.neo4j.tx.enabled is a boolean.
Default value	true
metrics.prefix	
Description	A common prefix for the reported metrics field names. By default, this is either be <i>neo4j</i> , or a computed value based on the cluster and instance names, when running in an HA configuration.
Valid values	metrics.prefix is a string.
Default value	neo4j

Part VI. Tools

The Tools part describes available Neo4j tools and how to use them.

29. Import tool	531
29.1. CSV file header format	532
29.2. Command line usage	533
29.3. Import tool examples	535
30. Web Interface	543
31. Neo4j Shell	544
31.1. Starting the shell	545
31.2. Passing options and arguments	547
31.3. Enum options	548
31.4. Filters	549
31.5. Node titles	550
31.6. How to use (individual commands)	551
31.7. An example shell session	556
31.8. A Matrix example	557

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” \[9\]](#)).



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” \[184\]](#).
- If you want to bulk import into an existing database see [Chapter 36, *Batch Insertion* \[634\]](#).

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

Linux

Under Unix/Linux/OSX, the command is named `neo4j-import`. Depending on the installation type, the tool is either available globally, or used by executing `./bin/neo4j-import` from inside the installation directory.

Windows

For help with running the import tool using Windows PowerShell, see [Section 23.3, "Windows PowerShell module"](#) [443].

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 <i>TAB</i> , between values in CSV data. The default option is ,.
--array-delimiter <array-delimiter-character>	Delimiter character, or <i>TAB</i> , 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.'
--multiline-fields <true/false>	Whether or not fields from input source can span multiple lines, i.e. contain newline characters. Default value: false
--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 <code>Charset#availableCharsets()</code> . If no input encoding is provided, the default character set of the JVM will be used.
--ignore-empty-strings <true/false>	Whether or not empty string fields, i.e. "" from input source are ignored, i.e. treated as null. Default value: false
--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 <true/false>	Enable printing of error stack traces.
--bad-tolerance <max number of bad entries>	Number of bad entries before the import is considered failed. This tolerance threshold is about relationships referring to missing nodes. Format errors in input data are still treated as errors. Default value: 1000
--skip-bad-relationships <true/false>	Whether or not to skip importing relationships that refer to missing node ids, i.e. either start or end node id/group referring to node that wasn't specified by the node input data. Skipped nodes will be logged, containing at most number of entities specified by bad-tolerance. Default value: true
--skip-duplicate-nodes <true/false>	Whether or not to skip importing nodes that have the same id/group. In the event of multiple nodes within the same group having the same id, the first encountered will be imported whereas consecutive such nodes will be skipped. Skipped nodes will be logged, containing at most number of entities specified by bad-tolerance. Default value: false
--ignore-extra-columns <true/false>	Whether or not to ignore extra columns in the data not specified by the header. Skipped columns will be logged, containing at most number of entities specified by bad-tolerance. Default value: false

Output and statistics

While an import is running through its different stages, some statistics and figures are printed in the console. The general interpretation of that output is to look at the horizontal line, which is divided up into sections, each section representing one type of work going on in parallel with the other sections. The wider a section is, the more time is spent there relative to the other sections, the widest being the bottleneck, also marked with *. If a section has a double line, instead of just a single line, it means that multiple threads are executing the work in that section. To the far right a number is displayed telling how many entities (nodes or relationships) have been processed by that stage.

As an example:

```
[*>:20,25 MB/s-----|PREPARE(3)=====|RELATIONSHIP(2)=====] 16M
```

Would be interpreted as:

- > data being read, and perhaps parsed, at 20,25 MB/s, data that is being passed on to ...
- PREPARE preparing the data for ...
- RELATIONSHIP creating actual relationship records and ...
- v writing the relationships to the store. This step isn't visible in this example, because it's so cheap compared to the other sections.

Observing the section sizes can give hints about where performance can be improved. In the example above, the bottleneck is the data read section (marked with >), which might indicate that the disk is being slow, or is poorly handling simultaneous read and write operations (since the last section often revolves around writing to disk).

Verbose error information

In some cases if an unexpected error occurs it might be useful to supply the command line option --stacktrace to the import (and rerun the import to actually see the additional information). This will have the error printed with additional debug information, useful for both developers and issue reporting.

29.3. Import tool examples

Let's look at a few examples. 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.

Basic example

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" \[440\]](#))

Once we've got the database up and running we can add appropriate indexes. (see [Section 3.6, "Labels, Constraints and Indexes" \[32\]](#).)

**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" \[533\]](#)) 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. Files within such an input group can be specified with multiple match strings, delimited by , , where each match string can be either: *the exact file name* or *a regular expression matching one or more files*. Multiple matching files will be sorted according to their characters *and* their natural number sort order for file names containing numbers.

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"
```

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.

In this case, 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” \[532\]](#))

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  
keanu,"Keanu Reeves",Actor  
laurence,"Laurence Fishburne",Actor  
carrieanne,"Carrie-Anne Moss",Actor  
laurence,"Laurence Harvey",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
```

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
```

Bad input data

The import tool has a threshold of how many bad entities (nodes/relationships) to tolerate and skip before failing the import. By default 1000 bad entities are tolerated. A bad tolerance of 0 will as an example fail the import on the first bad entity. For more information, see the [--bad-tolerance](#) option.

There are different types of bad input, which we will look into.

Relationships referring to missing nodes

Relationships that refer to missing node ids, either for :START_ID or :END_ID are considered bad relationships. Whether or not such relationships are skipped is controlled with [--skip-bad-relationships](#) flag which can have the values true or false or no value, which means true. Specifying false means that any bad relationship is considered an error and will fail the import. For more information, see the [--skip-bad-relationships](#) option.

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
```

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 bad relationships.

not-imported.bad

```
InputRelationship:
```

```
source: roles9.csv:11
properties: [role, Emil]
startNode: emil
endNode: tt0133093
type: ACTED_IN
refering to missing node emil
```

Multiple nodes with same id within same id space

Nodes that specify `:ID` which has already been specified within the id space are considered bad nodes. Whether or not such nodes are skipped is controlled with `--skip-duplicate-nodes` flag which can have the values `true` or `false` or no value, which means `true`. Specifying `false` means that any duplicate node is considered an error and will fail the import. For more information, see the [--skip-duplicate-nodes](#) option.

In the following example there is a node id that is specified twice within the same id space.

actors10.csv

```
personId:ID,name,:LABEL
keanu,"Keanu Reeves",Actor
laurence,"Laurence Fishburne",Actor
carrieanne,"Carrie-Anne Moss",Actor
laurence,"Laurence Harvey",Actor
```

```
neo4j-import --into path_to_target_directory --nodes actors10.csv --skip-duplicate-nodes
```

Since there was only one bad node the import process will complete successfully and a `not-imported.bad` file will be created and populated with the bad node.

not-imported.bad

```
Id 'laurence' is defined more than once in global id space, at least at actors10.csv:3 and actors10.csv:5
```

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
```

For help with running shell using Windows PowerShell, see [Section 23.3, “Windows PowerShell module” \[443\]](#).

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” \[546\]](#) for local databases and see [the section called “Enabling the shell server” \[545\]](#) 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 24.8, “Configuration Settings Reference” \[466\]](#). 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	<code>remote_shell_enabled</code> is a boolean.
Default value	<code>false</code>

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 <code>0.0.0.0</code> for all interfaces.
Valid values	<code>remote_shell_host</code> is a string which must be a valid name.
Default value	<code>127.0.0.1</code>

remote_shell_name

Description	The name of the shell.
Valid values	<code>remote_shell_name</code> is a string which must be a valid name.
Default value	<code>shell</code>

remote_shell_port

Description	The port the shell will listen on.
Valid values	<code>remote_shell_port</code> is an integer which must be a valid port number (is in the range <code>0</code> to <code>65535</code>).
Default value	<code>1337</code>

remote_shell_read_only

Description	Read only mode. Will only allow read operations.
Valid values	<code>remote_shell_read_only</code> 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" \[552\]](#)) 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” \[549\]](#) and for information about how to specify direction see [Section 31.3, “Enum options” \[548\]](#). 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 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

² <http://groovy.codehaus.org>

³ <http://www.python.org>

⁴ <http://www.jython.org>

with the `-f` option, see [Section 31.4, “Filters” \[549\]](#). See [Section 31.3, “Enum options” \[548\]](#) 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” \[553\]](#).

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 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
9 ms

# create one labeled node and a relationship
neo4j-sh (?)$ create (:Person:Hacker {name:'Mattias'}), (:Person)-[:KNOWS]->(:Person);
+-----+
| No data returned. |
+-----+
Nodes created: 1
Relationships created: 1
Properties set: 1
Labels added: 2
19 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
44 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'}"

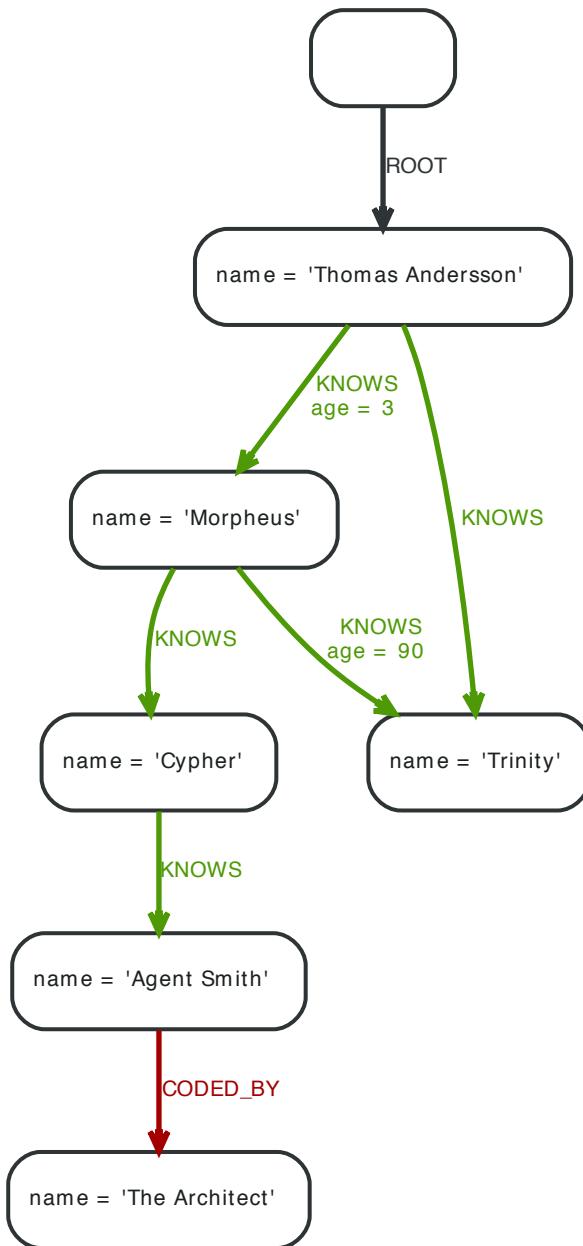
# print current history stack
neo4j-sh (1)$ pwd
Current is (1)
(Jon,0)-->(1)

# 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



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]->(4)
(me)-[:KNOWS]->(Trinity,2)
(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      |
+-----+
| "Cypher"          |
| "Trinity"         |
| "Thomas Andersson" |
+-----+
3 rows
39 ms

# Morpheus' friends, looking up Morpheus by name in the Neo4j autoindex
neo4j-sh (?)$ cypher 2.2 start morpheus = node:node_auto_index(name='Morpheus') match morpheus-[:KNOWS]-zionist return zionist.name;
+-----+
| zionist.name      |
+-----+
| "Cypher"          |
| "Trinity"         |
| "Thomas Andersson" |
+-----+
3 rows
535 ms

```

Part VII. 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](#)⁵ handy while reading!

⁵ <http://neo4j.com/docs/2.3.1/javadocs/>

32. Extending the Neo4j Server	562
32.1. Server Plugins	563
32.2. Unmanaged Extensions	567
32.3. Testing your extension	572
32.4. Installing Plugins and Extensions in Neo4j Desktop	574
33. Using Neo4j embedded in Java applications	575
33.1. Include Neo4j in your project	576
33.2. Hello World	581
33.3. Property values	584
33.4. User database with indexes	585
33.5. User database with legacy index	587
33.6. Managing resources when using long running transactions	588
33.7. Basic unit testing	589
33.8. Traversal	591
33.9. Domain entities	599
33.10. Graph Algorithm examples	600
33.11. Reading a management attribute	602
33.12. How to create unique nodes	603
33.13. Terminating a running transaction	605
33.14. Execute Cypher Queries from Java	607
33.15. Query Parameters	609
34. The Traversal Framework	611
34.1. Main concepts	612
34.2. Traversal Framework Java API	613
35. Legacy Indexing	619
35.1. Introduction	620
35.2. Create	621
35.3. Delete	622
35.4. Add	623
35.5. Remove	625
35.6. Update	626
35.7. Search	627
35.8. Relationship indexes	629
35.9. Scores	630
35.10. Configuration and fulltext indexes	631
35.11. Extra features for Lucene indexes	632
36. Batch Insertion	634
36.1. Batch Inserter Examples	635
36.2. Index Batch Insertion	637

Chapter 32. Extending the Neo4j Server

The Neo4j Server can be extended by either plugins or unmanaged extensions.

32.1. 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>2.3.1</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 )
    public Iterable<Node> getAllNodes( @Source GraphDatabaseService graphDb )
```

¹ <http://search.maven.org/#search|gav|1|g%3A%22org.neo4j%22%20AND%20a%3A%22server-api%22>

² <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/server/plugins/ServerPlugin.html>

```

{
    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.java³](#)

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] ) );
            }
            expander = expanderBuilder.build();
        }
    }
}

```

³ <https://github.com/neo4j/neo4j/blob/2.3.1/community/server-examples/src/main/java/org/neo4j/examples/server/plugins/GetAll.java>

```

        }
        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).



Caution

Whilst Neo4j is tested to run on JVM 8, Neo4j server is currently compiled using JDK 7, to ensure compatibility for JVM 7 deployments. When compiling plugins for Neo4j Server, we strongly recommend using JDK 7 also.



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",
```

⁴ <https://github.com/neo4j/neo4j/blob/2.3.1/community/server-examples/src/main/java/org/neo4j/examples/server/plugins/ShortestPath.java>

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

Performing a GET on one of the two extension URLs 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).

32.2. 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-RS⁵ 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 [Preface \[v\]](#)).

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>2.3.1</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
```



Caution

Whilst Neo4j is tested to run on JVM 8, Neo4j server is currently compiled using JDK 7, to ensure compatibility for JVM 7 deployments. When compiling plugins for Neo4j Server, we strongly recommend using JDK 7 also.

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” \[300\]](#)) 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();
                try ( Transaction tx = graphDb.beginTx() )
                {
                    StreamableIterable<Relationship> relationships = tx.getRelationships( Direction.INBOUND_OF, personName, ACTED_IN );
                    for ( Relationship relationship : relationships )
                    {
                        jg.writeStartObject();
                        jg.writeFieldName("name");
                        jg.writeString( relationship.getEndNode().getProperty("name").asString() );
                        jg.writeEndObject();
                    }
                }
                jg.writeEndArray();
                jg.writeEndObject();
            }
        };
        return Response.ok( stream ).build();
    }
}
```

⁶ <https://github.com/neo4j/neo4j/blob/2.3.1/community/server-examples/src/main/java/org/neo4j/examples/server/unmanaged/HelloWorldResource.java>

```

        jg.writeStartObject();
        jg.writeFieldName( "colleagues" );
        jg.writeStartArray();

        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.

⁷ <https://github.com/neo4j/neo4j/blob/2.3.1/community/server-examples/src/main/java/org/neo4j/examples/server/unmanaged/ColleaguesResource.java>

**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.

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";
}
```

⁸ <https://github.com/neo4j/neo4j/blob/2.3.1/community/server-examples/src/main/java/org/neo4j/examples/server/unmanaged/ColleaguesCypherExecutionResource.java>

The full source code is found here: [ColleaguesCypherExecutionResource.java](#)⁸

Our `findColleagues` method will now respond to GET requests at the URL: `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"]}
```

32.3. 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>2.3.1</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() );
    }
}

@Test
public void testMyExtensionWithFunctionFixture() throws Exception
{
    // Given
    try ( ServerControls server = TestServerBuilders.newInProcessBuilder()
        .withExtension( "/myExtension", MyUnmanagedExtension.class )
        .withFixture( new Function<GraphDatabaseService, Void>()
    {
        @Override
        public Void apply( GraphDatabaseService graphDatabaseService ) throws RuntimeException
        {
            try ( Transaction tx = graphDatabaseService.beginTx() )
            {
                graphDatabaseService.createNode( DynamicLabel.label( "User" ) );
                tx.success();
            }
            return null;
        }
    } )
        .newServer() )
    {
```

```
{
    // When
    Result result = server.graph().execute( "MATCH (n:User) return n" );

    // Then
    assertEquals( 1, IteratorUtil.count( result ) );
}
}
```

The full source code of the example is found here: [ExtensionTestingDocTest.java](#)⁹

Note the use of `server.httpURI().resolve("myExtension")` to ensure that the correct base URI is used.

If you are using the JUnit test framework, there is a JUnit rule available as well.

JUnit example

```
@Rule
public Neo4jRule neo4j = new Neo4jRule()
    .withFixture( "CREATE (admin:Admin)" )
    .withFixture( new Function<GraphDatabaseService, Void>()
{
    @Override
    public Void apply( GraphDatabaseService graphDatabaseService ) throws RuntimeException
    {
        try (Transaction tx = graphDatabaseService.beginTx())
        {
            graphDatabaseService.createNode( DynamicLabel.label( "Admin" ) );
            tx.success();
        }
        return null;
    }
} );

@Test
public void shouldWorkWithServer() throws Exception
{
    // Given
    URI serverURI = neo4j.httpURI();

    // When I access the server
    HTTP.Response response = HTTP.GET( serverURI.toString() );

    // Then it should reply
    assertEquals(200, response.status());

    // and we have access to underlying GraphDatabaseService
    try (Transaction tx = neo4j.getGraphDatabaseService().beginTx()) {
        assertEquals( 2, IteratorUtil.count(
            neo4j.getGraphDatabaseService().findNodes( DynamicLabel.label( "Admin" ) )
        ));
        tx.success();
    }
}
```

⁹ <https://github.com/neo4j/neo4j/blob/2.3.1/community/neo4j-harness/src/test/java/org/neo4j/harness/doc/ExtensionTestingDocTest.java>

32.4. 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 33. 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.

33.1. 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 <code>-classpath</code>
Eclipse	<ul style="list-style-type: none"> • Right-click on the project and then go <i>Build Path</i> → <i>Configure Build Path</i>. In the dialog, choose <i>Add External JARs</i>, browse to the Neo4j <i>lib/</i> directory and select all of the jar files. • Another option is to use User Libraries³.
IntelliJ IDEA	See Libraries, Global Libraries, and the Configure Library dialog ⁴
NetBeans	<ul style="list-style-type: none"> • Right-click on the <i>Libraries</i> node of the project, choose <i>Add JAR/Folder</i>, browse to the Neo4j <i>lib/</i> directory and select all of the jar files. • You can also handle libraries from the project node, see Managing a Project's Classpath⁵.

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



Note

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

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>

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>2.3.1</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:

```
...
<dependencies>
  ...
  <dependency org="org.neo4j" name="neo4j" rev="2.3.1"/>
```

⁸ <http://www.neo4j.org/learn/licensing>

⁹ <http://neo4j.com/docs/2.3.1/javadocs/>

¹⁰ <http://www.eclipse.org>

¹¹ <http://www.eclipse.org/m2e/>

```
..  
</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 = "2.3.1"  
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 open 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()  
.newEmbeddedDatabaseBuilder( testDirectory.graphDbDir() )  
.loadPropertiesFromFile( pathToConfig + "neo4j.properties" )  
.newGraphDatabase();
```

Configuration settings can also be applied programmatically, like so:

¹² <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/GraphDatabaseService.html>

```
GraphDatabaseService graphDb = new GraphDatabaseFactory()
    .newEmbeddedDatabaseBuilder( testDirectory.graphDbDir() )
    .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 \[450\]](#).

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.

Controlling Logging

Neo4j provides logging via its own `org.neo4j.logging.Log`¹³ layer, and does not natively use any existing Java logging framework. All logging events produced by Neo4j have a name, a level and a message. The name is a FQCN (fully qualified class name).

Neo4j uses the following log levels:

ERROR	For serious errors that are almost always fatal
WARN	For events that are serious, but not fatal
INFO	Informational events
DEBUG	Debugging events

To enable logging, an implementation of `org.neo4j.logging.LogProvider`¹⁴ must be provided to the `GraphDatabaseFactory`¹⁵, as follows:

```
LogProvider logProvider = new MyCustomLogProvider( output );
graphDb = new GraphDatabaseFactory().setUserLogProvider( logProvider ).newEmbeddedDatabase( DB_PATH );
```

Neo4j also includes a binding for SLF4J, which is available in the neo4j-slf4j library jar. This can be obtained via maven:

```
<project>
...
<dependencies>
    <dependency>
        <groupId>org.neo4j</groupId>
        <artifactId>neo4j-slf4j</artifactId>
        <version>2.3.1</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
    </dependency>
```

¹³ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/logging/Log.html>

¹⁴ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/logging/LogProvider.html>

¹⁵ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/GraphDatabaseFactory.html>

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

To use this binding, simply pass an instance of `org.neo4j.logging.slf4j.Slf4jLogProvider`¹⁶ to the `GraphDatabaseFactory`¹⁷, as follows:

```
graphDb = new GraphDatabaseFactory().setUserLogProvider( new Slf4jLogProvider() ).newEmbeddedDatabase( DB_PATH );
```

All log output can then be controlled via SLF4J configuration.

¹⁶ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/logging/sl4j/Slf4jLogProvider.html>

¹⁷ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/GraphDatabaseFactory.html>

33.2. Hello World

Learn how to create and access nodes and relationships. For information on project setup, see [Section 33.1, “Include Neo4j in your project” \[576\]](#).

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](#)¹⁸

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:

```
try ( Transaction tx = graphDb.beginTx() )
```

¹⁸ <https://github.com/neo4j/neo4j/blob/2.3.1/community/embedded-examples/src/main/java/org/neo4j/examples/EmbeddedNeo4j.java>

```
{
    // Database operations go here
    tx.success();
}
```

For more information on transactions, see [Chapter 18, Transaction Management \[287\]](#) and [Java API for Transaction](#)¹⁹.



Note

For brevity, we do not spell out wrapping of operations in a transaction throughout the manual.

Create a small graph

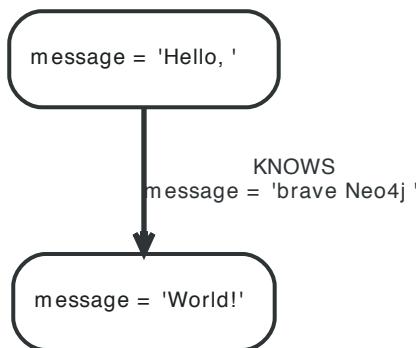
Now, let's create a few nodes. The API is very intuitive. Feel free to have a look at the [Neo4j Javadocs](#)²⁰. They're included in the distribution, as well. Here's how to create a small graph consisting of two nodes, connected with one relationship and some properties:

```
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 33.1. 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
firstNode.getSingleRelationship( RelTypes.KNOWS, Direction.OUTGOING ).delete();
```

¹⁹ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/Transaction.html>

²⁰ <http://neo4j.com/docs/2.3.1/javadocs/>

```
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();
```

33.3. 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.

NULLs can instead be modeled by the absence of a key.

Property value types

Type	Description	Value range
<code>boolean</code>		true/false
<code>byte</code>	8-bit integer	-128 to 127, inclusive
<code>short</code>	16-bit integer	-32768 to 32767, inclusive
<code>int</code>	32-bit integer	-2147483648 to 2147483647, inclusive
<code>long</code>	64-bit integer	-9223372036854775808 to 9223372036854775807, inclusive
<code>float</code>	32-bit IEEE 754 floating-point number	
<code>double</code>	64-bit IEEE 754 floating-point number	
<code>char</code>	16-bit unsigned integers representing Unicode characters	u0000 to uffff (0 to 65535)
<code>String</code>	sequence of Unicode characters	

For further details on float/double values, see [Java Language Specification](#)²¹.

²¹ <http://docs.oracle.com/javase/specs/jls/se5.0/html/typesValues.html#4.2.3>

33.4. 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](https://github.com/neo4j/neo4j/blob/2.3.1/community/embedded-examples/src/main/java/org/neo4j/examples/EmbeddedNeo4jWithNewIndexing.java)²²

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 33.6, “Managing resources when using long running transactions” [588] 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" );
int idToFind = 45;
```

²² <https://github.com/neo4j/neo4j/blob/2.3.1/community/embedded-examples/src/main/java/org/neo4j/examples/EmbeddedNeo4jWithNewIndexing.java>

```

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();
}

```

33.5. User database with legacy index

Unless you have specific reasons to use the legacy indexing, see [Section 33.4, “User database with indexes” \[585\]](#) instead.



Note

Please read [Section 33.6, “Managing resources when using long running transactions” \[588\]](#) 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](#)²³

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 ) );
```

²³ <https://github.com/neo4j/neo4j/blob/2.3.1/community/embedded-examples/src/main/java/org/neo4j/examples/EmbeddedNeo4jWithIndexing.java>

33.6. 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`²⁴'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();
}
```

²⁴ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/ResourceIterator.html>

33.7. 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-kernel²⁵](#) and [org.neo4j:neo4j-io²⁶](#).

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>2.3.1</version>
    <type>test-jar</type>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-io</artifactId>
    <version>2.3.1</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.java²⁷](#)

Before each test, create a fresh database:

```
@Before
```

²⁵ <http://search.maven.org/#search|ga|1|g%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-kernel%22>

²⁶ <http://search.maven.org/#search|ga|1|g%3A%22org.neo4j%22%20AND%20a%3A%22neo4j-io%22>

²⁷ <https://github.com/neo4j/neo4j/blob/2.3.1/community/embedded-examples/src/test/java/org/neo4j/examples/Neo4jBasicDocTest.java>

```
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();
```

33.8. Traversal

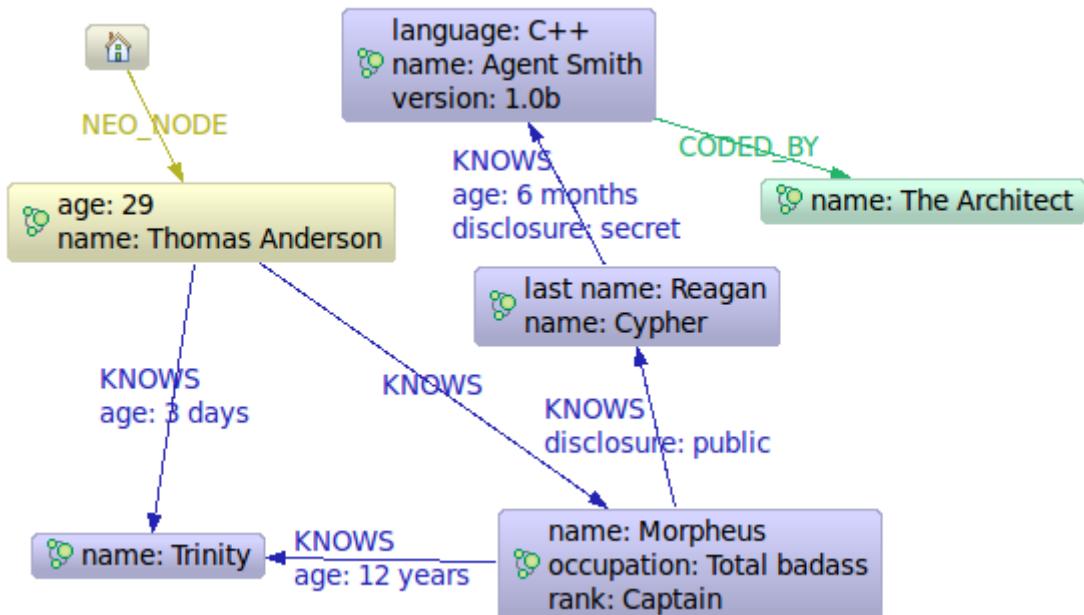
For reading about traversals, see [Chapter 34, The Traversal Framework \[611\]](#).

For more examples of traversals, see [Chapter 5, Basic Data Modeling Examples \[47\]](#).

The Matrix

This is the first graph we want to traverse into:

Figure 33.2. Matrix node space view



Tip

The source code of this example is found here: [NewMatrix.java](#)²⁸

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()
```

²⁸ <https://github.com/neo4j/neo4j/blob/2.3.1/community/embedded-examples/src/main/java/org/neo4j/examples/NewMatrix.java>

```
.getProperty( "name" ) + "\n";
    number_of_friends++;
}
output += "Number of friends found: " + number_of_friends + "\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 number_of_hackers = 0;
Traverser traverser = findHackers( getNeoNode() );
for ( Path hackerPath : traverser )
{
    output += "At depth " + hackerPath.length() + " => "
        + hackerPath.endNode()
        .getProperty( "name" ) + "\n";
    number_of_hackers++;
}
output += "Number of hackers found: " + number_of_hackers + "\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](#)²⁹

Create a toy graph

```
Node A = db.createNode();
Node B = db.createNode();
Node C = db.createNode();
```

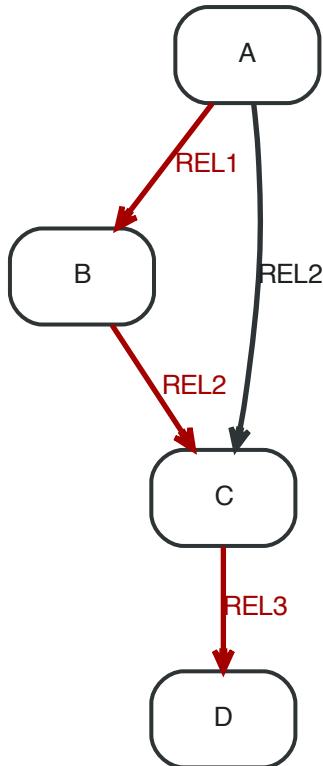
²⁹ <https://github.com/neo4j/neo4j/blob/2.3.1/community/embedded-examples/src/main/java/org/neo4j/examples/orderedpath/OrderedPath.java>

```

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`³⁰ as we want to be able to revisit the same node during 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, "" ) + ")";
    }

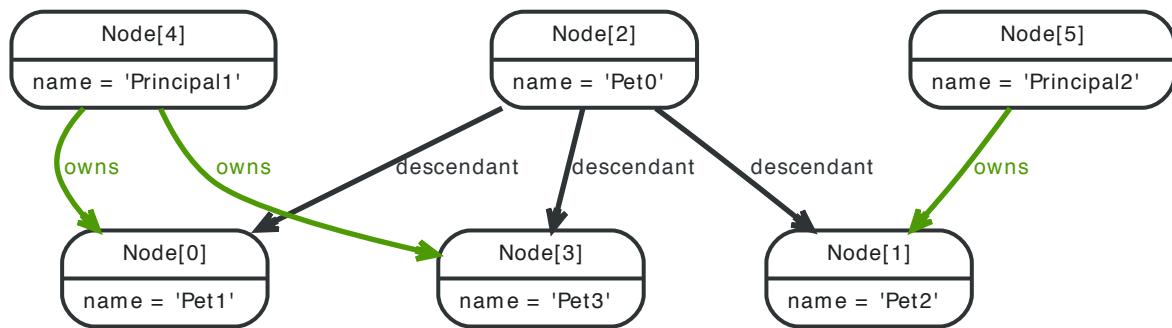
    @Override
    public String relationshipRepresentation( Path path, Node from, Relationship relationship )
    {
        String prefix = "--", suffix = "--";
        if ( from.equals( relationship.getEndNode() ) )
        {
            prefix = "<--";
        }
        else
        {
            suffix = "-->";
        }
        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.

³⁰ http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/Uniqueness.html#NODE_PATH

Figure 33.3. 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 than 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" );
TraverserDescription 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 `TraverserDescription` from the old one, having `NODE_GLOBAL` uniqueness to see the difference.



Tip

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

```
TraverserDescription 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 (known 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.

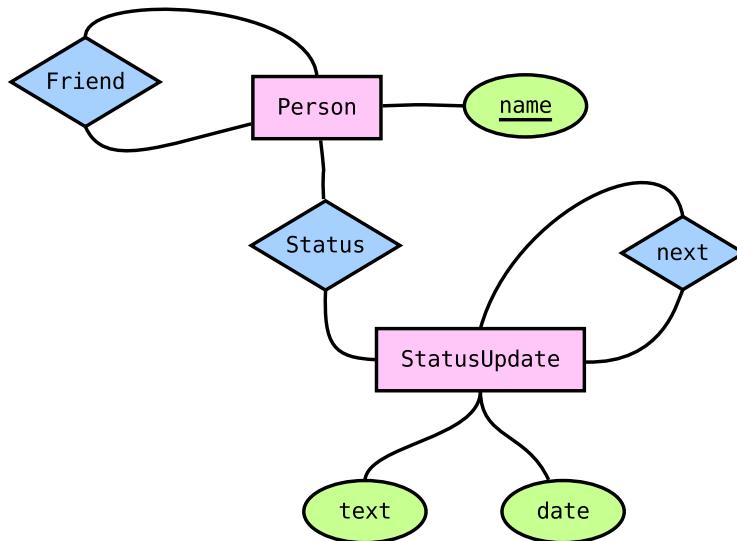


Tip

The source code of the example is found here: [socnet](#)³¹

Simple social model

Figure 33.4. 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.

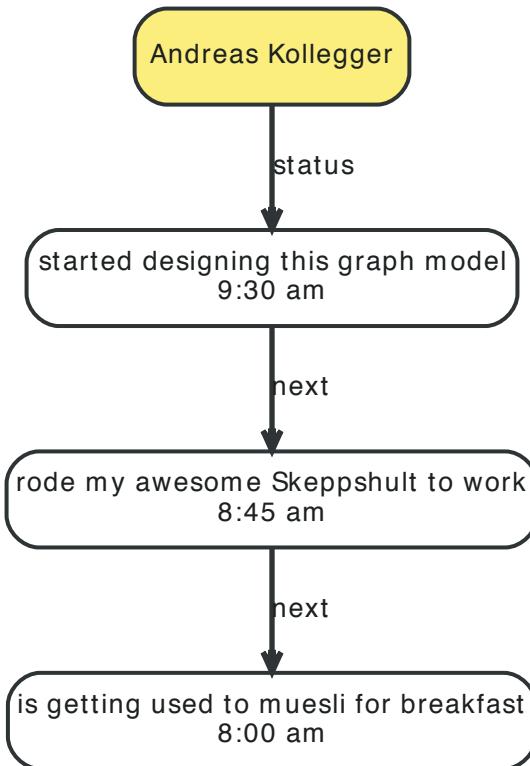
- Person
 - friend: relates two distinct Person instances (no self-reference)
 - status: connects to the most recent StatusUpdate
- StatusUpdate
 - next: points to the next StatusUpdate in the chain, which was posted before the current one

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:

³¹ <https://github.com/neo4j/neo4j/tree/2.3.1/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](#)³².

The code looks like:

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

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

³² <http://www.slideshare.net/systay/pattern-activity-stream>

```
{  
    first.next();  
    sort();  
}  
  
return returnVal;
```

33.9. 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](#)³³

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() + "]";
}
```

³³ <https://github.com/neo4j/neo4j/blob/2.3.1/community/embedded-examples/src/main/java/org/neo4j/examples/socnet/Person.java>

33.10. Graph Algorithm examples

For details on the graph algorithm usage, see the Javadocs³⁴.



Tip

The source code used in the example is found here: [PathFindingDocTest.java](#)³⁵

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 );
```

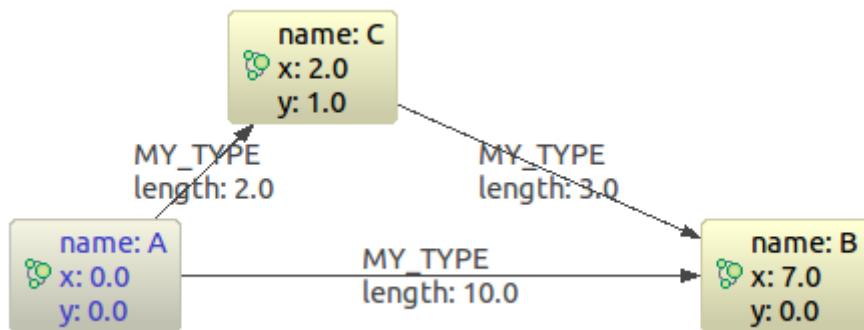
Using Dijkstra's algorithm³⁶ to calculate cheapest path between node A and B where each relationship can have a weight (i.e. cost) and the path(s) with least cost are found.

```
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*³⁷ 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:



³⁴ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphalgo/GraphAlgoFactory.html>

³⁵ <https://github.com/neo4j/neo4j/blob/2.3.1/community/embedded-examples/src/test/java/org/neo4j/examples/PathFindingDocTest.java>

³⁶ http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

³⁷ http://en.wikipedia.org/wiki/A*_search_algorithm

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

EstimateEvaluator<Double> estimateEvaluator = new EstimateEvaluator<Double>()
{
    @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 );
```

33.11. Reading a management attribute

The [JmxUtils³⁸](#) 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.java³⁹](#)

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.jmx⁴⁰](#) package.
- For the Enterprise edition, see the [org.neo4j.management⁴¹](#) package as well.

³⁸ <http://neo4j.com/docs/2.3.1/javadoc/org/neo4j/jmx/JmxUtils.html>

³⁹ <https://github.com/neo4j/neo4j/blob/2.3.1/community/embedded-examples/src/test/java/org/neo4j/examples/JmxDocTest.java>

⁴⁰ <http://neo4j.com/docs/2.3.1/javadoc/org/neo4j/jmx/package-summary.html>

⁴¹ <http://neo4j.com/docs/2.3.1/javadoc/org/neo4j/management/package-summary.html>

33.12. 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” \[295\]](#).

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” \[603\]](#) instead.

By using `put-if-absent`⁴² 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`⁴³ 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();
}
```

⁴² <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/index/Index.html#putIfAbsent%28T,%20java.lang.String,%20java.lang.Object%29>

⁴³ <http://neo4j.com/docs/2.3.1/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” \[603\]](#) 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.

33.13. 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](#)⁴⁴

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 15 in 1 sec
```

⁴⁴ <https://github.com/neo4j/neo4j/blob/2.3.1/community/embedded-examples/src/main/java/org/neo4j/examples/TerminateTransactions.java>

Finally, let's shut down the database again.

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

33.14. Execute Cypher Queries from Java



Tip

The full source code of the example: [JavaQuery.java](#)⁴⁵

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](#)⁴⁶.

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](#)⁴⁷ 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:

```
List<String> columns = result.columns();
```

⁴⁵ <https://github.com/neo4j/neo4j/blob/2.3.1/community/cypher/docs/cypher-docs/src/test/java/org/neo4j/cypher/example/JavaQuery.java>

⁴⁶ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/Result.html>

⁴⁷ <http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>

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 API⁴⁸](#).

For more information and examples for Cypher, see [Part III, “Cypher Query Language” \[102\]](#) and [Chapter 5, Basic Data Modeling Examples \[47\]](#).

⁴⁸ <http://neo4j.com/docs/2.3.1/javadocs/index.html>

33.15. Query Parameters

For more information on parameters see [Section 8.5, “Parameters” \[113\]](#).

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 34. 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 33.8, “Traversal” \[591\]](#).

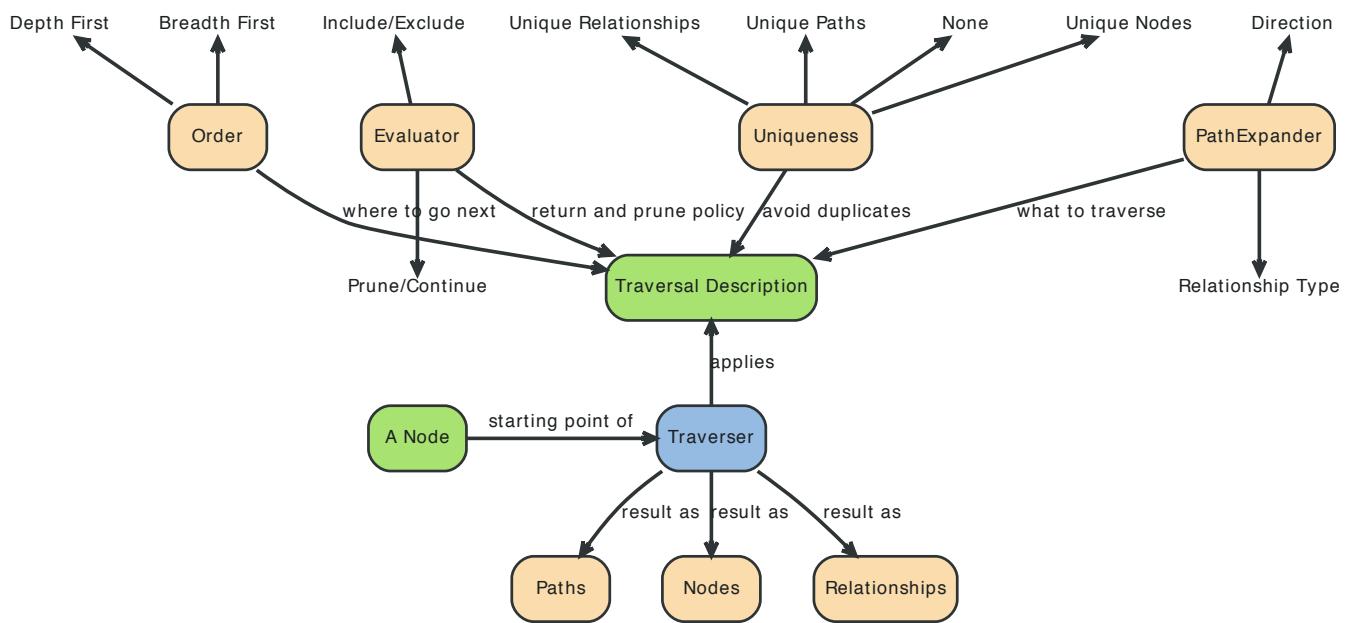
You can also use [The Cypher Query Language](#) as a powerful declarative way to query the graph.

¹ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/package-summary.html>

34.1. 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 34.2, “Traversal Framework Java API” \[613\]](#) for more details.

34.2. 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/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html>

³ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#relationships>

⁴ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#relationships>

⁵ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/Direction.html#BOTH>

⁶ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/Evaluator.html>

⁷ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/Traverser.html>

⁸ [http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#traverse\(org.neo4j.graphdb.Node\)](http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#traverse(org.neo4j.graphdb.Node))

⁹ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/Uniqueness.html>

¹⁰ http://neo4j.com/docs/2.3.1/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/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#order>

¹⁴ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/BranchOrderingPolicies.html>

¹⁵ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/BranchOrderingPolicy.html>

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

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

BranchOrderingPolicy

A factory for creating BranchSelectors to decide in what order branches are returned (where a branch's position is represented as a Path¹⁶ from the start node to the current node). Common policies are depth-first¹⁷ and breadth-first¹⁸ 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 TraversalBranch²⁰s from the current one.

Path

A Path²¹ 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 PathExpanders (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/2.3.1/javadocs/org/neo4j/graphdb/Path.html>

¹⁷ [http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#depthFirst\(\)](http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#depthFirst())

¹⁸ [http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#breadthFirst\(\)](http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#breadthFirst())

¹⁹ [http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#depthFirst\(\)](http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html#depthFirst())

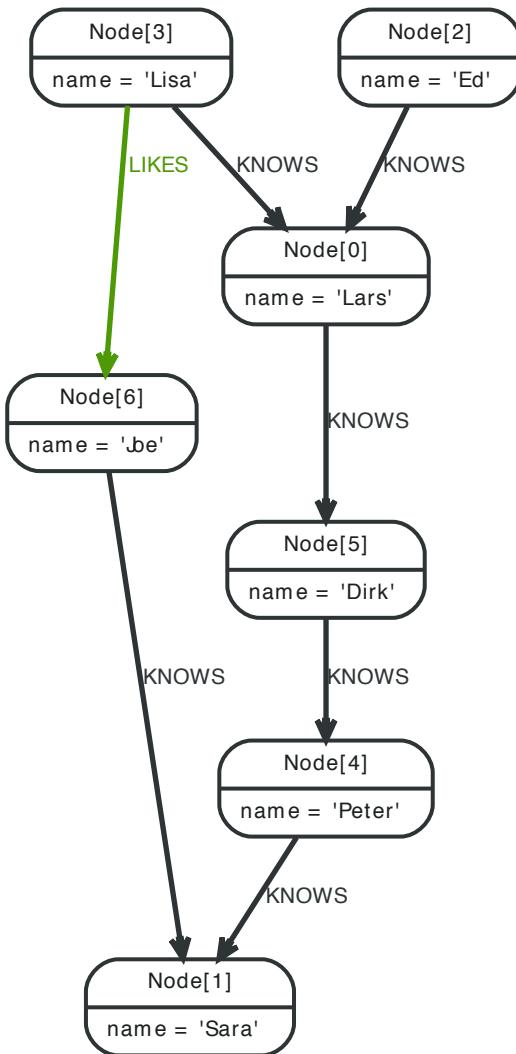
²⁰ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalBranch.html>

²¹ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/Path.html>

How to use the Traversal framework

A traversal description²² is built using a fluent interface and such a description can then spawn traversers²³.

Figure 34.1. 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/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html>

²³ <http://neo4j.com/docs/2.3.1/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,5]--(2)
(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)
```

Since [TraversalDescription](#)²⁴s 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)
```

For various useful evaluators, see the [Evaluators](#)²⁵ Java API or simply implement the [Evaluator](#)²⁶ interface yourself.

²⁴ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html>

²⁵ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/Evaluators.html>

²⁶ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/Evaluator.html>

If you're not interested in the `Path`²⁷s, but the `Node`²⁸s 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

²⁷ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/Path.html>

²⁸ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/Node.html>

²⁹ [http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/Traverser.html#nodes\(\)](http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/Traverser.html#nodes())

³⁰ [http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/Traverser.html#relationships\(\)](http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/traversal/Traverser.html#relationships())

³¹ <https://github.com/neo4j/neo4j/blob/2.3.1/community/embedded-examples/src/main/java/org/neo4j/examples/TraversalExample.java>

Chapter 35. 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. 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.

35.1. 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 36.2, “Index Batch Insertion” \[637\]](#).



Note

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

¹ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/index/package-summary.html>

² <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/Node.html>

³ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/Relationship.html>

35.2. 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 35.8, “Relationship indexes” \[629\]](#) for more information specific to relationship indexes.

See [Section 35.10, “Configuration and fulltext indexes” \[631\]](#) 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" );
```

35.3. 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/2.3.1/javadocs/org/neo4j/graphdb/index/Index.html#delete%28%29>

35.4. 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" ) );
```

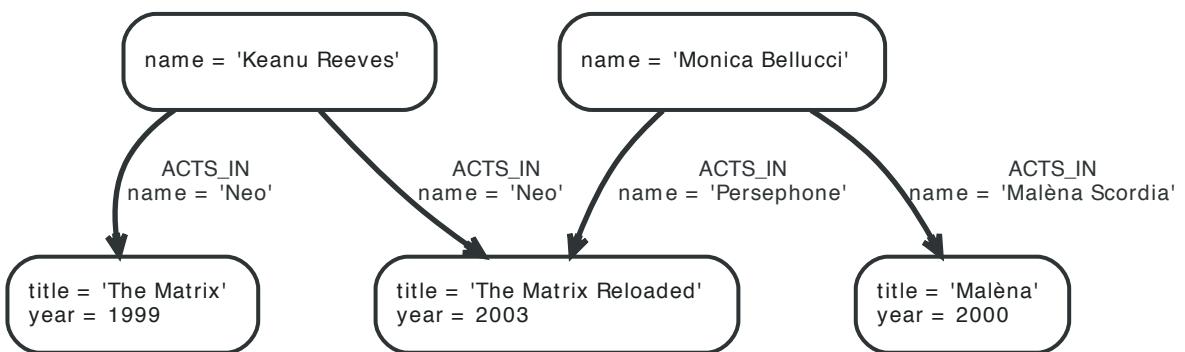
Note that there can be multiple values associated with the same entity and key.

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 35.1. Movie and Actor Graph



35.5. Remove

Removing⁵ 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/2.3.1/javadocs/org/neo4j/graphdb/index/Index.html#remove%28T,%20java.lang.String,%20java.lang.Object%29>

35.6. 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 35.5, “Remove” \[625\]](#).

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" ) );
```

35.7. 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.
}
```

Note

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

⁶ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/index/Index.html#get%28java.lang.String,%20java.lang.Object%29>

⁷ <http://neo4j.com/docs/2.3.1/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/2.3.1/javadocs/org/neo4j/graphdb/index/IndexHits.html>

¹⁰ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/index/IndexHits.html#getSingle%28%29>

¹¹ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/index/IndexHits.html#close%28%29>



Caution

You can't have *any whitespace* in the search term with this syntax. See [the section called "Querying with Lucene Query objects" \[633\]](#) for how to do that.

35.8. 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.close();

// 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/2.3.1/javadocs/org/neo4j/graphdb/index/RelationshipIndex.html>

¹³ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/graphdb/index/Index.html>

35.9. 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” [632] 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/2.3.1/javadocs/org/neo4j/graphdb/index/IndexHits.html#currentScore%28%29>

35.10. 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

35.11. 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/2.3.1/javadocs/org/neo4j/index/lucene/ValueContext.html>

¹⁸ <http://neo4j.com/docs/2.3.1/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 [TermQuery¹⁹](#) 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 [QueryContext²⁰](#) class:

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

¹⁹ http://lucene.apache.org/core/3_6_2/api/core/org/apache/lucene/search/TermQuery.html

²⁰ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/index/lucene/QueryContext.html>

Chapter 36. 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 is *not enforcing constraints* on the inserted data while inserting data.
- Batch insertion will re-populate all existing indexes and indexes created during batch insertion on shutdown.
- Batch insertion will verify all existing constraints and constraints 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.



Warning

Since the batch insertion doesn't enforce constraint during data loading, if the inserted data violate any constraint the batch inserter will fail on shutdown and the database *will* be inconsistent.

¹ <http://search.maven.org/#search|ga|1|neo4j-kernel>

36.1. 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() );

    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 the 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/2.3.1/javadocs/org/neo4j/unsafe/batchinsert/BatchInserter.html>

³ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/unsafe/batchinsert/BatchInserters.html>

⁴ <https://github.com/neo4j/neo4j/blob/2.3.1/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” \[465\]](#) for information on the available options.) e.g.

```
Map<String, String> config = new HashMap<>();
config.put( "dbms.pagecache.memory", "512m" );
BatchInserter inserter = BatchInsters.inserter(
    new File( "target/batchinserter-example-config" ).getAbsolutePath(), 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 ( FileReader input = new FileReader( new File( "target/docs/batchinsert-config" ).getAbsoluteFile() ) )
{
    Map<String, String> config = MapUtil.load( input );
    BatchInserter inserter = BatchInsters.inserter(
        "target/docs/batchinserter-example-config", 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.

36.2. Index Batch Insertion

For general notes on batch insertion, see [Chapter 36, *Batch Insertion*](#) [634].

Indexing during batch insertion is done using `BatchInserterIndex`⁵ which are provided via `BatchInserterIndexProvider`⁶. An example:

```
BatchInserter inserter = BatchInsters.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 35.10, “Configuration and fulltext indexes”](#) [631].

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/2.3.1/javadocs/org/neo4j/unsafe/batchinsert/BatchInserterIndex.html>

⁶ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/unsafe/batchinsert/BatchInserterIndexProvider.html>

⁷ <http://neo4j.com/docs/2.3.1/javadocs/org/neo4j/unsafe/batchinsert/BatchInserterIndex.html#flush%28%29>

⁸ <http://neo4j.com/docs/2.3.1/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	<i>for a graph or subgraph:</i> 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	<i>graph theory:</i> 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	<i>part of a database schema:</i> 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	<i>of a node</i> : 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	<i>graph theory</i> : 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 <i>projections</i> , as a <i>predicate</i> , or when setting <i>properties</i> on graph elements.
graph	1. data graph , 2. property graph , 3. <i>graph theory</i> : 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	<i>pertaining to a directed relationship</i> : 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: <i>graph theory</i> : This differs from mathematical graphs, where a label applies uniquely to a single vertex.
loop	a relationship that connects a node to itself.
neighbor	<i>of node</i> : another node , connected by a common relationship ; <i>of relationship</i> : 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. <i>Arithmetic</i> , such as +, /, % etc.; 2. <i>Logical</i> , such as OR, AND, NOT etc.; and 3. <i>Comparison</i> , such as <, >, = etc.
outgoing relationship	<i>pertaining to a directed relationship</i> : 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	<i>of a graph:</i> 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: <ul style="list-style-type: none">• 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	<i>graph theory:</i> the fundamental unit used to form a mathematical graph (plural: vertices). See node .

Appendix A. Resources

- Neo4j Cypher Refcard¹.
- Neo4j Javadocs². You can also download a *javadoc.jar* file from Maven Central, see [org.neo4j.doc:neo4j-javadoc³](http://org.neo4j.doc:neo4j-javadoc) or download it from [neo4j-javadoc-2.3.1-javadoc.jar⁴](http://neo4j-javadoc-2.3.1-javadoc.jar) 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” [102]
- Chapter 3, *Introduction to Cypher* [16]
- Chapter 21, *REST API* [299]
- Chapter 23, *Installation & Deployment* [438]
- Section 23.4, “Upgrading” [446]
- Chapter 27, *Security* [502]

¹ <http://neo4j.com/docs/2.3.1/cypher-refcard/>

² <http://neo4j.com/docs/2.3.1/javadoc/>

³ <http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.neo4j.doc%22%20AND%20a%3A%22neo4j-javadoc%22>

⁴ <http://central.maven.org/maven2/org/neo4j/doc/neo4j-javadoc/2.3.1/neo4j-javadoc-2.3.1-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 <http://neo4j.com/docs/stable/powershell.html>.

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.

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 <i>PORT</i>	Port of host to connect to (default: 1337).
-host <i>HOST</i>	Domain name or IP of host to connect to (default: localhost).
-name <i>NAME</i>	RMI name, i.e. <code>rmi://<host>:<port>/<name></code> (default: shell).
-readonly	Access the database in read-only mode.

LOCAL OPTIONS

-path <i>PATH</i>	The path to the database directory. If there is no database at the location, a new one will be created.
-pid <i>PID</i>	Process ID to connect to.
-readonly	Access the database in read-only mode.
-c <i>COMMAND</i>	Command line to execute. After executing it the shell exits.
-file <i>FILE</i>	File to read and execute. After executing it the shell exits. If - is supplied as filename data is read from stdin instead.
-config <i>CONFIG</i>	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.

For information regarding Windows, see <http://neo4j.com/docs/stable/powershell.html>.

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 <i>TAB</i> , between values in CSV data. The default option is ,.
--array-delimiter <array-delimiter-character>	Delimiter character, or <i>TAB</i> , 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.'
--multiline-fields <true/false>	Whether or not fields from input source can span multiple lines, i.e. contain newline characters. Default value: false
--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.
--ignore-empty-strings <true/false>	Whether or not empty string fields, i.e. "" from input source are ignored, i.e. treated as null. Default value: false
--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 <true/false>	Enable printing of error stack traces.

--bad-tolerance <max number of bad entries>	Number of bad entries before the import is considered failed. This tolerance threshold is about relationships referring to missing nodes. Format errors in input data are still treated as errors. Default value: 1000
--skip-bad-relationships <true/false>	Whether or not to skip importing relationships that refer to missing node ids, i.e. either start or end node id/group referring to node that wasn't specified by the node input data. Skipped nodes will be logged, containing at most number of entities specified by bad-tolerance. Default value: true
--skip-duplicate-nodes <true/false>	Whether or not to skip importing nodes that have the same id/group. In the event of multiple nodes within the same group having the same id, the first encountered will be imported whereas consecutive such nodes will be skipped. Skipped nodes will be logged, containing at most number of entities specified by bad-tolerance. Default value: false
--ignore-extra-columns <true/false>	Whether or not to ignore extra columns in the data not specified by the header. Skipped columns will be logged, containing at most number of entities specified by bad-tolerance. Default value: false

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.

For information regarding Windows, see <http://neo4j.com/docs/stable/powershell.html>.

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.

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.

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.