



The Neo4j Developer Manual

v3.2

Table of Contents

Introduction	1
1. Neo4j highlights	2
2. Graph database concepts	3
Get started	8
3. Install Neo4j	9
4. Get started with Cypher	10
Cypher	25
5. Introduction	26
6. Syntax	32
7. Clauses	63
8. Functions	140
9. Schema	190
10. Query tuning	206
11. Execution plans	221
12. Glossary of keywords	258
Drivers	264
13. Get started	265
14. Client applications	267
15. Sessions and transactions	272
16. Working with Cypher values	276
HTTP API	279
17. Transactional Cypher HTTP endpoint	280
18. Authentication and authorization	290
Extending Neo4j	293
19. Procedures	294
20. User-defined functions	301
21. Authentication and authorization plugins	305
Appendix A: Reference	309
22. Neo4j Status Codes	310
Appendix B: Terminology	315

This is the developer manual for Neo4j version 3.2, authored by the Neo4j Team.

The main parts of the manual are:

- [Introduction](#) — Introducing graph database concepts and Neo4j.
- [Get started](#) — Get started using Neo4j: Cypher and Drivers.
- [Cypher](#) — Reference for the Cypher query language.
- [Drivers](#) — Uniform language driver manual.
- [HTTP API](#) — Reference for the HTTP API for operating and querying Neo4j.
- [Extending Neo4j](#) — How to use procedures, user-defined functions, and authentication and authorization plugins for extending Neo4j.
- [Reference](#) — Neo4j status code reference.
- [Terminology](#) — Graph database terminology.

Who should read this?

This manual is written for the developer of a Neo4j client application.

Introduction

This chapter introduces graph database concepts and Neo4j highlights.

Chapter 1. Neo4j highlights

Connected data is all around us. Neo4j supports rapid development of graph powered systems that take advantage of the rich connectedness of data.

A native graph database: Neo4j is built from the ground up to be a graph database. The architecture is designed for optimizing fast management, storage, and traversal of nodes and relationships. In Neo4j, relationships are first class citizens that represent pre-materialized connections between entities. An operation known in the relational database world as a join, whose performance degrades exponentially with the number of relationships, is performed by Neo4j as navigation from one node to another, whose performance is linear.

This different approach to storing and querying connections between entities provides traversal performance of up to 4 million hops per second and core. As most graph searches are local to the larger neighborhood of a node, the total amount of data stored in a database will not affect operations runtime. Dedicated memory management, and highly scalable and memory efficient operations, contribute to the benefits.

Whiteboard friendly: The property graph approach allows consistent use of the same model throughout conception, design, implementation, storage, and visualization of any domain or use case. This allows all business stakeholders to participate throughout the development cycle. With the schema optional model, the domain model can be evolved continuously as requirements change, without penalty of expensive schema changes and migrations.

Cypher, the declarative graph query language, is designed to visually represent graph patterns of nodes and relationships. This highly capable, yet easily readable, query language is centered around the patterns that express concepts or questions from a specific domain. Cypher can also be extended for narrow optimizations for specific use cases.

Supports rapid development: Neo4j supports fast development of graph powered systems. Neo4j's development stems from the need to run real-time queries on highly related information; something no other database can provide. These unique Neo4j features get you up and running quickly and sustain fast application development for highly scalable applications.

Provides true data safety through ACID transactions: Neo4j uses transactions to guarantee that data is persisted in the case of hardware failure or system crashes.

Designed for business-critical and high-performance operations: Neo4j clustering is designed to support business-critical and high-performance applications. It can store hundreds of trillions of entities for the largest datasets imaginable while being sensitive to compact storage. Neo4j can be deployed as a scalable, fault-tolerant cluster of machines. Due to its high scalability, Neo4j clusters require only tens of machines, not hundreds or thousands, saving on cost and operational complexity. Other features for production applications include hot-backups and extensive monitoring.

Neo4j's application is only limited by your imagination.

Chapter 2. Graph database concepts

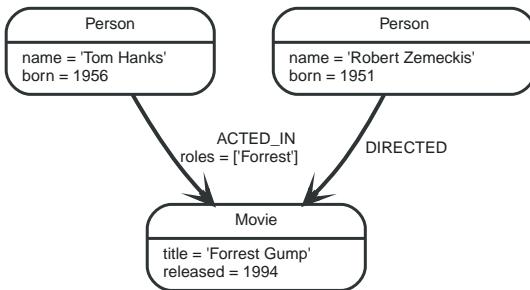
This chapter contains an introduction to the graph data model.

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. The Neo4j graph is based on the [property graph model](https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc) (<https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc>).

For graph database terminology, see [Terminology](#).

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

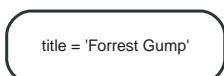


2.1.1. Nodes

A node in Neo4j is a node as described in the [property graph model](#) (<https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc#pgm-definitions-node>), with [properties](#) and [labels](#).

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

The simplest possible graph is a single node. Consider the graph below, consisting of one node with a single property **title**:



Let's add two more nodes and one more property on the node in the previous example:



2.1.2. Relationships

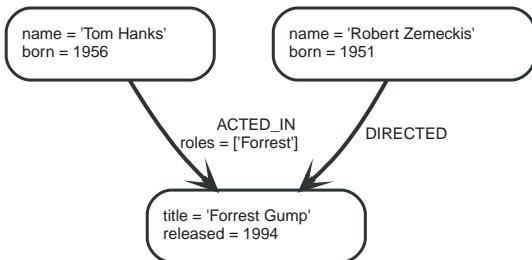
A relationship in Neo4j is a relationship as described in the [property graph model](#) (<https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc#pgm-definitions-relationship>), with a relationship type and [properties](#).

Relationships between nodes are the key feature of graph databases, as they allow for finding related

data. A relationship connects two nodes, and is guaranteed to have a valid source and target node.

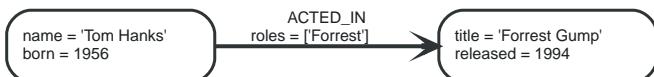
Relationships organize nodes into arbitrary structures, allowing a graph to resemble a list, a tree, a map, or a compound entity — any of which may be combined into yet more complex, richly interconnected structures.

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 the *source node* and **Forrest Gump** as the *target node*.



We observe 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.

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

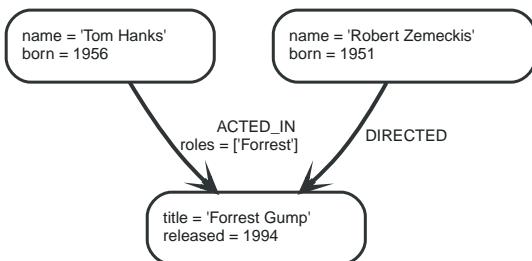


Table 1. Using relationship direction and type

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

2.1.3. Properties

A *property* in Neo4j is a property as described in the [property graph model](https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc#pgm-definitions-property) (<https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc#pgm-definitions-property>). Both nodes and relationships may have properties.

Properties are named values where the name (or key) is a string. The supported property values are:

- Numeric values
- String values
- Boolean values
- Lists of any of the above values



`null` is not a valid property value. Instead of storing it in the database, `null` can be modeled by the absence of a property key.

Table 2. Property value types

Type	Description	Value range
<code>boolean</code>	binary logic value	true/false
<code>integer</code>	64-bit integer	-9223372036854775808 to 9223372036854775807, inclusive
<code>float</code>	64-bit IEEE 754 floating-point number	-
<code>String</code>	sequence of Unicode characters	infinite

For further details on types and values, see the [Cypher type system CIP](https://github.com/opencypher/openCypher/blob/master/cip/1.accepted/CIP2015-09-16-public-type-system-type-annotation.adoc) (<https://github.com/opencypher/openCypher/blob/master/cip/1.accepted/CIP2015-09-16-public-type-system-type-annotation.adoc>).

2.1.4. Labels

A *label* in Neo4j is a label as described in the [property graph model](https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc#pgm-definitions-label) (<https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc#pgm-definitions-label>). 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.

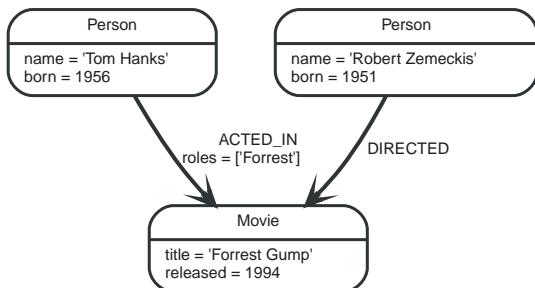
Labels are used when defining constraints and adding indexes for properties (see [Schema](#)).

For example, all nodes representing users could be labeled with the label `:User`. With that in place, you

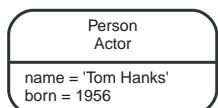
can ask Neo4j to perform operations only on your user nodes, such as finding all users with a given name.

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. A `:Suspended` label could be used to denote bank accounts that are suspended, a `:Seasonal` label to denote vegetables that are currently in season, and so on.

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



To exemplify how nodes may 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`. For more information on styling Cypher queries, refer to the [Cypher style guide](https://github.com/opencypher/openCypher/blob/master/docs/style-guide.adoc) (<https://github.com/opencypher/openCypher/blob/master/docs/style-guide.adoc>).

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

2.1.5. Traversal

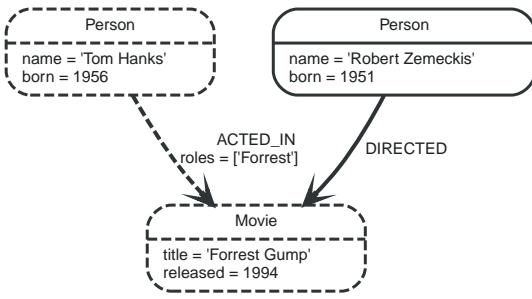
A traversal navigates through a graph to find paths.

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 [Cypher](#) for more information.

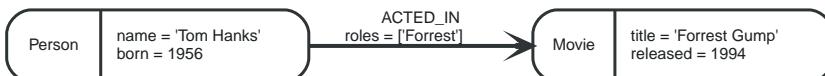
If we want to find out which movies Tom Hanks acted in according to our tiny example database, the traversal would start from the `Tom Hanks` node, follow any `:ACTED_IN` relationships connected to the node, and end up with `Forrest Gump` as the result (see the dashed lines):



2.1.6. Paths

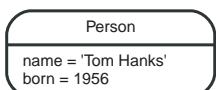
A path in Neo4j is a path as described in the [property graph model](#) (<https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc#pgm-definitions-path>). Paths are retrieved from a Cypher query or traversal.

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

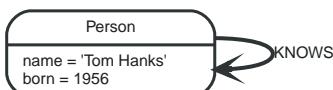


The path above has length one.

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:



2.1.7. 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.



Schema commands can only be applied on the master machine in a Neo4j cluster. If you apply them on a slave you will receive a [Neo.ClientError.Transaction.InvalidType](#) error code (see [Neo4j Status Codes](#)).

Indexes

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

Once you have 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.

For working with indexes in Cypher, see [Indexes](#).

Constraints

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

For working with constraints in Cypher, see [Constraints](#).

Get started

This chapter helps you get started quickly with Neo4j and the Cypher query language.

Chapter 3. Install Neo4j

Get started installing Neo4j.

Setting up a local environment for developing an application with Neo4j and its query language Cypher is easy. Download Neo4j from <http://neo4j.com/download/> and follow the installation instructions for your operating system. Read more about deploying Neo4j in the [Neo4j Operations Manual](#) Deployment. To get started with Cypher, continue reading. Open up a web browser and point it to <http://localhost:7474> to follow along and try out the queries explained below.

Chapter 4. Get started with Cypher

This guide will introduce you to Cypher, Neo4j's query language. It will help you:

- start thinking about graphs and patterns
- apply this knowledge to simple problems
- learn how to write Cypher statements

4.1. Patterns

- [Node syntax](#)
- [Relationship syntax](#)
- [Pattern syntax](#)
- [Pattern variables](#)
- [Clauses](#)

Neo4j's Property Graphs are composed of nodes and relationships, either of which may have properties. Nodes represent entities, for example concepts, events, places and things. Relationships 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.

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, for example 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.

Like SQL, used in relational databases, Cypher is a textual declarative query language. It uses a form of [ASCII art](#) (https://en.wikipedia.org/wiki/ASCII_art) to represent graph-related patterns. SQL-like clauses and keywords, for example `MATCH`, `WHERE` and `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, for example nodes, relationships, paths and lists. However, Cypher does *not* tell Neo4j *how* to find nodes, traverse relationships etc.

Diagrams made up of icons and arrows are commonly used to visualize graphs. Textual annotations

provide labels, define properties etc.

4.1.1. 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:

```
()  
(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 a variable, for example: `(matrix)`. A variable is restricted to a single statement. It 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, for example `title`, are represented as a list of key/value pairs, enclosed within a pair of braces, for example: `{name: "Keanu Reeves"}`. Properties can be used to store information and/or restrict patterns.

4.1.2. Relationship syntax

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

```
-->  
-[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. A variable (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.)

4.1.3. 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`. Variables (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.

4.1.4. Pattern variables

To increase modularity and reduce repetition, Cypher allows patterns to be assigned to variables. 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)`.

4.1.5. Clauses

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

- create and match patterns in the graph
- filter, project, sort, or paginate results
- 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.

4.2. Patterns in practice

- [Creating data](#)
- [Matching patterns](#)
- [Attaching structures](#)
- [Completing patterns](#)

4.2.1. 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.

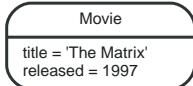
For the patterns we've looked at so far this could look like the following:

```
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.

```
+-----+
| No data returned. |
+-----+
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 variable we've assigned to our pattern elements.

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

This is what gets returned:

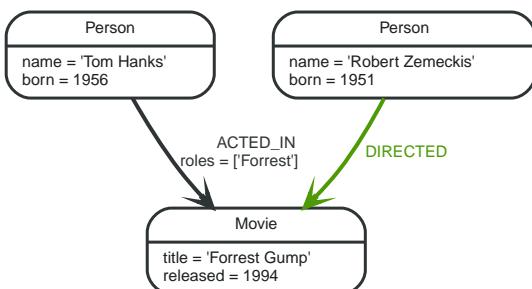
```
+-----+
| p |
+-----+
| Node[1]{name:"Keanu Reeves",born:1964} |
+-----+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

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.

4.2.2. 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.



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.

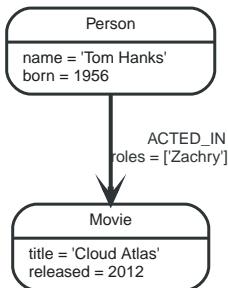
Now we know enough to connect new nodes to existing ones and can combine `MATCH` and `CREATE` to attach structures to the graph.

4.2.3. 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:



It is important to remember that we can assign variables 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.



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`.

4.2.4. 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.



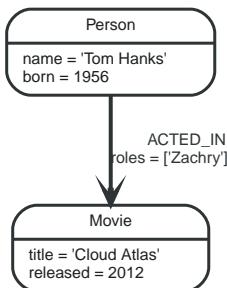
A **MERGE** clause without any previously assigned variables in it either matches the full pattern or creates the full pattern. It never produces a partial mix of matching and creating within a pattern. To achieve a partial match/create, make sure to use already defined variables for the parts that shouldn't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



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
```

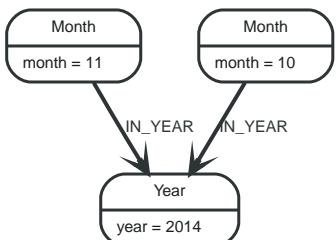


In case the direction of a relationship is arbitrary, you can leave off the arrowhead. **MERGE** will then check for the relationship in either direction, and create a new directed relationship if no matching relationship was found.

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.

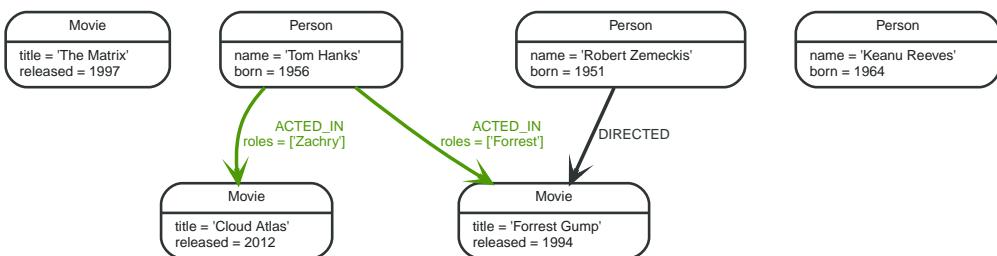
4.3. Getting correct results

- [Filtering results](#)
- [Returning results](#)
- [Aggregating information](#)
- [Ordering and pagination](#)
- [Collecting aggregation](#)

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:



4.3.1. 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 list.

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

```
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 list-predicates which we will look at later on.

4.3.2. Returning results

So far we've returned only nodes, relationships, or paths directly via their variables. 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.

4.3.3. 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)`.

4.3.4. 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	

4.3.5. Collecting aggregation

The most helpful aggregation function is `collect()`, which, appropriately collects all aggregated values into a list. This comes very handy in many situations as no information of details is lost while aggregating.

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

To retrieve the cast of each movie in our database this statement could be used:

```
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 list functions or predicates.

4.4. Composing large statements

- [UNION](#)
- [WITH](#)

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)
```

4.4.1. 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
```

4.4.2. WITH

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



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.

4.5. Constraints and indexes

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

4.5.1. Using constraints

You can 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 is an example of how to use labels and add constraints and indexes to them. Let's start out by adding a constraint. In this case we decide that every `Movie` node should have a unique `title`.

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

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

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

4.5.2. Using indexes

The main reason for using indexes in a graph database is to find the starting point in the graph as fast as possible. After that seek you rely on in-graph structures and the first class citizenship of relationships in the graph database to achieve high performance. Thus graph queries themselves do not need indexes to run fast.

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.

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

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

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;
```

4.6. 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 change the protocol, for example if the redirect is going from [HTTPS](#) to [HTTP](#).

For more details, see [LOAD CSV](#).

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 "{csv-dir}/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.



When using `MERGE` or `MATCH` with `LOAD CSV` we need to make sure we have an index (see [Indexes](#)) or a unique constraint (see [Constraints](#)) 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 "{csv-dir}/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 [PERIODIC COMMIT query hint](#)) 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 "{csv-dir}/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
```

Cypher

This chapter contains the complete and authoritative documentation for the Cypher query language.

Chapter 5. Introduction

For a short introduction, see [What is Cypher?](#). To take your first steps with Cypher, see [Get started with Cypher](#). For the terminology used, see [Terminology](#).

- [What is Cypher?](#)
- [Querying and updating the graph](#)
- [Transactions](#)
- [Uniqueness](#)
- [Compatibility](#)

5.1. What is Cypher?

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](#) (<http://gremlin.tinkerpop.com>), and [the JRuby Neo4j bindings](#) (<https://github.com/neo4jrb/neo4j/>). 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](#) (<http://en.wikipedia.org/wiki/SQL>). Pattern matching borrows expression approaches from [SPARQL](#) (<http://en.wikipedia.org/wiki/SPARQL>). Some of the list 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 variables 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:

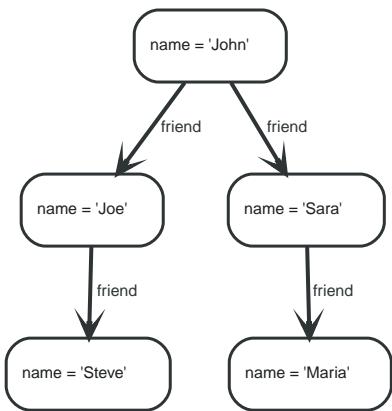


Figure 1. Example Graph

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
"Joe"	"Steve"
"John"	"Sara"

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.

5.2. Querying and updating the graph

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

5.2.1. The structure of update 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.friendsCount = friendsCount
RETURN n.friendsCount
```

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

5.2.2. 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.

5.3. 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 [\[rest-api-transactional\]](#).

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.

5.4. Uniqueness

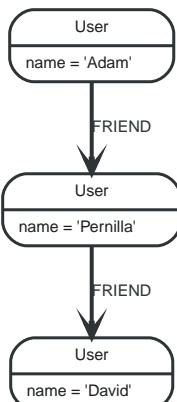
While pattern matching, Neo4j makes sure to not include matches where the same graph relationship is found multiple times in a single pattern. In most use cases, this is a sensible thing to do.

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
```

5.5. Compatibility

Cypher is not a static language. New versions introduce new features and sometimes old features get dropped. Older versions of the language can still be accessed if required. There are two ways to select which version to use in queries.

1. Setting a version for all queries You can configure your database with the configuration parameter `cypher.default_language_version`, and enter which version you'd like to use (see [Supported Language Versions](#)). Every Cypher query will use this version, provided the query hasn't explicitly been configured as described in the next item below.
2. Setting a version on a query by query basis The other way is on a query by query basis. By simply

putting `CYPHER 2.3` at the beginning, that particular query will be executed with the version of Cypher included in Neo4j 2.3.

Below is an example using the `START` clause to access a legacy index:

```
CYPHER 2.3
START n=node:nodes(name = "A")
RETURN n
```

5.5.1. 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
```



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 [MATCH](#) for more information on the correct syntax for this. The `START` clause should only be used when accessing legacy indexes.

5.5.2. Supported Language Versions

Neo4j 3.2 supports the following versions of the Cypher language:

- Neo4j Cypher 3.2
- Neo4j Cypher 3.1
- Neo4j Cypher 2.3



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 6. Syntax

This section describes the syntax of the Cypher query language.

- [Values](#)
- [Expressions](#)
 - Expressions in general
 - Note on string literals
 - `CASE` Expressions
- [Variables](#)
- [Parameters](#)
 - String literal
 - Regular expression
 - Case-sensitive string pattern matching
 - Create node with properties
 - Create multiple nodes with properties
 - Setting all properties on a node
 - `SKIP` and `LIMIT`
 - Node id
 - Multiple node ids
 - Calling procedures
 - Index value (legacy indexes)
 - Index query (legacy indexes)
- [Operators](#)
 - Operators at a glance
 - General operators
 - Mathematical operators
 - Comparison operators
 - Boolean operators
 - String operators
 - List operators
 - Property operators
 - Equality and comparison of values
 - Ordering and comparison of values
 - Chaining comparison operations
- [Comments](#)
- [Patterns](#)
 - Patterns for nodes
 - Patterns for related nodes

- Patterns for labels
- Specifying properties
- Patterns for relationships
- Variable-length pattern matching
- Assigning to path variables
- Lists
 - Lists in general
 - List comprehension
 - Pattern comprehension
- Maps
 - Literal maps
 - Map projection
- Working with `null`
 - Introduction to `null` in Cypher
 - Logical operations with `null`
 - The `IN` operator and `null`
 - Expressions that return `null`

6.1. 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,
- Lists of any other type of value.

Most types of values can be constructed in a query using literal expressions (see [Expressions](#)). Special care must be taken when using `null`, as `null` is a value of every type (see [Working with null](#)). 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.

6.2. Expressions

- Expressions in general
- Note on string literals
- CASE expressions
 - Simple CASE form: comparing an expression against multiple values

- Generic CASE form: allowing for multiple conditionals to be expressed
- Distinguishing between when to use the simple and generic CASE forms

6.2.1. 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, myFancyVariable, 'A name with weird stuff in it[]!`.`
- A property: `n.prop, x.prop, rel.thisProperty, myFancyVariable.`(weird property name)`.`
- A dynamic property: `n["prop"], rel[n.city + n.zip], map[coll[0]].`
- A parameter: `$param, $0`
- A list 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, exists(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.

6.2.2. Note on string literals

String literals can contain the following escape sequences:

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

Escape sequence	Character
\Uxxxxxxxx	Unicode UTF-32 code point (8 hex digits must follow the \U)

6.2.3. CASE expressions

Generic conditional expressions may be expressed using the well-known `CASE` construct. Two variants of `CASE` exist within Cypher: the simple form, which allows an expression to be compared against multiple values, and the generic form, which allows multiple conditional statements to be expressed.

The following graph is used for the examples below:

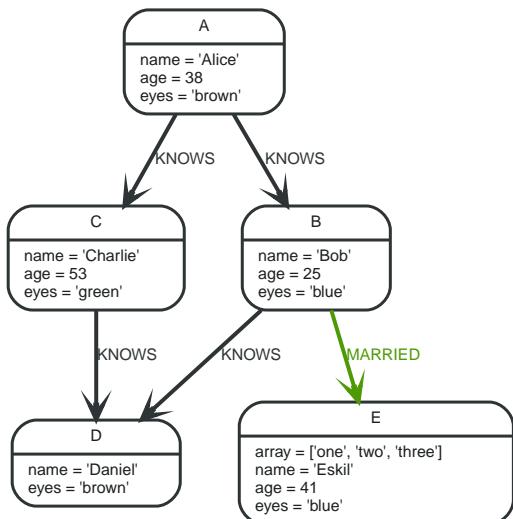


Figure 2. Graph

Simple `CASE` form: comparing an expression against multiple values

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 returned. However, if there is no `ELSE` case and no match is found, `null` will be returned.

Syntax:

```

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

Arguments:

Name	Description
<code>test</code>	A valid expression.
<code>value</code>	An expression whose result will be compared to <code>test</code> .
<code>result</code>	This is the expression returned as output if <code>value</code> matches <code>test</code> .
<code>default</code>	If no match is found, <code>default</code> is returned.

Query

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

Table 3. Result

result
2
1
3
2
1
5 rows

Generic **CASE** form: allowing for multiple conditionals to be expressed

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 returned. However, if there is no **ELSE** case and no match is found, **null** will be returned.

Syntax:

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

Arguments:

Name	Description
predicate	A predicate that is tested to find a valid alternative.
result	This is the expression returned as output if predicate evaluates to true .
default	If no match is found, default is returned.

Query

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

Table 4. Result

result

2
1
3
3
1
5 rows

Distinguishing between when to use the simple and generic **CASE** forms

Owing to the close similarity between the syntax of the two forms, sometimes it may not be clear at the outset as to which form to use. We illustrate this scenario by means of the following query, in which there is an expectation that `age_10_years_ago` is `-1` if `n.age` is `null`:

Query

```
MATCH (n)
RETURN n.name,
CASE n.age
WHEN n.age IS NULL THEN -1
ELSE n.age - 10 END AS age_10_years_ago
```

However, as this query is written using the simple **CASE** form, instead of `age_10_years_ago` being `-1` for the node named `Daniel`, it is `null`. This is because a comparison is made between `n.age` and `n.age IS NULL`. As `n.age IS NULL` is a boolean value, and `n.age` is an integer value, the `WHEN n.age IS NULL THEN -1` branch is never taken. This results in the `ELSE n.age - 10` branch being taken instead, returning `null`.

Table 5. Result

n.name	age_10_years_ago
"Alice"	28
"Bob"	15
"Charlie"	43
"Daniel"	<null>
"Eskil"	31
5 rows	

The corrected query, behaving as expected, is given by the following generic **CASE** form:

Query

```
MATCH (n)
RETURN n.name,
CASE
WHEN n.age IS NULL THEN -1
ELSE n.age - 10 END AS age_10_years_ago
```

We now see that the `age_10_years_ago` correctly returns `-1` for the node named `Daniel`.

Table 6. Result

n.name	age_10_years_ago
"Alice"	28

n.name	age_10_years_ago
"Bob"	15
"Charlie"	43
"Daniel"	-1
"Eskil"	31
5 rows	

6.3. Variables

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 **variables**.

In this example:

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

The variables are **n** and **b**.

Variable 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 variable using backquote (`) signs.

The same rules apply to property names.

Variables are only visible in the same query part



Variables are not carried over to subsequent queries. If multiple query parts are chained together using **WITH**, variables have to be listed in the **WITH** clause to be carried over to the next part. For more information see **WITH**.

6.4. Parameters

- [Introduction](#)
- [String literal](#)
- [Regular expression](#)
- [Case-sensitive string pattern matching](#)
- [Create node with properties](#)
- [Create multiple nodes with properties](#)
- [Setting all properties on a node](#)
- [SKIP and LIMIT](#)
- [Node id](#)
- [Multiple node ids](#)
- [Calling procedures](#)
- [Index value \(legacy indexes\)](#)
- [Index query \(legacy indexes\)](#)

6.4.1. Introduction

Cypher supports querying with parameters. This means developers don't have to resort to string building to create a query. Additionally, parameters make caching of execution plans much easier for Cypher, thus leading to faster query execution times.

Parameters can be used for:

- literals and expressions
- node and relationship ids
- for legacy indexes only: index values and queries

Parameters cannot be used for the following constructs, as these form part of the query structure that is compiled into a query plan:

- property keys; so, `MATCH (n) WHERE n.$param = 'something'` is invalid
- relationship types
- labels

Parameters may consist of letters and numbers, and any combination of these, but cannot start with a number or a currency symbol.

For details on using parameters via the Neo4j REST API, see [\[rest-api-transactional\]](#).

We provide below a comprehensive list of examples of parameter usage. In these examples, parameters are given in JSON; the exact manner in which they are to be submitted depends upon the driver being used.



It is recommended that the new parameter syntax `$param` is used, as the old syntax `{param}` is deprecated and will be removed altogether in a later release.

6.4.2. String literal

Parameters

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

Query

```
MATCH (n:Person)  
WHERE n.name = $name  
RETURN n
```

You can use parameters in this syntax as well:

Parameters

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

Query

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

6.4.3. Regular expression

Parameters

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

Query

```
MATCH (n:Person)
WHERE n.name =~ $regex
RETURN n.name
```

6.4.4. Case-sensitive string pattern matching

Parameters

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

Query

```
MATCH (n:Person)
WHERE n.name STARTS WITH $name
RETURN n.name
```

6.4.5. Create node with properties

Parameters

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

Query

```
CREATE ($props)
```

6.4.6. Create multiple nodes with properties

Parameters

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

Query

```
UNWIND $props AS properties  
CREATE (n:Person)  
SET n = properties  
RETURN n
```

6.4.7. Setting all properties on a node

Note that this will replace all the current properties.

Parameters

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

Query

```
MATCH (n:Person)  
WHERE n.name='Michaela'  
SET n = $props
```

6.4.8. SKIP and LIMIT

Parameters

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

Query

```
MATCH (n:Person)  
RETURN n.name  
SKIP $s  
LIMIT $l
```

6.4.9. Node id

Parameters

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

Query

```
MATCH (n)
WHERE id(n)= $id
RETURN n.name
```

6.4.10. Multiple node ids

Parameters

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

Query

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

6.4.11. Calling procedures

Parameters

```
{
  "indexname" : ":Person(name)"
}
```

Query

```
CALL db.resampleIndex($indexname)
```

6.4.12. Index value (legacy indexes)

Parameters

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

Query

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

6.4.13. Index query (legacy indexes)

Parameters

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

Query

```
START n=node:people($query)  
RETURN n
```

6.5. Operators

- Operators at a glance
- General operators
 - Using the **DISTINCT** operator
 - Accessing properties of a nested literal map using the **.** operator
 - Filtering on a dynamically-computed property key using the **[]** operator
- Mathematical operators
 - Using the exponentiation operator **^**
 - Using the unary minus operator **-**
- Comparison operators
 - Comparing two numbers
 - Using **STARTS WITH** to filter names
- Boolean operators
 - Using boolean operators to filter numbers
- String operators
 - Using a regular expression with **=~** to filter words
- List operators
 - Concatenating two lists using **+**
 - Using **IN** to check if a number is in a list
 - Accessing elements in a list using the **[]** operator
- Property operators
- Equality and comparison of values
- Ordering and comparison of values
- Chaining comparison operations

6.5.1. Operators at a glance

General operators	DISTINCT , . for property access, [] for dynamic property access
Mathematical operators	+, -, *, /, %, ^
Comparison operators	=, <>, <, >, <=, >=, IS NULL, IS NOT NULL

String-specific comparison operators	STARTS WITH, ENDS WITH, CONTAINS
Boolean operators	AND, OR, XOR, NOT
String operators	+ for concatenation, =~ for regex matching
List operators	+ for concatenation, IN to check existence of an element in a list, [] for accessing element(s)

6.5.2. General operators

The general operators comprise:

- remove duplicates values: DISTINCT
- access the property of a node, relationship or literal map using the dot operator: .
- dynamic property access using the subscript operator: []

Using the DISTINCT operator

Retrieve the unique eye colors from Person nodes.

Query

```
CREATE (a:Person { name: 'Anne', eyeColor: 'blue' }),(b:Person { name: 'Bill', eyeColor: 'brown' })
,(c:Person { name: 'Carol', eyeColor: 'blue' })
WITH a, b, c
MATCH (p:Person)
RETURN DISTINCT p.eyeColor
```

Even though both 'Anne' and 'Carol' have blue eyes, 'blue' is only returned once.

Table 7. Result

p.eyeColor
"blue"
"brown"
2 rows
Nodes created: 3
Properties set: 6
Labels added: 3

DISTINCT is commonly used in conjunction with aggregating functions.

Accessing properties of a nested literal map using the . operator

Query

```
WITH { person: { name: 'Anne', age: 25 } } AS p
RETURN p.person.name
```

Table 8. Result

p.person.name
"Anne"
1 row

Filtering on a dynamically-computed property key using the `[]` operator

Query

```
CREATE (a:Restaurant { name: 'Hungry Jo', rating_hygiene: 10, rating_food: 7 }),(b:Restaurant { name: 'Buttercup Tea Rooms', rating_hygiene: 5, rating_food: 6 }),(c1:Category { name: 'hygiene' }),(c2:Category { name: 'food' })
WITH a, b, c1, c2
MATCH (restaurant:Restaurant),(category:Category)
WHERE restaurant["rating_" + category.name] > 6
RETURN DISTINCT restaurant.name
```

Table 9. Result

restaurant.name
"Hungry Jo"
1 row
Nodes created: 4
Properties set: 8
Labels added: 4

See [Basic usage](#) for more details on dynamic property access.

6.5.3. Mathematical operators

The mathematical operators comprise:

- addition: `+`
- subtraction or unary minus: `-`
- multiplication: `*`
- division: `/`
- modulo division: `%`
- exponentiation: `^`

Using the exponentiation operator `^`

Query

```
WITH 2 AS number, 3 AS exponent
RETURN number ^ exponent AS result
```

Table 10. Result

result
8.0
1 row

Using the unary minus operator `-`

Query

```
WITH -3 AS a, 4 AS b
RETURN b - a AS result
```

Table 11. Result

result
7
1 row

6.5.4. Comparison operators

The comparison operators comprise:

- equality: `=`
- inequality: `<>`
- less than: `<`
- greater than: `>`
- less than or equal to: `<=`
- greater than or equal to: `>=`
- `IS NULL`
- `IS NOT NULL`

String-specific comparison operators comprise:

- `STARTS WITH`: perform case-sensitive prefix searching on strings
- `ENDS WITH`: perform case-sensitive suffix searching on strings
- `CONTAINS`: perform case-sensitive inclusion searching in strings

Comparing two numbers

Query

```
WITH 4 AS one, 3 AS two
RETURN one > two AS result
```

Table 12. Result

result
true
1 row

See [Equality and comparison of values](#) for more details on the behavior of comparison operators, and [Using ranges](#) for more examples showing how these may be used.

Using `STARTS WITH` to filter names

Query

```
WITH ['John', 'Mark', 'Jonathan', 'Bill'] AS somenames
UNWIND somenames AS names
WITH names AS candidate
WHERE candidate STARTS WITH 'Jo'
RETURN candidate
```

Table 13. Result

candidate
"John"
"Jonathan"
2 rows

[String matching](#) contains more information regarding the string-specific comparison operators as well as additional examples illustrating the usage thereof.

6.5.5. Boolean operators

The boolean operators — also known as logical operators — comprise:

- conjunction: `AND`
- disjunction: `OR`,
- exclusive disjunction: `XOR`
- negation: `NOT`

Here is the truth table for `AND`, `OR`, `XOR` and `NOT`.

a	b	a AND b	a OR b	a XOR b	NOT a
false	false	false	false	false	true
false	null	false	null	null	true
false	true	false	true	true	true
true	false	false	true	true	false
true	null	null	true	null	false
true	true	true	true	false	false
null	false	false	null	null	null
null	null	null	null	null	null
null	true	null	true	null	null

Using boolean operators to filter numbers

Query

```
WITH [2, 4, 7, 9, 12] AS numberlist
UNWIND numberlist AS number
WITH number
WHERE number = 4 OR (number > 6 AND number < 10)
RETURN number
```

Table 14. Result

number
4
7
9
3 rows

6.5.6. String operators

String operators comprise:

- concatenating strings: `+`
- matching a regular expression: `=~`

Using a regular expression with `=~` to filter words

Query

```
WITH ['mouse', 'chair', 'door', 'house'] AS wordlist
UNWIND wordlist AS word
WITH word
WHERE word =~ '.*ous.*'
RETURN word
```

Table 15. Result

word
"mouse"
"house"
2 rows

Further information and examples regarding the use of regular expressions in filtering can be found in [Regular expressions](#). In addition, refer to [String-specific comparison operators comprise](#): for details on string-specific comparison operators.

6.5.7. List operators

List operators comprise:

- concatenating lists: `+`
- checking if an element exists in a list: `IN`
- accessing an element(s) in a list using the subscript operator: `[]`

Concatenating two lists using `+`

Query

```
RETURN [1,2,3,4,5]+[6,7] AS myList
```

Table 16. Result

myList
[1,2,3,4,5,6,7]
1 row

Using `IN` to check if a number is in a list

Query

```
WITH [2, 3, 4, 5] AS numberlist
UNWIND numberlist AS number
WITH number
WHERE number IN [2, 3, 8]
RETURN number
```

Table 17. Result

number
2
3
2 rows

Accessing elements in a list using the `[]` operator

Query

```
WITH ['Anne', 'John', 'Bill', 'Diane', 'Eve'] AS names
RETURN names[1..3] AS result
```

The square brackets will extract the elements from the start index `1`, and up to (but excluding) the end index `3`.

Table 18. Result

result
["John", "Bill"]
1 row

More details on lists can be found in [Lists in general](#).

6.5.8. Property operators



Since version 2.0, the previously supported property operators `?` and `!` have been removed. This syntax is no longer supported. Missing properties are now returned as `null`. Please use `(NOT(exists(<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.

6.5.9. Equality and comparison of values

Equality

Cypher supports comparing values (see [Values](#)) 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 lists 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 lists of alternating nodes and relationships and are equal to all lists 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.

6.5.10. 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` (e.g. `null < 3` is `null`).
- It is an error to compare other types of values with each other for ordering.

6.5.11. 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 (although perhaps not elegant).

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 [Comparison operators](#).

6.6. 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
```

6.7. Patterns

- [Introduction](#)
- [Patterns for nodes](#)
- [Patterns for related nodes](#)
- [Patterns for labels](#)
- [Specifying properties](#)
- [Patterns for relationships](#)
- [Variable-length pattern matching](#)
- [Assigning to path variables](#)

6.7.1. Introduction

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 detail in:

- [MATCH](#)
- [OPTIONAL MATCH](#)
- [CREATE](#)
- [MERGE](#)
- [Using path patterns in WHERE](#)

6.7.2. 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 variable `a`.

6.7.3. Patterns for related nodes

A more powerful construct is a pattern that describes 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 manner 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, as follows:

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

6.7.4. Patterns for 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)
```

6.7.5. 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 is given by:

```
(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.

6.7.6. Patterns for 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, as exemplified by:

```
(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 variable 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 as follows:

```
(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, as exemplified by:

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

6.7.7. Variable-length pattern matching



Variable length pattern matching in versions 2.1.x and earlier does not enforce relationship uniqueness for patterns described within 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 [Uniqueness](#). 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)
```

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 graph and query below:

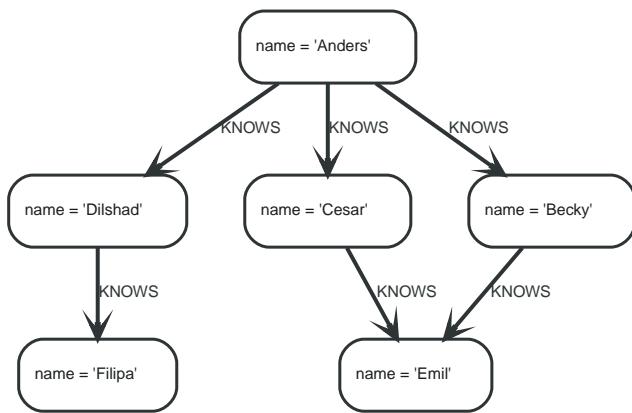


Figure 3. Graph

Query

```

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

Table 19. 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 hops away. This is a typical example of finding first and second degree friends.

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

6.7.8. Assigning to path variables

As described above, a series of connected nodes and relationships is called a "path". Cypher allows paths to be named using an identifier, as exemplified by:

```

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

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

6.8. Lists

Cypher has comprehensive support for lists.

- [Lists in general](#)
- [List comprehension](#)
- [Pattern comprehension](#)

6.8.1. Lists in general

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

Query

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

Table 20. Result

list
[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 list containing all numbers between given start and end numbers. Range is inclusive in both ends.

To access individual elements in the list, 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]
```

Table 21. Result

range(0, 10)[3]
3
1 row

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

Query

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

Table 22. Result

range(0, 10)[-3]
8
1 row

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

Query

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

Table 23. Result

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

Query

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

Table 24. Result

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

Query

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

Table 25. Result

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

Query

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

Table 26. Result

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



Out-of-bound slices are simply truncated, but out-of-bound single elements return `null`.

Query

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

Table 27. Result

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

Query

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

Table 28. Result

range(0, 10)[5..15]
[5,6,7,8,9,10]
1 row

You can get the `size` of a list as follows:

Query

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

Table 29. Result

size(range(0, 10)[0..3])
3
1 row

6.8.2. List comprehension

List comprehension is a syntactic construct available in Cypher for creating a list based on existing lists. 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
```

Table 30. 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
```

Table 31. Result

result
[0,2,4,6,8,10]
1 row

Query

```
RETURN [x IN range(0,10) | x^3] AS result
```

Table 32. 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

6.8.3. Pattern comprehension

Pattern comprehension is a syntactic construct available in Cypher for creating a list based on matchings of a pattern. A pattern comprehension will match the specified pattern just like a normal

`MATCH` clause, with predicates just like a normal `WHERE` clause, but will yield a custom projection as specified.

The following graph is used for the example below:

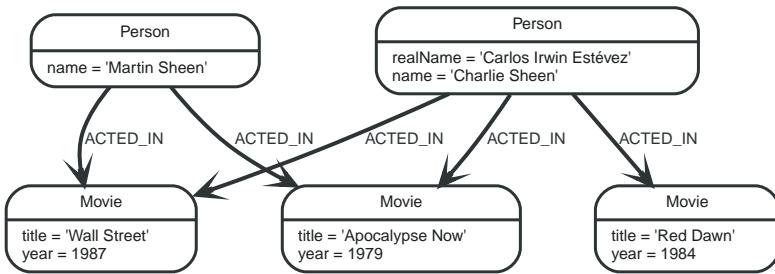


Figure 4. Graph

Query

```
MATCH (a:Person { name: 'Charlie Sheen' })
RETURN [(a)-->(b) WHERE b:Movie | b.year] AS years
```

Table 33. Result

years
[1979,1984,1987]
1 row

The whole predicate, including the `WHERE` keyword, is optional and may be omitted.

6.9. Maps

Cypher has solid support for maps.

- Literal maps
- Map projection
 - Examples of map projection

The following graph is used for the examples below:

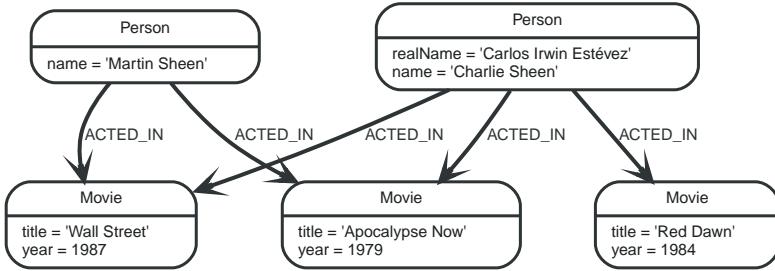


Figure 5. Graph

6.9.1. 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', listKey: [{ inner: 'Map1' }, { inner: 'Map2' }]}  
-----  
Table 34. Result
```

{key: 'Value', listKey: [{inner: 'Map1'}, {inner: 'Map2'}]}
{listKey -> [{inner -> "Map1"},{inner -> "Map2"}], key -> "Value"}
1 row

6.9.2. Map projection

Cypher supports a concept called "map projections". It allows for easily constructing map projections from nodes, relationships and other map values.

A map projection begins with the variable bound to the graph entity to be projected from, and contains a body of comma-separated map elements, enclosed by `{` and `}`.

```
map_variable {map_element, [ , ...n]}
```

A map element projects one or more key-value pairs to the map projection. There exist four different types of map projection elements:

- Property selector - Projects the property name as the key, and the value from the `map_variable` as the value for the projection.
- Literal entry - This is a key-value pair, with the value being arbitrary expression `key: <expression>`.
- Variable selector - Projects a variable, with the variable name as the key, and the value the variable is pointing to as the value of the projection. Its syntax is just the variable.
- All-properties selector - projects all key-value pairs from the `map_variable` value.

Note that if the `map_variable` points to a `null` value, the whole map projection will evaluate to `null`.

Examples of map projections

Find 'Charlie Sheen' and return data about him and the movies he has acted in. This example shows an example of map projection with a literal entry, which in turn also uses map projection inside the aggregating `collect()`.

Query

```
MATCH (actor:Person { name: 'Charlie Sheen' })-[:ACTED_IN]->(movie:Movie)  
RETURN actor { .name, .realName, movies: collect(movie { .title, .year })}  
-----  
Table 35. Result
```

actor
{name -> "Charlie Sheen", realName -> "Carlos Irwin Estévez", movies -> [{title -> "Apocalypse Now", year -> 1979},{title -> "Red Dawn", year -> 1984},{title -> "Wall Street", year -> 1987}]}
1 row

Find all persons that have acted in movies, and show number for each. This example introduces an variable with the count, and uses a variable selector to project the value.

Query

```
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)
WITH actor, count(movie) AS nrOfMovies
RETURN actor { .name, nrOfMovies }
```

Table 36. Result

actor
{name -> "Martin Sheen", nrOfMovies -> 2}
{name -> "Charlie Sheen", nrOfMovies -> 3}
2 rows

Again, focusing on '**Charlie Sheen**', this time returning all properties from the node. Here we use an all-properties selector to project all the node properties, and additionally, explicitly project the property `age`. Since this property does not exist on the node, a `null` value is projected instead.

Query

```
MATCH (actor:Person { name: 'Charlie Sheen' })
RETURN actor { .*, .age }
```

Table 37. Result

actor
{realName -> "Carlos Irwin Estévez", name -> "Charlie Sheen", age -> <null>}
1 row

6.10. Working with `null`

- [Introduction to `null` in Cypher](#)
- [Logical operations with `null`](#)
- [The `IN` operator and `null`](#)
- [Expressions that return `null`](#)

6.10.1. 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`.

6.10.2. Logical operations with `null`

The logical operators (`AND`, `OR`, `XOR`, `NOT`) treat `null` as the 'unknown' value of three-valued logic.

Here is the truth table for `AND`, `OR`, `XOR` and `NOT`.

a	b	a AND b	a OR b	a XOR b	NOT a
false	false	false	false	false	true
false	null	false	null	null	true
false	true	false	true	true	true
true	false	false	true	true	false
true	null	null	true	null	false
true	true	true	true	false	false
null	false	false	null	null	null
null	null	null	null	null	null
null	true	null	true	null	null

6.10.3. The IN operator and null

The `IN` operator follows similar logic. If Cypher knows that something exists in a list, the result will be `true`. Any list 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.

6.10.4. Expressions that return null

- Getting a missing element from a list: `[][], 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 7. Clauses

This section contains information on all the clauses in the Cypher query language.

- [Reading clauses](#)
- [Projecting clauses](#)
- [Reading sub-clauses](#)
- [Reading hints](#)
- [Writing clauses](#)
- [Reading/Writing clauses](#)
- [Set operations](#)
- [Importing data](#)
- [Schema clauses](#)

Reading clauses

These comprise clauses that read data from the database.

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 variables in the query and values derived from the database. This set is refined and augmented by subsequent parts of the query.

Clause	Description
MATCH	Specify the patterns to search for in the database.
OPTIONAL MATCH	Specify the patterns to search for in the database while using <code>nulls</code> for missing parts of the pattern.
START	Find starting points through legacy indexes.

Projecting clauses

These comprise clauses that define which expressions to return in the result set.

Clause	Description
RETURN	Defines what to include in the query result set.
WITH	Allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.
UNWIND	Expands a list into a sequence of rows.

Reading sub-clauses

These comprise sub-clauses that must operate as part of reading clauses.

Sub-clause	Description
WHERE	Adds constraints to the patterns in a MATCH or OPTIONAL MATCH clause or filters the results of a WITH clause.
ORDER BY	A sub-clause following RETURN or WITH , specifying that the output should be sorted in particular way.
SKIP	Defines from which row to start including the rows in the output.

Sub-clause	Description
LIMIT	Constrains the number of rows in the output.

Reading hints

These comprise clauses used to specify planner hints when tuning a query. More details regarding the usage of these — and query tuning in general — can be found in [Planner hints and the USING keyword](#).

Hint	Description
USING INDEX	Index hints are used to specify which index, if any, the planner should use as a starting point.
USING SCAN	Scan hints are used to force the planner to do a label scan (followed by a filtering operation) instead of using an index.
USING JOIN	Join hints are used to enforce a join operation at specified points.

Writing clauses

These comprise clauses that write the data to the database.

Clause	Description
CREATE	Create nodes and relationships.
DELETE	Delete graph elements — nodes, relationships or paths. Any node to be deleted must also have all associated relationships explicitly deleted.
DETACH DELETE	Delete a node or set of nodes. All associated relationships will automatically be deleted.
SET	Update labels on nodes and properties on nodes and relationships.
REMOVE	Remove properties and labels from nodes and relationships.
FOREACH	Update data within a list, whether components of a path, or the result of aggregation.

Reading/Writing clauses

These comprise clauses that both read data from and write data to the database.

Clause	Description
MERGE	Ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.
--- ON CREATE	Used in conjunction with <code>MERGE</code> , this write sub-clause specifies the actions to take if the pattern needs to be created.
--- ON MATCH	Used in conjunction with <code>MERGE</code> , this write sub-clause specifies the actions to take if the pattern already exists.
CALL [...YIELD]	Invoke a procedure deployed in the database and return any results.
CREATE UNIQUE	A mixture of <code>MATCH</code> and <code>CREATE</code> , matching what it can, and creating what is missing.

Set operations

Clause	Description
UNION	Combines the result of multiple queries into a single result set. Duplicates are removed.
UNION ALL	Combines the result of multiple queries into a single result set. Duplicates are retained.

Importing data

Clause	Description
LOAD CSV	Use when importing data from CSV files.
--- USING PERIODIC COMMIT	This query hint may be used to prevent an out-of-memory error from occurring when importing large amounts of data using LOAD CSV .

Schema clauses

These comprise clauses used to manage the schema; further details can found in [Schema](#).

Clause	Description
CREATE DROP CONSTRAINT	Create or drop an index on all nodes with a particular label and property.
CREATE DROP INDEX	Create or drop a constraint pertaining to either a node label or relationship type, and a property.

7.1. MATCH

The [MATCH clause](#) is used to search for the pattern described in it.

- [Introduction](#)
- [Basic node finding](#)
 - [Get all nodes](#)
 - [Get all nodes with a label](#)
 - [Related nodes](#)
 - [Match with labels](#)
- [Relationship basics](#)
 - [Outgoing relationships](#)
 - [Directed relationships and variable](#)
 - [Match on relationship type](#)
 - [Match on multiple relationship types](#)
 - [Match on relationship type and use a variable](#)
- [Relationships in depth](#)
 - [Relationship types with uncommon characters](#)
 - [Multiple relationships](#)
 - [Variable length relationships](#)
 - [Relationship variable in variable length relationships](#)
 - [Match with properties on a variable length path](#)

- Zero length paths
- Named paths
- Matching on a bound relationship
- Shortest path
 - Single shortest path
 - Single shortest path with predicates
 - All shortest paths
- Get node or relationship by id
 - Node by id
 - Relationship by id
 - Multiple nodes by id

7.1.1. 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 [Patterns](#).

`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, and should not be considered a filter applied only 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 [Indexes](#), and more about specifying hints to force Neo4j to solve a query in a specific way in [Planner hints and the USING keyword](#).



To understand more about the patterns used in the `MATCH` clause, read [Patterns](#)

The following graph is used for the examples below:

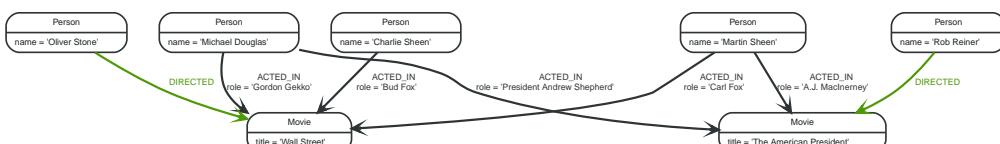


Figure 6. Graph

7.1.2. 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.

Table 38. Result

n
Node[0]{name:"Charlie Sheen"}
Node[1]{name:"Martin Sheen"}
Node[2]{name:"Michael Douglas"}
Node[3]{name:"Oliver Stone"}
Node[4]{name:"Rob Reiner"}
Node[5]{title:"Wall Street"}
Node[6]{title:"The American President"}
7 rows

Get all nodes with a label

Getting all nodes with a label on them is done with a single node pattern where the node has a label on it.

Query

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

Returns all the movies in the database.

Table 39. Result

movie.title
"Wall Street"
"The American President"
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'.

Table 40. 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 (:Person { name: 'Oliver Stone' })--(movie:Movie)
RETURN movie.title
```

Returns any nodes connected with the Person 'Oliver' that are labeled Movie.

Table 41. Result

movie.title
"Wall Street"
1 row

7.1.3. Relationship basics

Outgoing relationships

When the direction of a relationship is of interest, it is shown by using `→` or `←`, like this:

Query

```
MATCH (:Person { name: 'Oliver Stone' })-->(movie)
RETURN movie.title
```

Returns any nodes connected with the Person 'Oliver' by an outgoing relationship.

Table 42. Result

movie.title
"Wall Street"
1 row

Directed relationships and variable

If a variable is required, either for filtering on properties of the relationship, or to return the relationship, this is how you introduce the variable.

Query

```
MATCH (:Person { name: 'Oliver Stone' })-[r]->(movie)
RETURN type(r)
```

Returns the type of each outgoing relationship from 'Oliver'.

Table 43. Result

type(r)
"DIRECTED"
1 row

Match on 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:Movie { title: 'Wall Street' })<[:-ACTED_IN]-(actor)
RETURN actor.name
```

Returns all actors that **ACTED_IN** 'Wall Street'.

Table 44. Result

actor.name
"Michael Douglas"
"Martin Sheen"
"Charlie Sheen"
3 rows

Match on 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.name
```

Returns nodes with an **ACTED_IN** or **DIRECTED** relationship to 'Wall Street'.

Table 45. Result

person.name
"Oliver Stone"
"Michael Douglas"
"Martin Sheen"
"Charlie Sheen"
4 rows

Match on relationship type and use a variable

If you both want to introduce a variable 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.role
```

Returns **ACTED_IN** roles for '**Wall Street**'.

Table 46. Result

r.role
"Gordon Gekko"
"Carl Fox"
"Bud Fox"
3 rows

7.1.4. Relationships in depth



Inside a single pattern, relationships will only be matched once. You can read more about this in [Uniqueness](#).

Relationship types with uncommon characters

Sometimes your database will have types with non-letter characters, or with spaces in them. Use ` (backtick) to quote these. To demonstrate this we can add an additional relationship between '**Charlie Sheen**' and '**Rob Reiner**':

Query

```
MATCH (charlie:Person { name: 'Charlie Sheen' }),(rob:Person { name: 'Rob Reiner' })
CREATE (rob)-[:`TYPE WITH SPACE`]->(charlie)
```

Which leads to the following graph:

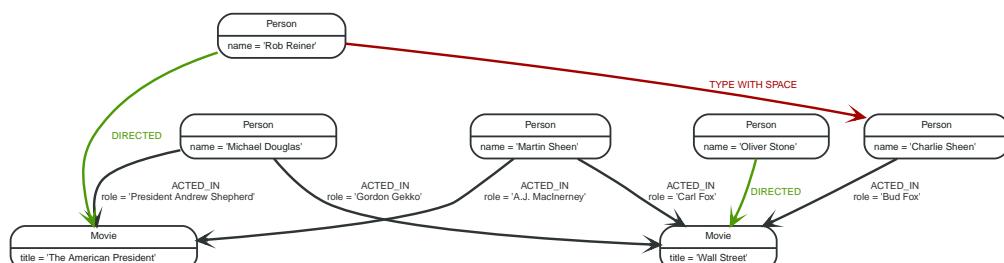


Figure 7. Graph

Query

```
MATCH (n { name: 'Rob Reiner' })-[r:`TYPE WITH SPACE`]->()
RETURN type(r)
```

Returns a relationship type with a space in it

Table 47. Result

type(r)
"TYPE WITH SPACE"

```
type(r)
```

```
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 movie.title, director.name
```

Returns the movie '**Charlie Sheen**' acted in and its director.

Table 48. Result

movie.title	director.name
"Wall Street"	"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. The dots may also be omitted when setting only one bound and this implies a fixed length pattern.

Query

```
MATCH (martin { name: 'Charlie Sheen' })-[:ACTED_IN*1..3]-(movie:Movie)
RETURN movie.title
```

Returns all movies related to '**Charlie Sheen**' by 1 to 3 hops.

Table 49. Result

movie.title
"Wall Street"
"The American President"
"The American President"
3 rows

Relationship variable in variable length relationships

When the connection between two nodes is of variable length, a relationship variable becomes a list of relationships.

Query

```
MATCH (actor { name: 'Charlie Sheen' })-[r:ACTED_IN*2]-(co_actor)
RETURN r
```

Returns a list of relationships.

Table 50. Result

r
[:ACTED_IN[0]{role:"Bud Fox"},:ACTED_IN[1]{role:"Carl Fox"}]
[:ACTED_IN[0]{role:"Bud Fox"},:ACTED_IN[2]{role:"Gordon Gekko"}]
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 father '**Martin Sheen**'. One of them 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:

Query

```
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:

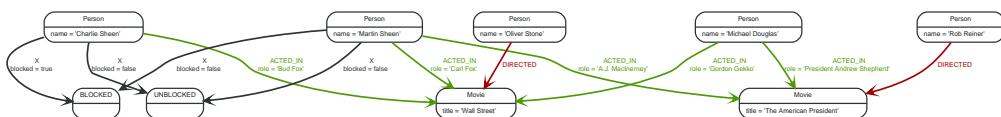


Figure 8. Graph

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 Sheen**' and '**Martin Sheen**' where all relationships have the **blocked** property set to **false**.

Table 51. Result

p
[Node[0]{name:"Charlie Sheen"},:X[7]{blocked:false},Node[7]{},:X[8]{blocked:false},Node[1]{name:"Martin Sheen"}]
1 row

Zero length paths

Using variable length paths that have the lower bound zero means that two variables can point to the same node. If the path length 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 the movie itself as well as actors and directors one relationship away

Table 52. Result

x
Node[5]{title:"Wall Street"}
Node[0]{name:"Charlie Sheen"}
Node[1]{name:"Martin Sheen"}
Node[2]{name:"Michael Douglas"}
Node[3]{name:"Oliver Stone"}
5 rows

Named paths

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 Douglas'**

Table 53. Result

p
[Node[2]{name:"Michael Douglas"},:ACTED_IN[5]{role:"President Andrew Shepherd"},Node[6]{title:"The American President"}]
[Node[2]{name:"Michael Douglas"},:ACTED_IN[2]{role:"Gordon Gekko"},Node[5]{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

Table 54. Result

a	b
Node[0]{name:"Charlie Sheen"}	Node[5]{title:"Wall Street"}
Node[5]{title:"Wall Street"}	Node[0]{name:"Charlie Sheen"}
2 rows	

7.1.5. 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. Within 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. If there is a `WHERE` clause following the match of a `shortestPath`, relevant predicates will be included in the `shortestPath`. If the predicate is a `none()` or `all()` on the relationship elements of the path, it will be used during the search to improve performance (see [Shortest path planning](#)).

Table 55. Result

p
[Node[1]{name: "Martin Sheen"}, :ACTED_IN[1]{role: "Carl Fox"}, Node[5]{title: "Wall Street"}, :DIRECTED[3]{}, Node[3]{name: "Oliver Stone"}]
1 row

Single shortest path with predicates

Predicates used in the `WHERE` clause that apply to the shortest path pattern are evaluated before deciding what the shortest matching path is.

Query

```
MATCH (charlie:Person { name: 'Charlie Sheen' }),(martin:Person { name: 'Martin Sheen' }), p =  
shortestPath((charlie)-[*]-(martin))  
WHERE NONE (r IN relationships(p) WHERE type(r)= 'FATHER')  
RETURN p
```

This query will find the shortest path between '**Charlie Sheen**' and '**Martin Sheen**', and the `WHERE` predicate will ensure that we don't consider the father/son relationship between the two.

Table 56. Result

p
[Node[0]{name: "Charlie Sheen"}, :ACTED_IN[0]{role: "Bud Fox"}, Node[5]{title: "Wall Street"}, :ACTED_IN[1]{role: "Carl Fox"}, Node[1]{name: "Martin Sheen"}]
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 Sheen' and 'Michael Douglas'.

Table 57. Result

p
[Node[1]{name:"Martin Sheen"},:ACTED_IN[1]{role:"Carl Fox"},Node[5]{title:"Wall Street"},:ACTED_IN[2]{role:"Gordon Gekko"},Node[2]{name:"Michael Douglas"}]
[Node[1]{name:"Martin Sheen"},:ACTED_IN[4]{role:"A.J. MacInerney"},Node[6]{title:"The American President"},:ACTED_IN[5]{role:"President Andrew Shepherd"},Node[2]{name:"Michael Douglas"}]
2 rows

7.1.6. Get node or relationship by id

Node by id

Searching for nodes by id can be done with the `id()` function in a predicate.



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. It is therefore recommended to rather use application-generated ids.

Query

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

The corresponding node is returned.

Table 58. Result

n
Node[0]{name:"Charlie Sheen"}
1 row

Relationship by id

Search for relationships by id can be done with the `id()` function in a predicate.

This is not recommended practice. See [Node by id](#) 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.

Table 59. Result

r
:ACTED_IN[0]{role:"Bud Fox"}
1 row

Multiple nodes by id

Multiple nodes are selected by specifying them in an IN clause.

Query

```
MATCH (n)
WHERE id(n) IN [0, 3, 5]
RETURN n
```

This returns the nodes listed in the IN expression.

Table 60. Result

n
Node[0]{name:"Charlie Sheen"}
Node[3]{name:"Oliver Stone"}
Node[5]{title:"Wall Street"}
3 rows

7.2. 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 relationships](#)
- [Properties on optional elements](#)
- [Optional typed and named relationship](#)

7.2.1. 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 a null 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.



To understand the patterns used in the OPTIONAL MATCH clause, read [Patterns](#).

The following graph is used for the examples below:

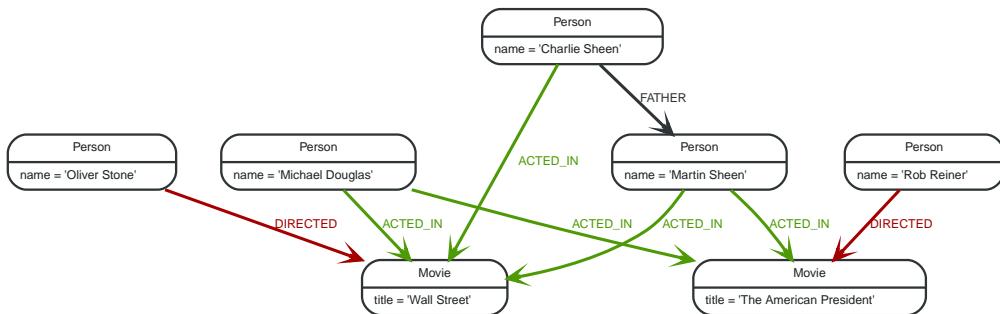


Figure 9. Graph

7.2.2. Optional relationships

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.

Table 61. Result

x
<null>
1 row

7.2.3. 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.

Table 62. Result

x	x.name
<null>	<null>
1 row	

7.2.4. Optional typed and named relationship

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

Query

```
MATCH (a:Movie { title: 'Wall Street' })
OPTIONAL MATCH (a)-[r:ACTS_IN]->()
RETURN a.title, r
```

This returns the title of the node, 'Wall Street', and, since the node has no outgoing ACTS_IN relationships, `null` is returned for the relationship denoted by `r`.

Table 63. Result

a.title	r
"Wall Street"	<null>
1 row	

7.3. START

Find starting points through legacy indexes.



The `START` clause should only be used when accessing legacy indexes. In all other cases, use `MATCH` instead (see [MATCH](#)).

- [Node by index seek](#)
- [Relationship by index seek](#)
- [Node by index query](#)

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:

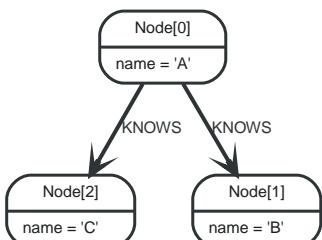


Figure 10. Graph

7.3.1. 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
```

7.4. RETURN

The `RETURN` clause defines what to include in the query result set.

- [Introduction](#)

- Return nodes
- Return relationships
- Return property
- Return all elements
- Variable with uncommon characters
- Column alias
- Optional properties
- Other expressions
- Unique results

7.4.1. Introduction

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.



If what you actually want is the value of a property, make sure to not return the full node/relationship. This will improve performance.

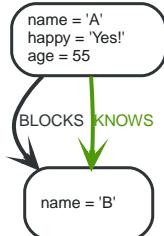


Figure 11. Graph

7.4.2. 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.

Table 64. Result

n
Node[1]{name: "B"}
1 row

7.4.3. 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.

Table 65. Result

r
:KNOWS[0]{}
1 row

7.4.4. 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.

Table 66. Result

n.name
"A"
1 row

7.4.5. 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.

Table 67. 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			

7.4.6. Variable with uncommon characters

To introduce a placeholder that is made up of characters that are not contained in the English alphabet, you can use the ` to enclose the variable, like this:

Query

```
MATCH (`This isn't a common variable`)
WHERE `This isn't a common variable`.name = 'A'
RETURN `This isn't a common variable`.happy
```

The node with name "A" is returned.

Table 68. Result

`This isn't a common variable`.happy

"Yes!"

1 row

7.4.7. 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.

Table 69. Result

SomethingTotallyDifferent

55

1 row

7.4.8. 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.

Table 70. Result

n.age

55

<null>

2 rows

7.4.9. 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.

Table 71. Result

a.age > 30	"I'm a literal"	(a)-->()
true	"I'm a literal"	<code>[[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"}]]</code>
1 row		

7.4.10. 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.

Table 72. Result

b
<code>Node[1]{name: "B"}</code>
1 row

7.5. 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.

- [Introduction](#)
- [Filter on aggregate function results](#)
- [Sort results before using collect on them](#)
- [Limit branching of a path search](#)

7.5.1. Introduction

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 the 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.

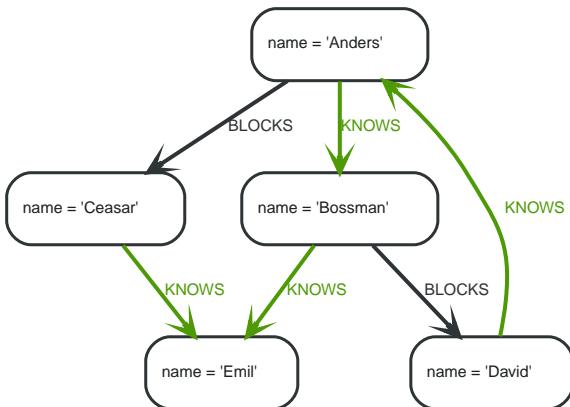


Figure 12. Graph

7.5.2. 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.name
```

The name of the person connected to '**David**' with at least more than one outgoing relationship will be returned by the query.

Table 73. Result

otherPerson.name
"Anders"
1 row

7.5.3. Sort results before using collect on them

You can sort your results before passing them to collect, thus sorting the resulting list.

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, is returned in a list.

Table 74. Result

collect(n.name)
["Emil", "David", "Ceasar"]
1 row

7.5.4. Limit branching of a 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.

Table 75. Result

o.name
"Bossman"
"Anders"
2 rows

7.6. UNWIND

UNWIND expands a list into a sequence of rows.

- [Introduction](#)
- [Unwind a list](#)
- [Create a distinct list](#)
- [Create nodes from a list parameter](#)

7.6.1. Introduction

With **UNWIND**, you can transform any list back into individual rows. These lists can be parameters that were passed in, previously `collect`-ed result or other list expressions.

One common usage of unwind is to create distinct lists. Another is to create data from parameter lists that are provided to the query.

UNWIND requires you to specify a new name for the inner values.

7.6.2. Unwind a list

We want to transform the literal list into rows named `x` and return them.

Query

```
UNWIND [1, 2, 3] AS x
RETURN x
```

Each value of the original list is returned as an individual row.

Result

```
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
3 rows
```

7.6.3. Create a distinct list

We want to transform a list 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 list is unwound and passed through `DISTINCT` to create a unique set.

Result

```
+-----+
| set   |
+-----+
| [1,2] |
+-----+
1 row
```

7.6.4. Create nodes from a list 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 list 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
```

7.7. WHERE

`WHERE` adds constraints to the patterns in a `MATCH` or `OPTIONAL MATCH` clause or filters the results of a `WITH` clause.

- [Introduction](#)
- [Basic usage](#)
 - Boolean operations
 - Filter on node label
 - Filter on node property
 - Filter on relationship property
 - Filter on dynamically-computed property
 - Property existence checking
- [String matching](#)
 - Match the beginning of a string
 - Match the ending of a string
 - Match anywhere within a string
 - String matching negation
- [Regular expressions](#)
 - Matching using regular expressions
 - Escaping in regular expressions
 - Case-insensitive regular expressions
- [Using path patterns in WHERE](#)
 - Filter on patterns
 - Filter on patterns using `NOT`
 - Filter on patterns with properties
 - Filter on relationship type
- [Lists](#)
 - `IN` operator

- Missing properties and values
 - Default to `false` if property is missing
 - Default to `true` if property is missing
 - Filter on `null`
- Using ranges
 - Simple range
 - Composite range

7.7.1. Introduction

`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.*



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.

The following graph is used for the examples below:

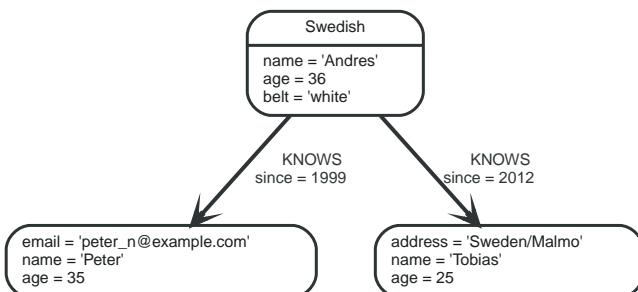


Figure 13. Graph

7.7.2. Basic usage

Boolean operations

You can use the boolean operators `AND`, `OR`, `XOR` and `NOT`. See [Working with null](#) 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.name, n.age
  
```

Table 76. Result

n.name	n.age
"Andres"	36

n.name	n.age
"Tobias"	25
"Peter"	35
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.name, n.age
```

The name and age for the 'Andres' node will be returned.

Table 77. Result

n.name	n.age
"Andres"	36
1 row	

Filter on node property

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

Query

```
MATCH (n)
WHERE n.age < 30
RETURN n.name, n.age
```

The name and age values for the 'Tobias' node are returned because he is less than 30 years of age.

Table 78. Result

n.name	n.age
"Tobias"	25
1 row	

Filter on relationship property

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

Query

```
MATCH (n)-[k:KNOWS]->(f)
WHERE k.since < 2000
RETURN f.name, f.age, f.email
```

The name, age and email values for the 'Peter' node are returned because Andrés has known him since before 2000.

Table 79. Result

f.name	f.age	f.email
"Peter"	35	"peter_n@example.com"
1 row		

Filter on dynamically-computed node property

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

Query

```
WITH 'AGE' AS propname
MATCH (n)
WHERE n[toLowerCase(propname)] < 30
RETURN n.name, n.age
```

The name and age values for the 'Tobias' node are returned because he is less than 30 years of age.

Table 80. Result

n.name	n.age
"Tobias"	25
1 row	

Property existence checking

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

Query

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

The name and belt for the 'Andres' node are returned because he is the only one with a `belt` property.



The `has()` function has been superseded by `exists()` and has been removed.

Table 81. Result

n.name	n.belt
"Andres"	"white"
1 row	

7.7.3. 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 beginning 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.name, n.age
```

The name and age for the 'Peter' node are returned because his name starts with 'Pet'.

Table 82. Result

n.name	n.age
"Peter"	35
1 row	

Match the ending 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.name, n.age
```

The name and age for the 'Peter' node are returned because his name ends with 'ter'.

Table 83. Result

n.name	n.age
"Peter"	35
1 row	

Match anywhere within 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.name, n.age
```

The name and age for the 'Peter' node are returned because his name contains with 'ete'.

Table 84. Result

n.name	n.age
"Peter"	35
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.name, n.age
```

The name and age for the 'Peter' node are returned because his name does not end with 's'.

Table 85. Result

n.name	n.age
"Peter"	35
1 row	

7.7.4. Regular expressions

Cypher supports filtering using regular expressions. The regular expression syntax is inherited from [the Java regular expressions](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html) (<https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>). 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 beginning of the regular expression, for example `MATCH (n) WHERE n.name =~ '(?i)Lon.*'` will return nodes with name 'London' or with name 'LonDoN'.

Matching using regular expressions

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

Query

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

The name and age for the 'Tobias' node are returned because his name starts with 'Tob'.

Table 86. Result

n.name	n.age
"Tobias"	25
1 row	

Escaping in regular expressions

If you need a forward slash within your regular expression, escape it. Remember that the backslash needs to be escaped in string literals.

Query

```
MATCH (n)
WHERE n.address =~ 'Sweden\\Malmo'
RETURN n.name, n.age, n.address
```

The name, age and address for the 'Tobias' node are returned because his address is in 'Sweden/Malmo'.

Table 87. Result

n.name	n.age	n.address
"Tobias"	25	"Sweden/Malmo"
1 row		

Case-insensitive regular expressions

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

Query

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

The name and age for the 'Andres' node are returned because his name starts with 'ANDR' irrespective of casing.

Table 88. Result

n.name	n.age
"Andres"	36
1 row	

7.7.5. Using path patterns in WHERE

Filter on patterns

Patterns are expressions in Cypher, expressions that return a list of paths. List expressions are also predicates — an empty list 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 cannot 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 cannot introduce new variables 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`, whereas 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.name, others.age
```

The name and age for nodes that have an outgoing relationship to the 'Tobias' node are returned.

Table 89. Result

others.name	others.age
"Andres"	36
1 row	

Filter on patterns using NOT

The **NOT** operator can be used to exclude a pattern.

Query

```
MATCH (persons),(peter { name: 'Peter' })
WHERE NOT (persons)-->(peter)
RETURN persons.name, persons.age
```

Name and age values for nodes that do not have an outgoing relationship to the '**Peter**' node are returned.

Table 90. Result

persons.name	persons.age
"Tobias"	25
"Peter"	35
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.name, n.age
```

Finds all name and age values for nodes that have a **KNOWS** relationship to a node with the name '**Tobias**'.

Table 91. Result

n.name	n.age
"Andres"	36
1 row	

Filter 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 type(r), r.since
```

This returns all relationships having a type whose name starts with 'K'.

Table 92. Result

type(r)	r.since
"KNOWS"	1999
"KNOWS"	2012
2 rows	

7.7.6. Lists

IN operator

To check if an element exists in a list, you can use the `IN` operator.

Query

```
MATCH (a)
WHERE a.name IN ['Peter', 'Tobias']
RETURN a.name, a.age
```

This query shows how to check if a property exists in a literal list.

Table 93. Result

a.name	a.age
"Tobias"	25
"Peter"	35
2 rows	

7.7.7. 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.name, n.age, n.belt
```

Only the name, age and belt values of nodes with white belts are returned.

Table 94. Result

n.name	n.age	n.belt
"Andres"	36	"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.name, n.age, n.belt
ORDER BY n.name
```

This returns all values for all nodes, even those without the belt property.

Table 95. Result

n.name	n.age	n.belt
"Andres"	36	"white"
"Peter"	35	<null>
"Tobias"	25	<null>
3 rows		

Filter on `null`

Sometimes you might want to test if a value or a variable is `null`. This is done just like SQL does it, using `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.name, person.age, person.belt
```

The name and age values for nodes that have name 'Peter' but no belt property are returned.

Table 96. Result

person.name	person.age	person.belt
"Peter"	35	<null>
1 row		

7.7.8. 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.name, a.age
```

The name and age values of nodes having a name property lexicographically greater than or equal to 'Peter' are returned.

Table 97. Result

a.name	a.age
"Tobias"	25
"Peter"	35
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.name, a.age
```

The name and age values of nodes having a name property lexicographically between 'Andres' and 'Tobias' are returned.

Table 98. Result

a.name	a.age
"Peter"	35
1 row	

7.8. ORDER BY

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

- [Introduction](#)
- [Order nodes by property](#)
- [Order nodes by multiple properties](#)
- [Order nodes in descending order](#)
- [Ordering null](#)

7.8.1. Introduction

Note that you cannot sort on nodes or relationships, just on properties on these. `ORDER BY` relies on comparisons to sort the output, see [Ordering and comparison of values](#).

In terms of scope of variables, `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 variables available in the projection are available. If the projection does not alter the output cardinality (which aggregation and `DISTINCT` do), variables available from before the projecting clause are also available. When the projection clause shadows already existing variables, only the new variables 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.

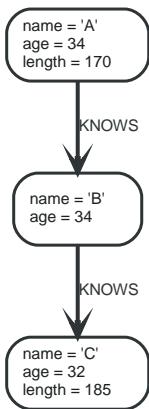


Figure 14. Graph

7.8.2. Order nodes by property

`ORDER BY` is used to sort the output.

Query

```

MATCH (n)
RETURN n.name, n.age
ORDER BY n.name

```

The nodes are returned, sorted by their name.

Table 99. Result

n.name	n.age
"A"	34
"B"	34
"C"	32
3 rows	

7.8.3. Order nodes by multiple properties

You can order by multiple properties by stating each variable in the `ORDER BY` clause. Cypher will sort the result by the first variable listed, and for equals values, go to the next property in the `ORDER BY` clause, and so on.

Query

```

MATCH (n)
RETURN n.name, n.age
ORDER BY n.age, n.name

```

This returns the nodes, sorted first by their age, and then by their name.

Table 100. Result

n.name	n.age
"C"	32
"A"	34
"B"	34
3 rows	

7.8.4. Order nodes in descending order

By adding `DESC[ENDING]` after the variable to sort on, the sort will be done in reverse order.

Query

```
MATCH (n)
RETURN n.name, n.age
ORDER BY n.name DESC
```

The example returns the nodes, sorted by their name in reverse order.

Table 101. Result

n.name	n.age
"C"	32
"B"	34
"A"	34
3 rows	

7.8.5. 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.name, n.age
ORDER BY n.length
```

The nodes are returned sorted by the length property, with a node without that property last.

Table 102. Result

n.length	n.name	n.age
170	"A"	34
185	"C"	32
<null>	"B"	34
3 rows		

7.9. SKIP

`SKIP` defines from which row to start including the rows in the output.

- [Introduction](#)
- [Skip first three rows](#)
- [Return middle two rows](#)
- [Using an expression with `SKIP` to return a subset of the rows](#)

7.9.1. Introduction

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.

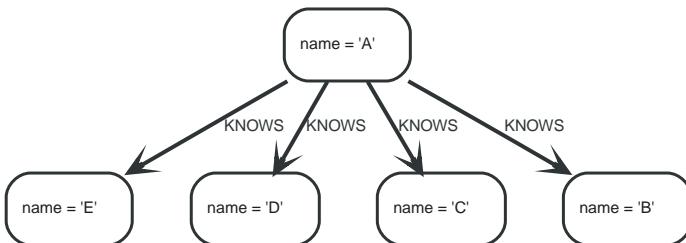


Figure 15. Graph

7.9.2. Skip first three rows

To return a subset of the result, starting from the fourth result, use the following syntax:

Query

```
MATCH (n)
RETURN n.name
ORDER BY n.name
SKIP 3
```

The first three nodes are skipped, and only the last two are returned in the result.

Table 103. Result

n.name
"D"
"E"
2 rows

7.9.3. Return middle two rows

To return a subset of the result, starting from somewhere in the middle, use this syntax:

Query

```
MATCH (n)
RETURN n.name
ORDER BY n.name
SKIP 1
LIMIT 2
```

Two nodes from the middle are returned.

Table 104. Result

n.name
"B"
"C"
2 rows

7.9.4. Using an expression with **SKIP** to return a subset of the rows

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

Query

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

Table 105. Result

n.name
"B"
"C"
"D"
"E"
4 rows

7.10. LIMIT

LIMIT constrains the number of rows in the output.

- [Introduction](#)
- [Return a subset of the rows](#)
- [Using an expression with **LIMIT** to return a subset of the rows](#)

7.10.1. Introduction

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

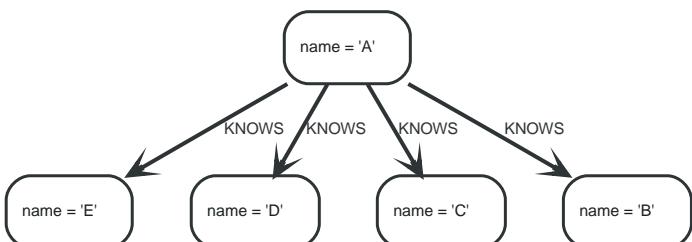


Figure 16. Graph

7.10.2. Return a subset of the rows

To return a subset of the result, starting from the top, use this syntax:

Query

```
MATCH (n)
RETURN n.name
ORDER BY n.name
LIMIT 3
```

The top three items are returned by the example query.

Table 106. Result

n.name
"A"
"B"
"C"
3 rows

7.10.3. Using an expression with LIMIT to return a subset of the rows

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

Query

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

Returns one to three top items.

Table 107. Result

n.name
"A"
"B"
"C"
3 rows

7.11. CREATE

The **CREATE** clause is used to create graph elements — nodes and relationships.

- [Create nodes](#)
 - [Create single node](#)
 - [Create multiple nodes](#)
 - [Create a node with a label](#)
 - [Create a node with multiple labels](#)
 - [Create node and add labels and properties](#)
 - [Return created node](#)

- Create relationships
 - Create a relationship between two nodes
 - Create a relationship and set properties
- Create a full path
- Use parameters with `CREATE`
 - Create node with a parameter for the properties
 - Create multiple nodes with a parameter for their properties



In the `CREATE` clause, patterns are used a lot. Read [Patterns](#) for an introduction.

7.11.1. 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

No data returned.
Nodes created: 1

Create multiple nodes

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

Query

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

Result

No data returned.
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

```
+-----+  
| No data returned. |  
+-----+  
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

```
+-----+  
| No data returned. |  
+-----+  
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

```
+-----+  
| No data returned. |  
+-----+  
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
```

7.11.2. 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
```

7.11.3. 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 variable, 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
```

7.11.4. 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

```
+-----+  
| No data returned. |  
+-----+  
Nodes created: 2  
Properties set: 4
```

7.12. DELETE

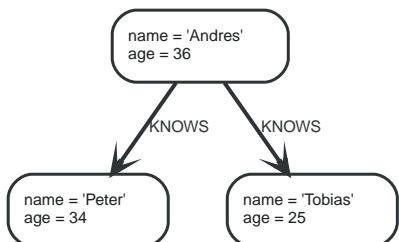
The **DELETE** clause is used to delete graph elements — nodes, relationships or paths.

- [Introduction](#)
- [Delete a single node](#)
- [Delete all nodes and relationships](#)
- [Delete a node with all its relationships](#)
- [Delete relationships only](#)

7.12.1. Introduction

For removing properties and labels, see [REMOVE](#). Remember that you cannot 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:



7.12.2. Delete single node

To delete a node, use the **DELETE** clause.

Query

```
MATCH (n:Useless)
DELETE n
```

Result

```
+-----+
| No data returned. |
+-----+
Nodes deleted: 1
```

7.12.3. 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

```
+-----+
| No data returned. |
+-----+
Nodes deleted: 3
Relationships deleted: 2
```

7.12.4. 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

```
+-----+
| No data returned. |
+-----+
Nodes deleted: 1
Relationships deleted: 2
```

7.12.5. Delete relationships only

It is also possible to delete relationships only, leaving the node(s) otherwise unaffected.

Query

```
MATCH (n { name: 'Andres' })-[r:KNOWS]->()
DELETE r
```

This deletes all outgoing **KNOWS** relationships from the node with the name '**Andres**'.

Result

```
+-----+
| No data returned. |
+-----+
Relationships deleted: 2
```

7.13. SET

The **SET** clause is used to update labels on nodes and properties on nodes and relationships.

- [Introduction](#)
- [Set a property](#)
- [Remove a property](#)
- [Copying properties between nodes and relationships](#)
- [Adding properties from maps](#)
- [Set a property using a parameter](#)
- [Set all properties using a parameter](#)
- [Set multiple properties using one **SET** clause](#)
- [Set a label on a node](#)
- [Set multiple labels on a node](#)

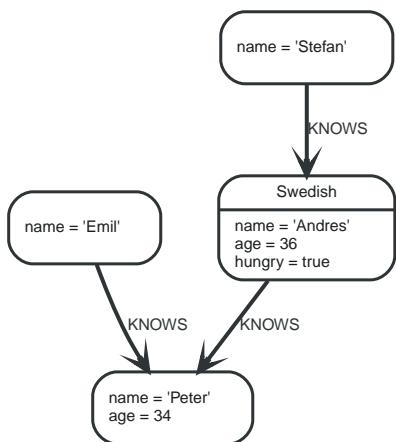
7.13.1. Introduction

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



Setting labels on a node is an idempotent operations — if you try to set a label on a node that already has that label on it, nothing happens. The query statistics will tell you if something needed to be done or not.

The examples use this graph as a starting point:



7.13.2. 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[0]{surname:"Taylor",name:"Andres",age:36,hungry:true} |
+-----+
1 row
Properties set: 1
```

7.13.3. 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[0]{hungry:true,age:36} |
+-----+
1 row
Properties set: 1
```

7.13.4. 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[0]{name:"Peter",age:34} | Node[2]{name:"Peter",age:34} |  
+-----+  
1 row  
Properties set: 3
```

7.13.5. 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

```
+-----+  
| No data returned. |  
+-----+  
Properties set: 2
```

7.13.6. 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 a surname added.

Result

```
+-----+  
| n |  
+-----+  
| Node[0]{surname:"Taylor",name:"Andres",age:36,hungry:true} |  
+-----+  
1 row  
Properties set: 1
```

7.13.7. 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 its properties replaced by the properties in the **props** parameter.

Result

```
+-----+  
| n |  
+-----+  
| Node[0]{name:"Andres",position:"Developer"} |  
+-----+  
1 row  
Properties set: 4
```

7.13.8. 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

```
+-----+  
| No data returned. |  
+-----+  
Properties set: 2
```

7.13.9. 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[3]{name:"Stefan"} |  
+-----+  
1 row  
Labels added: 1
```

7.13.10. 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[1]{name:"Emil"} |  
+-----+  
1 row  
Labels added: 2
```

7.14. REMOVE

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

- [Introduction](#)
- [Remove a property](#)
- [Remove a label from a node](#)
- [Removing multiple labels](#)

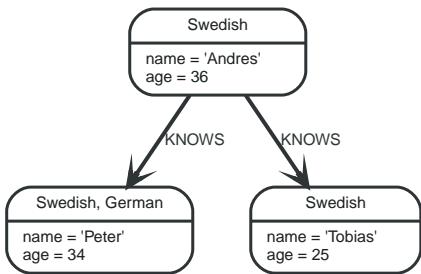
7.14.1. Introduction

For deleting nodes and relationships, see [DELETE](#).



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 use the following database:



7.14.2. 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[0]{name:"Andres"} |
+-----+
1 row
Properties set: 1

```

7.14.3. 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[2]{name:"Peter",age:34} |
+-----+
1 row
Labels removed: 1

```

7.14.4. Removing multiple labels

To remove multiple labels, you use `REMOVE`.

Query

```
MATCH (n { name: 'Peter' })
REMOVE n:German:Swedish
RETURN n
```

Result

```
+-----+
| n
+-----+
| Node[2]{name:"Peter",age:34} |
+-----+
1 row
Labels removed: 2
```

7.15. FOREACH

The **FOREACH** clause is used to update data within a list, whether components of a path, or result of aggregation.

- [Introduction](#)
- [Mark all nodes along a path](#)

7.15.1. Introduction

Lists 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 path, or a list created by aggregation.

The variable context within the foreach parenthesis is separate from the one outside it. This means that if you **CREATE** a node variable within a **FOREACH**, you will *not* be able to use it outside of the foreach statement, unless you match to find it.

Within 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 list then **UNWIND** (see **UNWIND**) would be a more appropriate command.

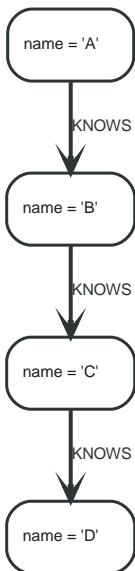


Figure 17. Data for the examples

7.15.2. 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

```

+-----+
| No data returned. |
+-----+
Properties set: 4

```

7.16. 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 nodes](#)
 - Merge single node with a label
 - Merge single node with properties
 - Merge single node specifying both label and property
 - Merge single node derived from an existing node property
- [Use ON CREATE and ON MATCH](#)
 - Merge with `ON CREATE`
 - Merge with `ON MATCH`

- Merge with `ON CREATE` and `ON MATCH`
- Merge with `ON MATCH` setting multiple properties
- Merge relationships
 - Merge on a relationship
 - Merge on multiple relationships
 - Merge on an undirected relationship
 - Merge on a relationship between two existing nodes
 - Merge on a relationship between an existing node and a merged node derived from a node property
- Using unique constraints with `MERGE`
 - Merge using unique constraints creates a new node if no node is found
 - Merge using unique constraints matches an existing node
 - Merge with unique constraints and partial matches
 - Merge with unique constraints and conflicting matches
- Using map parameters with `MERGE`

7.16.1. Introduction

`MERGE` either matches existing nodes and binds them, or it creates new data and binds that. It's like a combination of `MATCH` and `CREATE` that additionally allows you to specify what happens if the data was matched or created.

For example, you can specify that the graph must contain a node for a user with a certain name. If there isn't a node with the correct name, a new node will be created and its name property set.

When using `MERGE` on full patterns, the behavior is that either the whole pattern matches, or the whole pattern is created. `MERGE` will not partially use existing patterns — it's all or nothing. If partial matches are needed, this can be accomplished by splitting a pattern up into multiple `MERGE` clauses.

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

The last part of `MERGE` is the `ON CREATE` and `ON MATCH`. These allow a query to express additional changes to the properties of a node or relationship, depending on if the element was `MATCH`-ed in the database or if it was `CREATE`-ed.

The rule planner (see [How are queries executed?](#)) expands a `MERGE` pattern from the end point that has the variable 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 variables could improve performance.

The following graph is used for the examples below:

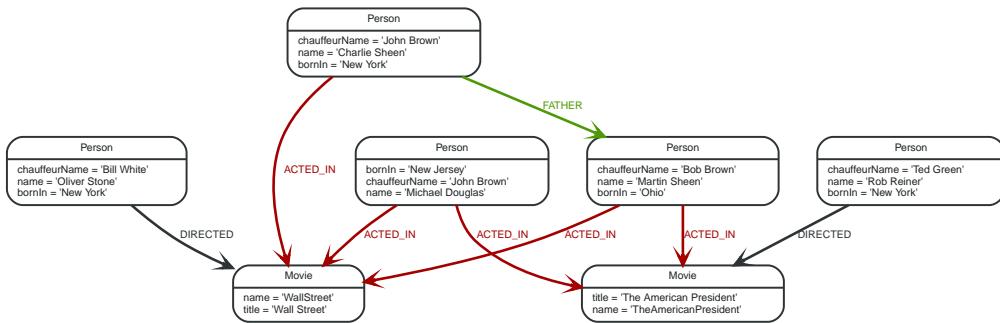


Figure 18. Graph

7.16.2. 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           |
+-----+
| "Rob Reiner"    | "New York"   | Node[7]{name:"New York"} |
| "Oliver Stone"  | "New York"   | Node[7]{name:"New York"} |
| "Charlie Sheen" | "New York"   | Node[7]{name:"New York"} |
| "Michael Douglas" | "New Jersey" | Node[8]{name:"New Jersey"} |
| "Martin Sheen"   | "Ohio"        | Node[9]{name:"Ohio"}    |
+-----+
5 rows
Nodes created: 3
Properties set: 3
Labels added: 3
```

7.16.3. 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" | 1500055180779 |
+-----+
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 |
+-----+
| "Rob Reiner"    | true
| "Oliver Stone"  | true
| "Charlie Sheen" | true
| "Michael Douglas" | true
| "Martin Sheen"  | 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" | 1500055187160 | <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 |  
+-----+  
| "Rob Reiner" | true | 1500055184886 |  
| "Oliver Stone" | true | 1500055184886 |  
| "Charlie Sheen" | true | 1500055184886 |  
| "Michael Douglas" | true | 1500055184886 |  
| "Martin Sheen" | true | 1500055184886 |  
+-----+  
5 rows  
Properties set: 10
```

7.16.4. 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 [Merge single node derived from an existing node property](#). 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
"Rob Reiner"	"New York"	Node[7]{name:"New York"}
"Oliver Stone"	"New York"	Node[7]{name:"New York"}
"Charlie Sheen"	"New York"	Node[7]{name:"New York"}
"Michael Douglas"	"New Jersey"	Node[8]{name:"New Jersey"}
"Martin Sheen"	"Ohio"	Node[9]{name:"Ohio"}

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 [Merge on a relationship between two existing nodes](#), 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
"Rob Reiner"	"Ted Green"	Node[7]{name:"Ted Green"}
"Oliver Stone"	"Bill White"	Node[8]{name:"Bill White"}
"Charlie Sheen"	"John Brown"	Node[9]{name:"John Brown"}
"Michael Douglas"	"John Brown"	Node[10]{name:"John Brown"}
"Martin Sheen"	"Bob Brown"	Node[11]{name:"Bob Brown"}

5 rows

Nodes created: 5

Relationships created: 5

Properties set: 5

Labels added: 5

7.16.5. 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)`, 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"

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 michael and can not create a new node due to
conflicts with existing unique nodes
```

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 oliver and can not create a new node due to
conflicts with existing unique nodes
```

7.16.6. Using map parameters with **MERGE**

MERGE does not support map parameters the same way **CREATE** does. To use map parameters with **MERGE**, it is necessary to explicitly use the expected properties, such as in the following example. For more information on parameters, see [Parameters](#).

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
```

7.17. CALL[...YIELD]

The `CALL` clause is used to call a procedure deployed in the database.

- [Introduction](#)
- [Call a procedure using `CALL`](#)
- [Call a procedure using a quoted namespace and name](#)
- [Call a procedure with literal arguments](#)
- [Call a procedure with parameter arguments](#)
- [Call a procedure with mixed literal and parameter arguments](#)
- [Call a procedure with literal and default arguments](#)
- [Call a procedure within a complex query using `CALL...YIELD`](#)
- [Call a procedure and filter its results](#)
- [Call a procedure within a complex query and rename its outputs](#)

7.17.1. Introduction

Procedures are called using the `CALL` clause.

Each procedure call needs to specify all required procedure arguments. This may be done either explicitly, by using a comma-separated list wrapped in parentheses after the procedure name, or implicitly by using available query parameters as procedure call arguments. The latter form is available only in a so-called standalone procedure call, when the whole query consists of a single `CALL` clause.

Most procedures return a stream of records with a fixed set of result fields, similar to how running a Cypher query returns a stream of records. The `YIELD` sub-clause is used to explicitly select which of the available result fields are returned as newly-bound variables from the procedure call to the user or for further processing by the remaining query. Each yielded result field may optionally be renamed using

aliasing (i.e. `resultFieldName AS newName`). All new variables bound by a procedure call are added to the set of variables already bound in the current scope. It is an error if a procedure call tries to rebind a previously bound variable (i.e. a procedure call cannot shadow a variable that was previously bound in the current scope).

Inside a larger query, the records returned from a procedure call with an explicit `YIELD` may be further filtered using a `WHERE` sub-clause followed by a predicate (similar to `WITH ... WHERE ...`).

If the called procedure declares at least one result field, `YIELD` may generally not be omitted. However `YIELD` may always be omitted in a standalone procedure call. In this case, all result fields are yielded as newly-bound variables from the procedure call to the user.

Neo4j supports the notion of `VOID` procedures. A `VOID` procedure is a procedure that does not declare any result fields and returns no result records and that has explicitly been declared as `VOID`. Calling a `VOID` procedure may only have a side effect and thus does neither allow nor require the use of `YIELD`. Calling a `VOID` procedure in the middle of a larger query will simply pass on each input record (i.e. it acts like `WITH *` in terms of the record stream).

The following examples show how to pass arguments to and yield result fields from a procedure call. All examples use the following procedure:

```
public class IndexingProcedure
{
    @Context
    public GraphDatabaseService db;

    /**
     * Adds a node to a named legacy index. Useful to, for instance, update
     * a full-text index through cypher.
     * @param indexName the name of the index in question
     * @param nodeId id of the node to add to the index
     * @param propKey property to index (value is read from the node)
     */
    @Procedure(mode = Mode.WRITE)
    public void addNodeToIndex( @Name("indexName") String indexName,
                               @Name("node") long nodeId,
                               @Name( value = "propKey", defaultValue = "name" ) String propKey )
    {
        Node node = db.getNodeById( nodeId );
        db.index()
            .forNodes( indexName )
            .add( node, propKey, node.getProperty( propKey ) );
    }
}
```



This clause cannot be combined with other clauses.

Unresolved directive in neo4j-manual-contents/3.2.3/docs/neo4j-cypher-docs-docs-jar/dev/ql/call/index.asciidoc - include::call-a-procedure.asciidoc[]

7.17.2. Call a procedure using a quoted namespace and name

This calls the built-in procedure `db.labels`, which lists all labels used in the database.

Query

```
CALL `db`.`labels`
```

Result

```
+-----+  
| label |  
+-----+  
| "User" |  
| "Administrator" |  
+-----+  
2 rows
```

7.17.3. Call a procedure with literal arguments

This calls the example procedure `org.neo4j.procedure.example.addNodeToIndex` using literal arguments, i.e. arguments that are written out directly in the statement text.

Query

```
CALL org.neo4j.procedure.example.addNodeToIndex('users', 0, 'name')
```

Since our example procedure does not return any result, the result is empty.

Result

```
+-----+  
| No data returned, and nothing was changed. |  
+-----+
```

7.17.4. Call a procedure with parameter arguments

This calls the example procedure `org.neo4j.procedure.example.addNodeToIndex` using parameters as arguments. Each procedure argument is taken to be the value of a corresponding statement parameters with the same name (or null if no such parameter has been given).

Parameters

```
{  
    "indexName" : "users",  
    "node" : 0,  
    "propKey" : "name"  
}
```

Query

```
CALL org.neo4j.procedure.example.addNodeToIndex
```

Since our example procedure does not return any result, the result is empty.

Result

```
+-----+  
| No data returned, and nothing was changed. |  
+-----+
```

7.17.5. Call a procedure with mixed literal and parameter arguments

This calls the example procedure `org.neo4j.procedure.example.addNodeToIndex` using both literal and parameter arguments.

Parameters

```
{  
    "node" : 0  
}
```

Query

```
CALL org.neo4j.procedure.example.addNodeToIndex('users', $node, 'name')
```

Since our example procedure does not return any result, the result is empty.

Result

```
+-----+  
| No data returned, and nothing was changed. |  
+-----+
```

7.17.6. Call a procedure with literal and default arguments

This calls the example procedure `org.neo4j.procedure.example.addNodeToIndex` using literal arguments, i.e. arguments that are written out directly in the statement text, and a trailing default argument that is provided by the procedure itself.

Query

```
CALL org.neo4j.procedure.example.addNodeToIndex('users', 0)
```

Since our example procedure does not return any result, the result is empty.

Result

```
+-----+  
| No data returned, and nothing was changed. |  
+-----+
```

Unresolved directive in neo4j-manual-contents/3.2.3/docs/neo4j-cypher-docs-docs-jar/dev/ql/call/index.asciidoc - include::call-a-procedure-within-a-complex-query.asciidoc[]

7.17.7. Call a procedure and filter its results

This calls the built-in procedure `db.labels` to count all in-use labels in the database that contain the word 'User'

Query

```
CALL db.labels() YIELD label  
WHERE label CONTAINS 'User'  
RETURN count(label) AS numLabels
```

Since the procedure call is part of a larger query, all outputs must be named explicitly.

Result

numLabels
1

1 row

7.17.8. Call a procedure within a complex query and rename its outputs

This calls the built-in procedure `db.propertyKeys` as part of counting the number of nodes per property key that is currently used in the database.

Query

```
CALL db.propertyKeys() YIELD propertyKey AS prop
MATCH (n)
WHERE n[prop] IS NOT NULL RETURN prop, count(n) AS numNodes
```

Since the procedure call is part of a larger query, all outputs must be named explicitly.

Result

prop	numNodes
"name"	1

1 row

7.18. 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](#)
- [Create unique nodes](#)
 - [Create node if missing](#)
 - [Create nodes with values](#)
 - [Create labeled node if missing](#)
- [Create unique relationships](#)
 - [Create relationship if it is missing](#)
 - [Create relationship with values](#)
- [Describe complex pattern](#)

7.18.1. Introduction



`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.

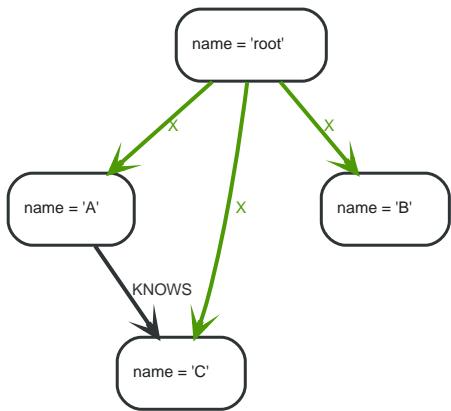
`CREATE UNIQUE` is in the middle of `MATCH` and `CREATE` — it will match what it can, and create what is missing. `CREATE UNIQUE` will always make the least change possible to the graph — if it can use parts of the existing graph, it will.

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.



In the `CREATE UNIQUE` clause, patterns are used a lot. Read [Patterns](#) for an introduction.

The examples use the following graph:



7.18.2. 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
```

7.18.3. 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
```

7.18.4. 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
```

7.19. UNION

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

- [Introduction](#)
- [Combine two queries and retain duplicates](#)
- [Combine two queries and remove duplicates](#)

7.19.1. Introduction

UNION 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.

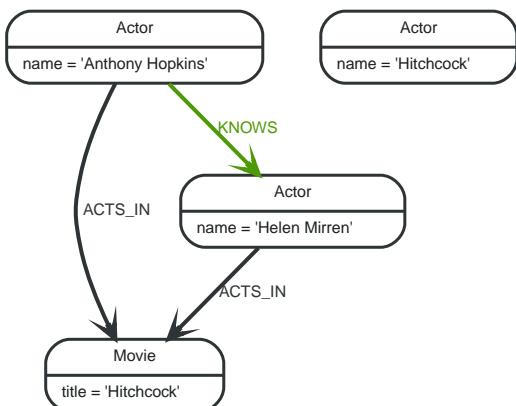


Figure 19. Graph

7.19.2. Combine two queries and retain duplicates

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.

Table 108. Result

name
"Anthony Hopkins"
"Helen Mirren"
"Hitchcock"
"Hitchcock"
4 rows

7.19.3. 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.

Table 109. Result

name
"Anthony Hopkins"
"Helen Mirren"
"Hitchcock"
3 rows

7.20. LOAD CSV

`LOAD CSV` is used to import data from CSV files.

- [Introduction](#)
- [CSV file format](#)
- [Import data from a CSV file](#)
- [Import data from a CSV file containing headers](#)
- [Import data from a CSV file with a custom field delimiter](#)
- [Importing large amounts of data](#)
- [Setting the rate of periodic commits](#)
- [Import data containing escaped characters](#)

7.20.1. Introduction

- 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 a variable 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*.
- `LOAD CSV` is often used in conjunction with the query hint `PERIODIC COMMIT`; more information on this may be found in `PERIODIC COMMIT query hint`.

Configuration settings for file URLs

`dbms.security.allow_csv_import_from_file_urls`

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`. Setting `dbms.security.allow_csv_import_from_file_urls=false` will completely disable access to the file system for `LOAD CSV`.

`dbms.directories.import`

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). The default value is `import`. This is a security measure which prevents the database from accessing files outside the standard `import` directory. Setting `dbms.directories.import` to be empty removes this security measure and instead allows access to any file on the system. This is not recommended.

File URLs will be resolved relative to the `dbms.directories.import` directory. For example, a file URL will typically look like `file:///myfile.csv` or `file:///myproject/myfile.csv`.

- If `dbms.directories.import` is set to the default value `import`, using the above URLs in `LOAD CSV` would read from `<NEO4J_HOME>/import/myfile.csv` and `<NEO4J_HOME>/import/myproject/myfile.csv` respectively.
- If it is set to `/data/csv`, using the above URLs in `LOAD CSV` would read from `/data/csv/myfile.csv` and `/data/csv/myproject/myfile.csv` respectively.

See the examples below for further details.

There is also a worked example, see [Importing CSV files with Cypher](#).

7.20.2. 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 `\`.

7.20.3. 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 '{csv-dir}/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

```
+-----+
| No data returned. |
+-----+
Nodes created: 4
Properties set: 8
Labels added: 4
```

7.20.4. 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 '{csv-dir}/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

```
+-----+
| No data returned. |
+-----+
Nodes created: 4
Properties set: 8
Labels added: 4
```

7.20.5. 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 '{csv-dir}/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

```
+-----+
| No data returned. |
+-----+
Nodes created: 4
Properties set: 8
Labels added: 4
```

7.20.6. 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 [PERIODIC COMMIT query hint](#).

Query

```
USING PERIODIC COMMIT
LOAD CSV FROM '{csv-dir}/artists.csv' AS line
CREATE (:Artist { name: line[1], year:toInt(line[2])})
```

Result

```
+-----+
| No data returned. |
+-----+
Nodes created: 4
Properties set: 8
Labels added: 4
```

7.20.7. 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 '{csv-dir}/artists.csv' AS line
CREATE (:Artist { name: line[1], year:toInt(line[2])})
```

Result

```
+-----+
| No data returned. |
+-----+
Nodes created: 4
Properties set: 8
Labels added: 4
```

7.20.8. 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 '{csv-dir}/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, size(a.name) AS size
```

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 | size |
+-----+
| "The ""Symbol"" | 1992 | 12   |
+-----+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

Chapter 8. Functions

This section contains information on all functions in the Cypher query language.

- Predicate functions [[Summary](#) | [Detail](#)]
- Scalar functions [[Summary](#) | [Detail](#)]
- Aggregating functions [[Summary](#) | [Detail](#)]
- List functions [[Summary](#) | [Detail](#)]
- Mathematical functions - numeric [[Summary](#) | [Detail](#)]
- Mathematical functions - logarithmic [[Summary](#) | [Detail](#)]
- Mathematical functions - trigonometric [[Summary](#) | [Detail](#)]
- String functions [[Summary](#) | [Detail](#)]
- Spatial functions [[Summary](#) | [Detail](#)]
- [User-defined functions](#)

Note that related information exists in [Operators](#).

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

Predicate functions

These functions return either true or false for the arguments given.

Function	Description
<code>all()</code>	Tests whether a predicate holds for all elements in the list.
<code>any()</code>	Tests whether a predicate holds for at least one element in the list.
<code>exists()</code>	Returns true if a match for the pattern exists in the graph, or the property exists in the node, relationship or map.
<code>none()</code>	Returns true if the predicate holds for no element in the list.
<code>single()</code>	Returns true if the predicate holds for exactly one of the elements in the list.

Scalar functions

These functions return a single value.

Function	Description
<code>coalesce()</code>	Returns the first non- <code>null</code> value in the list of expressions passed to it.
<code>endNode()</code>	Returns the last node of a relationship.
<code>head()</code>	Returns the first element in a list.
<code>id()</code>	Returns the id of the relationship or node.
<code>last()</code>	Returns the last element in a list.
<code>length()</code>	Returns the length of a path.
<code>properties()</code>	If the argument is a node or a relationship, the returned map is a map of its properties.

Function	Description
<code>size()</code>	Returns the number of items in a list.
<code>Size of pattern expression</code>	Returns the number of sub-graphs matching the pattern expression.
<code>Size of string</code>	Returns the size of a string.
<code>startNode()</code>	Returns the first node of a relationship.
<code>timestamp()</code>	Returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.
<code>toBoolean()</code>	Converts the argument to a boolean and returns the result.
<code>toFloat()</code>	Converts the argument to a float and returns the result.
<code>toInteger()</code>	Converts the argument to an integer and returns the result.
<code>type()</code>	Returns a string representation of the relationship type.

Aggregating functions

These functions take multiple values as arguments, and calculate and return an aggregated value from them.

Function	Description
<code>avg()</code>	Returns the average of a numeric column.
<code>collect()</code>	Returns a list containing all collected values.
<code>count()</code>	Returns the number of rows.
<code>max()</code>	Returns the highest value in a numeric column.
<code>min()</code>	Returns the lowest value in a numeric column.
<code>percentileCont()</code>	Returns the percentile of a given value over a group using linear interpolation.
<code>percentileDisc()</code>	Returns the nearest value to a given percentile over a group using a rounding method.
<code>stDev()</code>	Returns the standard deviation for a given value over a group for a sample of a population.
<code>stDevP()</code>	Returns the standard deviation for a given value over a group for an entire population.
<code>sum()</code>	Returns the sum of a numeric column.

List functions

These functions return lists of other values. Further details and examples of lists may be found in [Lists](#).

Function	Description
<code>extract()</code>	Returns a single property, or the value of a function from a list of nodes or relationships.
<code>filter()</code>	Returns all the elements in a list complying with a predicate.
<code>keys()</code>	Returns a list of string representations for the property names of a node, relationship, or map.
<code>labels()</code>	Returns a list of string representations for the labels attached to a node.

Function	Description
<code>nodes()</code>	Returns all nodes in a path.
<code>range()</code>	Returns numerical values in a range.
<code>reduce()</code>	Runs an expression against individual elements of a list, storing the result of the expression in an accumulator.
<code>relationships()/rels()</code>	Returns all relationships in a path.
<code>tail()</code>	Returns all but the first element in a list.

Mathematical functions - numeric

These functions all operate on numerical expressions only, and will return an error if used on any other values.

Function	Description
<code>abs()</code>	Returns the absolute value of a number.
<code>ceil()</code>	Returns the smallest integer greater than or equal to the argument.
<code>floor()</code>	Returns the greatest integer less than or equal to the expression.
<code>rand()</code>	Returns a random number in the range from 0 (inclusive) to 1 (exclusive), $[0, 1]$.
<code>round()</code>	Returns the numerical expression, rounded to the nearest integer.
<code>sign()</code>	Returns the signum of a number — zero if the expression is zero, -1 for any negative number, and 1 for any positive number.

Mathematical functions - logarithmic

These functions all operate on numerical expressions only, and will return an error if used on any other values.

Function	Description
<code>e()</code>	Returns the base of the natural logarithm, e .
<code>exp()</code>	Returns e^n , where e is the base of the natural logarithm, and n is the value of the argument expression.
<code>log()</code>	Returns the natural logarithm of the expression.
<code>log10()</code>	Returns the common logarithm (base 10) of the expression.
<code>sqrt()</code>	Returns the square root of a number.

Mathematical functions - trigonometric

These functions all operate on numerical expressions only, and will return an error if used on any other values.

All trigonometric functions operate on radians, unless otherwise specified.

Function	Description
<code>acos()</code>	Returns the arccosine of the expression.
<code>asin()</code>	Returns the arcsine of the expression.

Function	Description
atan()	Returns the arctangent of the expression.
atan2()	Returns the arctangent2 of a set of coordinates.
cos()	Returns the cosine of the expression.
cot()	Returns the cotangent of the expression.
degrees()	Converts radians to degrees.
haversin()	Returns half the versine of the expression.
pi()	Returns the mathematical constant <i>pi</i> .
radians()	Converts degrees to radians.
sin()	Returns the sine of the expression.
tan()	Returns the tangent of the expression.

String functions

These functions are used to manipulate strings or to create a string representation of another value.

Function	Description
left()	Returns a string containing the left n characters of the original string.
lTrim()	Returns the original string with whitespace removed from the left side.
replace()	Returns a string with the search string replaced by the replace string, replacing all occurrences.
reverse()	Returns the original string reversed.
right()	Returns a string containing the right n characters of the original string.
rTrim()	Returns the original string with whitespace removed from the right side.
split()	Returns the sequence of strings which are delimited by split patterns.
substring()	Returns a substring of the original, with a 0-based index start and length.
toLower()	Returns the original string in lowercase.
toString()	Converts the argument to a string.
toUpper()	Returns the original string in uppercase.
trim()	Returns the original string with whitespace removed from both sides.

Spatial functions

These functions are used to specify points in a 2D coordinate system and to calculate the geodesic distance between two points.

Function	Description
distance()	Returns a floating point number representing the geodesic distance between any two given points.
point() - cartesian 2D	Returns a point object, given two coordinate values in the cartesian coordinate system.

Function	Description
point() - WGS 84	Returns a point object, given two coordinate values in the WGS 84 coordinate system.

8.1. Predicate functions

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.

Functions:

- all()
- any()
- exists()
- none()
- single()

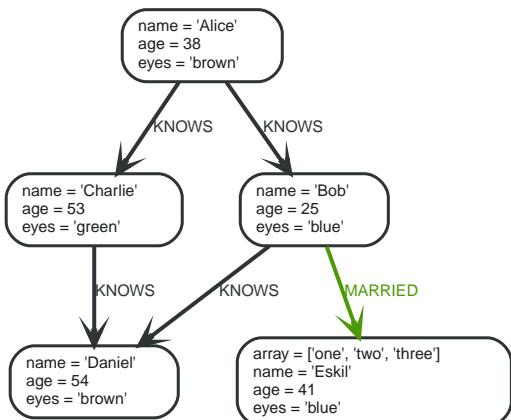


Figure 20. Graph

8.1.1. all()

Tests whether a predicate holds for all elements of this list.

Syntax: `all(variable IN list WHERE predicate)`

Arguments:

Name	Description
list	An expression that returns a list
variable	This is the variable that can be used from the predicate.
predicate	A predicate that is tested against all items in the list.

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'.

Table 110. Result

p
[Node[0]{name: "Alice", age:38, eyes: "brown"}, :KNOWS[1]{}, Node[2]{name: "Charlie", age:53, eyes: "green"}, :KNOWS[3]{}, Node[3]{name: "Daniel", age:54, eyes: "brown"}]
1 row

8.1.2. any()

Tests whether a predicate holds for at least one element in the list.

Syntax: `any(variable IN list WHERE predicate)`

Arguments:

Name	Description
<code>list</code>	An expression that returns a list
<code>variable</code>	This is the variable that can be used from the predicate.
<code>predicate</code>	A predicate that is tested against all items in the list.

Query

```
MATCH (a)
WHERE a.name = 'Eskil' AND ANY (x IN a.array WHERE x = 'one')
RETURN a.name, a.array
```

All nodes in the returned paths have at least one 'one' value set in the array property named `array`.

Table 111. Result

a.name	a.array
"Eskil"	["one", "two", "three"]
1 row	

8.1.3. 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:

Name	Description
<code>pattern-or-property</code>	A pattern or a property (in the form 'variable.prop').

Query

```
MATCH (n)
WHERE exists(n.name)
RETURN n.name AS name, exists((n)-[:MARRIED]->()) AS is_married
```

This query returns the names of all nodes with a name property along with a boolean `true / false` indicating if they are married.

Table 112. Result

name	is_married
"Alice"	false
"Bob"	true
"Charlie"	false
"Daniel"	false
"Eskil"	false
5 rows	

8.1.4. none()

Returns true if the predicate holds for no element in the list.

Syntax: `none(variable IN list WHERE predicate)`

Arguments:

Name	Description
<code>list</code>	An expression that returns a list
<code>variable</code>	This is the variable that can be used from the predicate.
<code>predicate</code>	A predicate that is tested against all items in the list.

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 an `age` property set to '25'.

Table 113. Result

p
[Node[0]{name:"Alice",age:38,eyes:"brown"},:KNOWS[1]{},Node[2]{name:"Charlie",age:53,eyes:"green"}]
[Node[0]{name:"Alice",age:38,eyes:"brown"},:KNOWS[1]{},Node[2]{name:"Charlie",age:53,eyes:"green"},:KNOWS[3]{},Node[3]{name:"Daniel",age:54,eyes:"brown"}]
2 rows

8.1.5. single()

Returns true if the predicate holds for exactly one of the elements in the list.

Syntax: `single(variable IN list WHERE predicate)`

Arguments:

Name	Description
<code>list</code>	An expression that returns a list
<code>variable</code>	This is the variable that can be used from the predicate.
<code>predicate</code>	A predicate that is tested against all items in the list.

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'.

Table 114. Result

p
[Node[0]{name:"Alice",age:38,eyes:"brown"},:KNOWS[],Node[1]{name:"Bob",age:25,eyes:"blue"}]
1 row

8.2. Scalar functions

Scalar functions return a single value.



The `length()` and `size()` functions are quite similar, and so it is important to take note of the difference. Owing to backwards compatibility, `length()` currently works on four types: strings, paths, lists and pattern expressions. However, it is recommended to use `length()` only for paths, and the `size()` function for strings, lists and pattern expressions. `length()` on those types may be deprecated in future.



The function `toInt()` has been superseded by `toInteger()`, and will be deprecated in a future release.

Functions:

- [coalesce\(\)](#)
- [endNode\(\)](#)
- [head\(\)](#)
- [id\(\)](#)
- [last\(\)](#)
- [length\(\)](#)
- [properties\(\)](#)
- [size\(\)](#)
- [Size of pattern expression](#)
- [Size of string](#)
- [startNode\(\)](#)
- [timestamp\(\)](#)
- [toBoolean\(\)](#)
- [toFloat\(\)](#)
- [toInteger\(\)](#)
- [type\(\)](#)

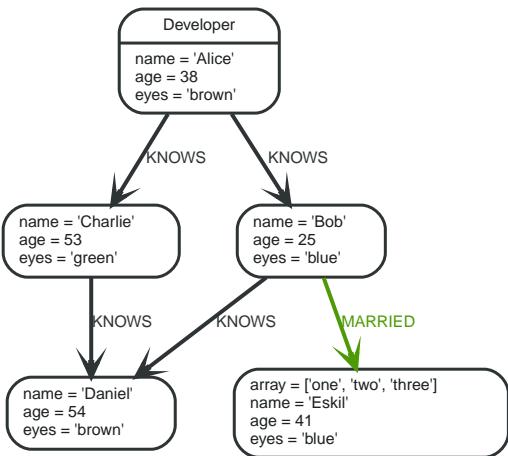


Figure 21. Graph

8.2.1. 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:

Name	Description
<code>expression</code>	The expression that may return <code>null</code> .

Query

```

MATCH (a)
WHERE a.name = 'Alice'
RETURN coalesce(a.hairColor, a.eyes)
  
```

Table 115. Result

<code>coalesce(a.hairColor, a.eyes)</code>
"brown"
1 row

8.2.2. endNode()

`endNode()` returns the end node of a relationship.

Syntax: `endNode(relationship)`

Arguments:

Name	Description
<code>relationship</code>	An expression that returns a relationship.

Query

```

MATCH (x:Developer)-[r]-()
RETURN endNode(r)
  
```

Table 116. Result

endNode(r)
Node[2]{name: "Charlie", age: 53, eyes: "green"}
Node[1]{name: "Bob", age: 25, eyes: "blue"}
2 rows

8.2.3. head()

`head()` returns the first element in a list.

Syntax: `head(expression)`

Arguments:

Name	Description
<code>expression</code>	This expression should return a list of some kind.

Query

```
MATCH (a)
WHERE a.name = 'Eskil'
RETURN a.array, head(a.array)
```

The first element in the list is returned.

Table 117. Result

a.array	<code>head(a.array)</code>
[“one”, “two”, “three”]	“one”
1 row	

8.2.4. id()

Returns the id of the relationship or node.

Syntax: `id(expression)`

Arguments:

Name	Description
<code>expression</code>	An expression that returns a node or a relationship.

Query

```
MATCH (a)
RETURN id(a)
```

This returns the node id for all the nodes.

Table 118. Result

<code>id(a)</code>
0

id(a)
1
2
3
4
5 rows

8.2.5. last()

`last()` returns the last element in a list.

Syntax: `last(expression)`

Arguments:

Name	Description
<code>expression</code>	This expression should return a list of some kind.

Query

```
MATCH (a)
WHERE a.name = 'Eskil'
RETURN a.array, last(a.array)
```

The last element in the list is returned.

Table 119. Result

a.array	last(a.array)
["one", "two", "three"]	"three"
1 row	

8.2.6. length()

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

Syntax: `length(path)`

Arguments:

Name	Description
<code>path</code>	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.

Table 120. Result

<code>length(p)</code>
2
2
2
3 rows

8.2.7. properties()

`properties()` converts the arguments to a map of its properties. If the argument is a node or a relationship, the returned map is a map of its properties. If the argument is already a map, it is returned unchanged.

Syntax: `properties(expression)`

Arguments:

Name	Description
<code>expression</code>	An expression that returns a node, a relationship, or a map.

Query

```
CREATE (p:Person { name: 'Stefan', city: 'Berlin' })
RETURN properties(p)
```

Table 121. Result

<code>properties(p)</code>
<code>{name -> "Stefan", city -> "Berlin"}</code>
1 row
Nodes created: 1
Properties set: 2
Labels added: 1

8.2.8. size()

To return or filter on the size of a list, use the `size()` function.

Syntax: `size(list)`

Arguments:

Name	Description
<code>list</code>	An expression that returns a list.

Query

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

Table 122. Result

<code>size(['Alice', 'Bob'])</code>
2

```
size(['Alice', 'Bob'])
```

1 row

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

8.2.9. Size of pattern expression

This is the same `size()` method described before, but instead of passing in a list directly, you provide a pattern expression that can be used in a match query to provide a new set of results. These results are a *list* of paths. The size of the result is calculated, not the length of the expression itself.

Syntax: `size(pattern expression)`

Arguments:

Name	Description
<code>pattern expression</code>	A pattern expression that returns a list.

Query

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

Table 123. Result

<code>fof</code>
3
1 row

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

8.2.10. Size of string

To return or filter on the size of a string, use the `size()` function.

Syntax: `size(string)`

Arguments:

Name	Description
<code>string</code>	An expression that returns a string.

Query

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

Table 124. Result

<code>size(a.name)</code>
7
1 row

The size of the name '**Charlie**' is returned by the query.

8.2.11. startNode()

`startNode()` returns the starting node of a relationship.

Syntax: `startNode(relationship)`

Arguments:

Name	Description
<code>relationship</code>	An expression that returns a relationship.

Query

```
MATCH (x:Developer)-[r]-(y)
RETURN startNode(r)
```

Table 125. Result

<code>startNode(r)</code>
<code>Node[0]{name:"Alice",age:38,eyes:"brown"}</code>
<code>Node[0]{name:"Alice",age:38,eyes:"brown"}</code>
2 rows

8.2.12. 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()`

Query

```
RETURN timestamp()
```

The time in milliseconds is returned.

Table 126. Result

<code>timestamp()</code>
1500055198779
1 row

8.2.13. toBoolean()

`toBoolean()` converts the argument to a boolean. It attempts to convert a string to a boolean value, and if passed a boolean, it will leave it unchanged. If the parsing fails, `null` will be returned.

Syntax: `toBoolean(expression)`

Arguments:

Name	Description
expression	An expression that returns a boolean or a string.

Query

```
RETURN toBoolean('TRUE'), toBoolean('not a boolean')
```

Table 127. Result

toBoolean('TRUE')	toBoolean('not a boolean')
true	<null>
1 row	

8.2.14. 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:

Name	Description
expression	An expression that returns anything.

Query

```
RETURN toFloat('11.5'), toFloat('not a number')
```

Table 128. Result

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

8.2.15. toInteger()

`toInteger()` 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: `toInteger(expression)`

Arguments:

Name	Description
expression	An expression that returns anything.

Query

```
RETURN toInteger('42'), toInteger('not a number')
```

Table 129. Result

<code>toInteger('42')</code>	<code>toInteger('not a number')</code>
<code>42</code>	<code><null></code>
1 row	

8.2.16. type()

Returns a string representation of the relationship type.

Syntax: `type(relationship)`

Arguments:

Name	Description
<code>relationship</code>	A relationship.

Query

```
MATCH (n)-[r]->()
WHERE n.name = 'Alice'
RETURN type(r)
```

The relationship type of `r` is returned by the query.

Table 130. Result

<code>type(r)</code>
<code>"KNOWS"</code>
<code>"KNOWS"</code>
2 rows

8.3. Aggregating functions

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 not an 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** operator. It is used to make all values unique before running them through an aggregate function. More information about **DISTINCT** may be found [here](#).

Functions:

- [avg\(\)](#)
- [collect\(\)](#)
- [count\(\)](#)
- [max\(\)](#)
- [min\(\)](#)
- [percentileCont\(\)](#)
- [percentileDisc\(\)](#)
- [stDev\(\)](#)
- [stDevP\(\)](#)
- [sum\(\)](#)

The following graph is used for the examples below:

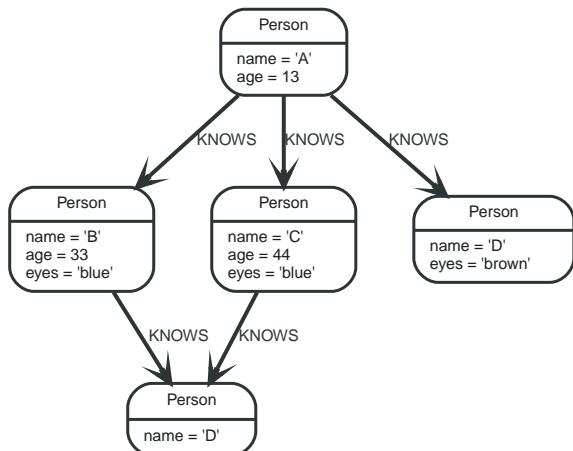


Figure 22. Graph

8.3.1. avg()

avg() calculates the average of a numeric column.

Syntax: `avg(expression)`

Arguments:

Name	Description
<code>expression</code>	A numeric expression.

Query

```
MATCH (n:Person)
RETURN avg(n.age)
```

The average of all the values in the property `age` is returned by the example query.

Table 131. Result

```
avg(n.age)
```

```
30.0
```

```
1 row
```

8.3.2. collect()

`collect()` collects all the values into a list. It will ignore `null` values.

Syntax: `collect(expression)`

Arguments:

Name	Description
<code>expression</code>	An expression.

Query

```
MATCH (n:Person)
RETURN collect(n.age)
```

Returns a single row, with all the values collected.

Table 132. Result

<code>collect(n.age)</code>
<code>[13, 33, 44]</code>
1 row

8.3.3. 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(<expression>)`, which counts the number of non-`null` values in `<expression>`.

Syntax: `count(expression)`

Arguments:

Name	Description
<code>expression</code>	An expression.

Count nodes

`count(*)` can be used to count the number of nodes; for example, the number of nodes connected to one node.

Query

```
MATCH (n { name: 'A' })-->(x)
RETURN labels(n), n.age, count(*)
```

This returns the start node and the count of related nodes.

Table 133. Result

labels(n)	n.age	count(*)
["Person"]	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.

Table 134. 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 variable you care about.

Query

```
MATCH (n { name: 'A' })-->(x)
RETURN count(x)
```

The number of nodes connected to the start node is returned.

Table 135. Result

count(x)
3
1 row

Count non-`null` values

The non-`null` values can be counted by using `count(<expression>)`.

Query

```
MATCH (n:Person)
RETURN count(n.age)
```

The count of related nodes with the `age` property set is returned.

Table 136. Result

count(n.age)
3

count(n.age)

| 1 row |

Counting with and without duplicates

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`.

Query

```
MATCH (me:Person)-->(friend:Person)-->(friend_of_friend:Person)
WHERE me.name = 'A'
RETURN count(DISTINCT friend_of_friend), count(friend_of_friend)
```

Table 137. Result

count(DISTINCT friend_of_friend)	count(friend_of_friend)
1	2
1 row	

8.3.4. max()

`max()` takes a numeric property as input, and returns the highest value in that column.

Syntax: `max(expression)`

Arguments:

Name	Description
expression	A numeric expression.

Query

```
MATCH (n:Person)
RETURN max(n.age)
```

The highest of all the values in the property `age` is returned.

Table 138. Result

max(n.age)
44
1 row

8.3.5. min()

`min()` takes a numeric property as input, and returns the lowest value in that column.

Syntax: `min(expression)`

Arguments:

Name	Description
expression	A numeric expression.

Query

```
MATCH (n:Person)
RETURN min(n.age)
```

The lowest of all the values in the property `age` is returned.

Table 139. Result

min(n.age)
13
1 row

8.3.6. 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`.

Syntax: `percentileCont(expression, percentile)`

Arguments:

Name	Description
expression	A numeric expression.
percentile	A numeric value between 0.0 and 1.0

Query

```
MATCH (n:Person)
RETURN percentileCont(n.age, 0.4)
```

The 40th percentile of the values in the property `age` is returned by the example query, calculated with a weighted average.

Table 140. Result

percentileCont(n.age, 0.4)
29.0
1 row

8.3.7. 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`.

Syntax: `percentileDisc(expression, percentile)`

Arguments:

Name	Description
expression	A numeric expression.
percentile	A numeric value between 0.0 and 1.0

Query

```
MATCH (n:Person)
RETURN percentileDisc(n.age, 0.5)
```

The 50th percentile of the values in the property `age` is returned by the example query. In this case, 0.5 is the median, or 50th percentile.

Table 141. Result

percentileDisc(n.age, 0.5)
33
1 row

8.3.8. 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, `stdDevP` should be used.

Syntax: `stDev(expression)`

Arguments:

Name	Description
expression	A numeric expression.

Query

```
MATCH (n)
WHERE n.name IN ['A', 'B', 'C']
RETURN stDev(n.age)
```

The standard deviation of the values in the property `age` is returned.

Table 142. Result

stDev(n.age)
15.716233645501712
1 row

8.3.9. 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.

Syntax: `stDevP(expression)`

Arguments:

Name	Description
expression	A numeric expression.

Query

```
MATCH (n)
WHERE n.name IN ['A', 'B', 'C']
RETURN stDevP(n.age)
```

The population standard deviation of the values in the property `age` is returned by the example query.

Table 143. Result

stDevP(n.age)
12.832251036613439
1 row

8.3.10. sum()

`sum()` simply adds up all the numeric values it encounters. Any `null` values are silently dropped.

Syntax: `sum(expression)`

Arguments:

Name	Description
expression	A numeric expression.

Query

```
MATCH (n:Person)
RETURN sum(n.age)
```

This returns the sum of all the values in the property `age`.

Table 144. Result

sum(n.age)
90
1 row

8.4. List functions

List functions return lists of things — nodes in a path, and so on.

Further details and examples of lists may be found in [Lists](#) and [List operators](#).

Functions:

- [extract\(\)](#)
- [filter\(\)](#)

- `keys()`
- `labels()`
- `nodes()`
- `range()`
- `reduce()`
- `relationships()`
- `tail()`

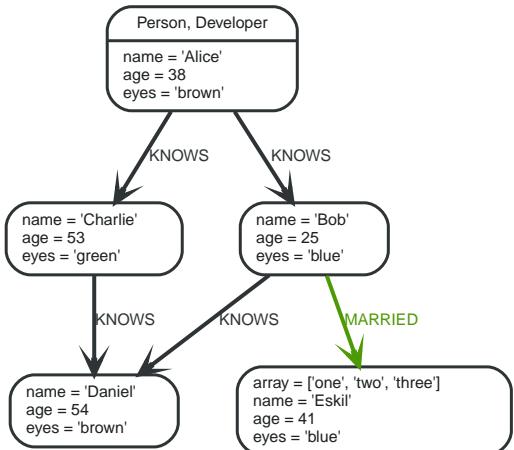


Figure 23. Graph

8.4.1. extract()

To return a single property, or the value of a function from a list of nodes or relationships, you can use `extract()`. It will go through a list, run an expression on every element, and return the results in a list with these values. It works like the `map` method in functional languages such as Lisp and Scala.

Syntax: `extract(variable IN list | expression)`

Arguments:

Name	Description
<code>list</code>	An expression that returns a list
<code>variable</code>	The closure will have a variable introduced in its context. Here you decide which variable to use.
<code>expression</code>	This expression will run once per value in the list, and produces the result list.

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.

Table 145. Result

extracted
[38,25,54]
1 row

8.4.2. filter()

`filter()` returns all the elements in a list that comply to a predicate.

Syntax: `filter(variable IN list WHERE predicate)`

Arguments:

Name	Description
<code>list</code>	An expression that returns a list
<code>variable</code>	This is the variable that can be used from the predicate.
<code>predicate</code>	A predicate that is tested against all items in the list.

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'.

Table 146. Result

<code>a.array</code>	<code>filter(x IN a.array WHERE size(x) = 3)</code>
<code>["one", "two", "three"]</code>	<code>["one", "two"]</code>
1 row	

8.4.3. keys()

Returns a list of string representations for the property names of a node, relationship, or map.

Syntax: `keys(expression)`

Arguments:

Name	Description
<code>expression</code>	An expression that returns a node, a relationship, or a map.

Query

```
MATCH (a)
WHERE a.name = 'Alice'
RETURN keys(a)
```

The name of the properties of `n` is returned by the query.

Table 147. Result

<code>keys(a)</code>
<code>["name", "age", "eyes"]</code>
1 row

8.4.4. labels()

Returns a list of string representations for the labels attached to a node.

Syntax: `labels(node)`

Arguments:

Name	Description
<code>node</code>	Any expression that returns a single node.

Query

```
MATCH (a)
WHERE a.name = 'Alice'
RETURN labels(a)
```

The labels of `n` are returned by the query.

Table 148. Result

<code>labels(a)</code>
<code>["Person", "Developer"]</code>
1 row

8.4.5. nodes()

Returns all nodes in a path.

Syntax: `nodes(path)`

Arguments:

Name	Description
<code>path</code>	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.

Table 149. Result

<code>nodes(p)</code>
<code>[Node[0]{name:"Alice",age:38,eyes:"brown"},Node[1]{name:"Bob",age:25,eyes:"blue"},Node[4]{array:["one","two","three"],name:"Eskil",age:41,eyes:"blue"}]</code>
1 row

8.4.6. range()

`range()` returns numerical values in a range. The default distance between values in the range is `1`. The range is inclusive in both ends.

Syntax: `range(start, end [, step])`

Arguments:

Name	Description
<code>start</code>	A numerical expression.
<code>end</code>	A numerical expression.
<code>step</code>	A numerical expression.

Query

```
RETURN range(0, 10), range(2, 18, 3)
```

Two lists of numbers in the given ranges are returned.

Table 150. Result

<code>range(0, 10)</code>	<code>range(2, 18, 3)</code>
<code>[0,1,2,3,4,5,6,7,8,9,10]</code>	<code>[2,5,8,11,14,17]</code>
1 row	

8.4.7. `reduce()`

To run an expression against individual elements of a list, and store the result of the expression in an accumulator, you can use `reduce()`. It will go through a list, 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, variable IN list | expression)`

Arguments:

Name	Description
<code>accumulator</code>	A variable that will hold the result and the partial results as the list is iterated.
<code>initial</code>	An expression that runs once to give a starting value to the accumulator.
<code>list</code>	An expression that returns a list.
<code>variable</code>	The closure will have a variable introduced in its context. Here you decide which variable to use.
<code>expression</code>	This expression will run once per value in the list, 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.

Table 151. Result

reduction
117
1 row

8.4.8. relationships()

Returns all relationships in a path.

Syntax: `relationships(path)`

Arguments:

Name	Description
<code>path</code>	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.

Table 152. Result

<code>relationships(p)</code>
<code>[:KNOWS[0]{},:MARRIED[4]{}]</code>
1 row

8.4.9. tail()

`tail()` returns all but the first element in a list.

Syntax: `tail(expression)`

Arguments:

Name	Description
<code>expression</code>	This expression should return a list 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.

Table 153. Result

<code>a.array</code>	<code>tail(a.array)</code>
<code>["one", "two", "three"]</code>	<code>["two", "three"]</code>
1 row	

8.5. Mathematical functions - numeric

These functions all operate on numerical expressions only, and will return an error if used on any other values. See also [Mathematical operators](#).

Functions:

- [abs\(\)](#)
- [ceil\(\)](#)
- [floor\(\)](#)
- [rand\(\)](#)
- [round\(\)](#)
- [sign\(\)](#)

The following graph is used for the examples below:

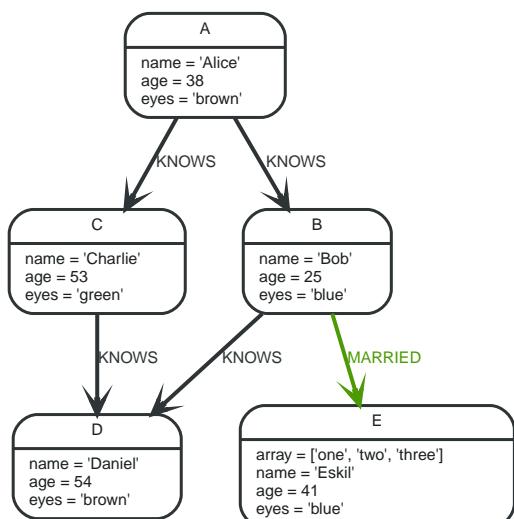


Figure 24. Graph

8.5.1. abs()

`abs()` returns the absolute value of a number.

Syntax: `abs(expression)`

Arguments:

Name	Description
<code>expression</code>	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.

Table 154. Result

a.age	e.age	abs(a.age - e.age)
38	41	3
1 row		

8.5.2. ceil()

`ceil()` returns the smallest integer greater than or equal to the argument.

Syntax: `ceil(expression)`

Arguments:

Name	Description
expression	A numeric expression.

Query

```
RETURN ceil(0.1)
```

The ceil of `0.1`.

Table 155. Result

ceil(0.1)
1.0
1 row

8.5.3. floor()

`floor()` returns the greatest integer less than or equal to the expression.

Syntax: `floor(expression)`

Arguments:

Name	Description
expression	A numeric expression.

Query

```
RETURN floor(0.9)
```

The floor of `0.9` is returned.

Table 156. Result

floor(0.9)
0.0
1 row

8.5.4. rand()

`rand()` returns a random number in the range from 0 (inclusive) to 1 (exclusive), [0,1). The numbers returned follow an approximate uniform distribution.

Syntax: `rand()`

Query

```
RETURN rand()
```

A random number is returned.

Table 157. Result

<code>rand()</code>
0.008924905099155334
1 row

8.5.5. round()

`round()` returns the numerical expression, rounded to the nearest integer.

Syntax: `round(expression)`

Arguments:

Name	Description
<code>expression</code>	A numeric expression that represents the angle in radians.

Query

```
RETURN round(3.141592)
```

3.0 is returned.

Table 158. Result

<code>round(3.141592)</code>
3.0
1 row

8.5.6. 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:

Name	Description
<code>expression</code>	A numeric expression.

Query

```
RETURN sign(-17), sign(0.1)
```

The signs of `-17` and `0.1` are returned.

Table 159. Result

sign(-17)	sign(0.1)
-1	1
1 row	

8.6. Mathematical functions - logarithmic

These functions all operate on numerical expressions only, and will return an error if used on any other values. See also [Mathematical operators](#).

Functions:

- [e\(\)](#)
- [exp\(\)](#)
- [log\(\)](#)
- [log10\(\)](#)
- [sqrt\(\)](#)

8.6.1. e()

`e()` returns the base of the natural logarithm, `e`.

Syntax: `e()`

Query

```
RETURN e()
```

The base of the natural logarithm, `e`, is returned.

Table 160. Result

e()
2.718281828459045
1 row

8.6.2. exp()

`exp()` returns e^n , where `e` is the base of the natural logarithm, and `n` is the value of the argument expression.

Syntax: `e(expression)`

Arguments:

Name	Description
expression	A numeric expression.

Query

```
RETURN exp(2)
```

e to the power of 2 is returned.

Table 161. Result

exp(2)
7.38905609893065
1 row

8.6.3. log()

`log()` returns the natural logarithm of the expression.

Syntax: `log(expression)`

Arguments:

Name	Description
expression	A numeric expression.

Query

```
RETURN log(27)
```

The natural logarithm of 27 is returned.

Table 162. Result

log(27)
3.295836866004329
1 row

8.6.4. log10()

`log10()` returns the common logarithm (base 10) of the expression.

Syntax: `log10(expression)`

Arguments:

Name	Description
expression	A numeric expression.

Query

```
RETURN log10(27)
```

The common logarithm of 27 is returned.

Table 163. Result

log10(27)
1.4313637641589874
1 row

8.6.5. sqrt()

`sqrt()` returns the square root of a number.

Syntax: `sqrt(expression)`

Arguments:

Name	Description
<code>expression</code>	A numeric expression.

Query

```
RETURN sqrt(256)
```

The square root of 256 is returned.

Table 164. Result

sqrt(256)
16.0
1 row

8.7. Mathematical functions - trigonometric

These functions all operate on numerical expressions only, and will return an error if used on any other values. See also [Mathematical operators](#).

Functions:

- [acos\(\)](#)
- [asin\(\)](#)
- [atan\(\)](#)
- [atan2\(\)](#)
- [cos\(\)](#)
- [cot\(\)](#)
- [degrees\(\)](#)
- [haversin\(\)](#)
- Spherical distance using the `haversin()` function
- [pi\(\)](#)
- [radians\(\)](#)

- [sin\(\)](#)
- [tan\(\)](#)

All trigonometric functions operate on radians, unless otherwise specified.

8.7.1. acos()

`acos()` returns the arccosine of the expression, in radians.

Syntax: `acos(expression)`

Arguments:

Name	Description
<code>expression</code>	A numeric expression that represents the angle in radians.

Query

```
RETURN acos(0.5)
```

The arccosine of `0.5`.

Table 165. Result

<code>acos(0.5)</code>
<code>1.0471975511965979</code>
1 row

8.7.2. asin()

`asin()` returns the arcsine of the expression, in radians.

Syntax: `asin(expression)`

Arguments:

Name	Description
<code>expression</code>	A numeric expression that represents the angle in radians.

Query

```
RETURN asin(0.5)
```

The arcsine of `0.5`.

Table 166. Result

<code>asin(0.5)</code>
<code>0.5235987755982989</code>
1 row

8.7.3. atan()

`atan()` returns the arctangent of the expression, in radians.

Syntax: `atan(expression)`

Arguments:

Name	Description
<code>expression</code>	A numeric expression that represents the angle in radians.

Query

```
RETURN atan(0.5)
```

The arctangent of `0.5`.

Table 167. Result

<code>atan(0.5)</code>
<code>0.4636476090008061</code>
1 row

8.7.4. atan2()

`atan2()` returns the arctangent2 of a set of coordinates, in radians.

Syntax: `atan2(expression1, expression2)`

Arguments:

Name	Description
<code>expression1</code>	A numeric expression for y that represents the angle in radians.
<code>expression2</code>	A numeric expression for x that represents the angle in radians.

Query

```
RETURN atan2(0.5, 0.6)
```

The arctangent2 of `0.5` and `0.6`.

Table 168. Result

<code>atan2(0.5, 0.6)</code>
<code>0.6947382761967033</code>
1 row

8.7.5. cos()

`cos()` returns the cosine of the expression.

Syntax: `cos(expression)`

Arguments:

Name	Description
expression	A numeric expression that represents the angle in radians.

Query

```
RETURN cos(0.5)
```

The cosine of `0.5`.

Table 169. Result

cos(0.5)
0.8775825618903728
1 row

8.7.6. cot()

`cot()` returns the cotangent of the expression.

Syntax: `cot(expression)`

Arguments:

Name	Description
expression	A numeric expression that represents the angle in radians.

Query

```
RETURN cot(0.5)
```

The cotangent of `0.5`.

Table 170. Result

cot(0.5)
1.830487721712452
1 row

8.7.7. degrees()

`degrees()` converts radians to degrees.

Syntax: `degrees(expression)`

Arguments:

Name	Description
expression	A numeric expression that represents the angle in radians.

Query

```
RETURN degrees(3.14159)
```

The number of degrees in something close to π .

Table 171. Result

degrees(3.14159)
179.99984796050427
1 row

8.7.8. haversin()

`haversin()` returns half the versine of the expression.

Syntax: `haversin(expression)`

Arguments:

Name	Description
expression	A numeric expression that represents the angle in radians.

Query

```
RETURN haversin(0.5)
```

The haversine of `0.5` is returned.

Table 172. Result

haversin(0.5)
0.06120871905481362
1 row

8.7.9. 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 estimated distance between 'Berlin' and 'San Mateo' is returned.

Table 173. Result

dist
9129.969740051658

dist

1 row
Nodes created: 2
Properties set: 4
Labels added: 2

8.7.10. pi()

`pi()` returns the mathematical constant *pi*.

Syntax: `pi()`

Query

```
RETURN pi()
```

The constant *pi* is returned.

Table 174. Result

pi()
3.141592653589793
1 row

8.7.11. radians()

`radians()` converts degrees to radians.

Syntax: `radians(expression)`

Arguments:

Name	Description
<code>expression</code>	A numeric expression that represents the angle in degrees.

Query

```
RETURN radians(180)
```

The number of radians in `180` degrees is returned (*pi*).

Table 175. Result

radians(180)
3.141592653589793
1 row

8.7.12. sin()

`sin()` returns the sine of the expression.

Syntax: `sin(expression)`

Arguments:

Name	Description
expression	A numeric expression that represents the angle in radians.

Query

```
RETURN sin(0.5)
```

The sine of `0.5` is returned.

Table 176. Result

sin(0.5)
0.479425538604203
1 row

8.7.13. tan()

`tan()` returns the tangent of the expression.

Syntax: `tan(expression)`

Arguments:

Name	Description
expression	A numeric expression that represents the angle in radians.

Query

```
RETURN tan(0.5)
```

The tangent of `0.5` is returned.

Table 177. Result

tan(0.5)
0.5463024898437905
1 row

8.8. 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 and booleans.



The functions `lower()` and `upper()` have been superseded by `toLower()` and `toUpper()`, respectively, and will be deprecated in a future release.

See also [String operators](#).

Functions:

- [left\(\)](#)
- [lTrim\(\)](#)
- [replace\(\)](#)
- [reverse\(\)](#)
- [right\(\)](#)
- [rTrim\(\)](#)
- [split\(\)](#)
- [substring\(\)](#)
- [toLower\(\)](#)
- [toString\(\)](#)
- [toUpper\(\)](#)
- [trim\(\)](#)

8.8.1. left()

`left()` returns a string containing the left n characters of the original string.

Syntax: `left(original, length)`

Arguments:

Name	Description
<code>original</code>	An expression that returns a string
<code>n</code>	An expression that returns a positive number.

Query

```
RETURN left('hello', 3)
```

Table 178. Result

<code>left('hello', 3)</code>
"hel"
1 row

8.8.2. ltrim()

`lTrim()` returns the original string with whitespace removed from the left side.

Syntax: `lTrim(original)`

Arguments:

Name	Description
<code>original</code>	An expression that returns a string

Query

```
RETURN lTrim('    hello')
```

Table 179. Result

lTrim(' hello')
"hello"
1 row

8.8.3. replace()

`replace()` returns a string with the search string replaced by the replace string. It replaces all occurrences.

Syntax: `replace(original, search, replace)`

Arguments:

Name	Description
<code>original</code>	An expression that returns a string
<code>search</code>	An expression that returns a string to search for
<code>replace</code>	An expression that returns the string to replace the search string with

Query

```
RETURN replace("hello", "l", "w")
```

Table 180. Result

replace("hello", "l", "w")
"hewwo"
1 row

8.8.4. reverse()

`reverse()` returns the original string reversed.

Syntax: `reverse(original)`

Arguments:

Name	Description
<code>original</code>	An expression that returns a string

Query

```
RETURN reverse('anagram')
```

Table 181. Result

reverse('anagram')
"margana"
1 row

8.8.5. right()

`right()` returns a string containing the right n characters of the original string.

Syntax: `right(original, length)`

Arguments:

Name	Description
<code>original</code>	An expression that returns a string
<code>n</code>	An expression that returns a positive number.

Query

```
RETURN right('hello', 3)
```

Table 182. Result

<code>right('hello', 3)</code>
"llo"
1 row

8.8.6. rtrim()

`rTrim()` returns the original string with whitespace removed from the right side.

Syntax: `rTrim(original)`

Arguments:

Name	Description
<code>original</code>	An expression that returns a string

Query

```
RETURN rTrim('hello  ')
```

Table 183. Result

<code>rTrim('hello ')</code>
"hello"
1 row

8.8.7. split()

`split()` returns the sequence of strings which are delimited by split patterns.

Syntax: `split(original, splitPattern)`

Arguments:

Name	Description
<code>original</code>	An expression that returns a string

Name	Description
splitPattern	The string to split the original string with

Query

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

Table 184. Result

split('one,two', ',')
["one", "two"]
1 row

8.8.8. 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:

Name	Description
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)
```

Table 185. Result

substring('hello', 1, 3)	substring('hello', 2)
"ell"	"llo"
1 row	

8.8.9. toLower()

`toLower()` returns the original string in lowercase.

Syntax: `toLower(original)`

Arguments:

Name	Description
original	An expression that returns a string

Query

```
RETURN toLower('HELLO')
```

Table 186. Result

toLowerCase('HELLO')
"hello"
1 row

8.8.10. `toString()`

`toString()` converts the argument to a string. It converts integral and floating point numbers and booleans to strings, and if called with a string will leave it unchanged.

Syntax: `toString(expression)`

Arguments:

Name	Description
<code>expression</code>	An expression that returns a number, a boolean, or a string.

Query

```
RETURN toString(11.5), toString('already a string'), toString(TRUE )
```

Table 187. Result

<code>toString(11.5)</code>	<code>toString('already a string')</code>	<code>toString(true)</code>
"11.5"	"already a string"	"true"
1 row		

8.8.11. `toUpperCase()`

`toUpperCase()` returns the original string in uppercase.

Syntax: `toUpperCase(original)`

Arguments:

Name	Description
<code>original</code>	An expression that returns a string

Query

```
RETURN toUpperCase('hello')
```

Table 188. Result

<code>toUpperCase('hello')</code>
"HELLO"
1 row

8.8.12. `trim()`

`trim()` returns the original string with whitespace removed from both sides.

Syntax: `trim(original)`

Arguments:

Name	Description
<code>original</code>	An expression that returns a string

Query

```
RETURN trim(' hello ')
```

Table 189. Result

<code>trim(' hello ')</code>
<code>"hello"</code>
1 row

8.9. Spatial functions

These functions are used to specify points in a 2D coordinate system and to calculate the geodesic distance between two points.

Two coordinate reference systems (CRS) are supported: (i) [WGS 84](http://spatialreference.org/ref/epsg/4326/) (<http://spatialreference.org/ref/epsg/4326/>) and (ii) [cartesian 2D](http://spatialreference.org/ref/sr-org/7203/) (<http://spatialreference.org/ref/sr-org/7203/>).

WGS 84 is specified with a map containing coordinate values for either `longitude` and `latitude` (this is the default), or `x` and `y`. *Cartesian* is specified with a map containing only `x` and `y` coordinate values.

Functions:

- [distance\(\)](#)
- [point\(\) - WGS 84](#)
- [point\(\) - cartesian 2D](#)

The following graph is used for some of the examples below.

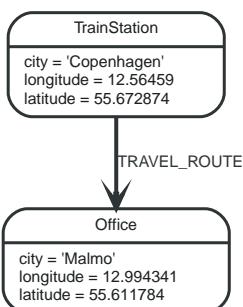


Figure 25. Graph

8.9.1. `distance()`

`distance()` returns the geodesic distance between two points in the same CRS. If the points are in the *cartesian* CRS, then the units of the returned distance will be the same as the units of the points, calculated using Pythagoras' theorem. If the points are in the *WGS-84* CRS, then the units of the returned distance will be meters, based on the haversine formula over a spherical earth

approximation.

Syntax: `distance(point1, point2)`

Arguments:

Name	Description
<code>point1</code>	A point in either the WGS 84 or cartesian CRS.
<code>point2</code>	A point in the same CRS as 'point1'.

Query

```
WITH point({ x: 2.3, y: 4.5, crs: 'cartesian' }) AS p1, point({ x: 1.1, y: 5.4, crs: 'cartesian' }) AS p2
RETURN distance(p1,p2) AS dist
```

The distance between two points in the *cartesian* CRS is returned.

Table 190. Result

<code>dist</code>
<code>1.5</code>
1 row

Query

```
MATCH (t:TrainStation)-[:TRAVEL_ROUTE]->(o:Office)
WITH point({ longitude: t.longitude, latitude: t.latitude }) AS trainPoint, point({ longitude: o.longitude, latitude: o.latitude }) AS officePoint
RETURN round(distance(trainPoint, officePoint)) AS travelDistance
```

The distance between the train station in Copenhagen and the Neo Technology office in Malmo is returned.

Table 191. Result

<code>travelDistance</code>
<code>27842.0</code>
1 row

Query

```
RETURN distance(NULL , point({ longitude: 56.7, latitude: 12.78 })) AS d
```

If `null` is provided as one or both of the arguments, `null` is returned.

Table 192. Result

<code>d</code>
<code><null></code>
1 row

8.9.2. `point()` - WGS 84

`point()` returns a value of type 'Point', which represents a point in the *WGS 84* coordinate system.

Syntax: `point({longitude | x, latitude | y [, crs]})`

Arguments:

Name	Description
A single map consisting of the following	
<code>longitude/x</code>	A numeric expression
<code>latitude/y</code>	A numeric expression
<code>crs</code>	The string 'WGS-84'

Query

```
RETURN point({ longitude: 56.7, latitude: 12.78 }) AS point
```

A point with a `longitude` of `56.7` and a `latitude` of `12.78` in the `WGS 84` CRS is returned.

Table 193. Result

<code>point</code>
<code>GeographicPoint(56.7,12.78,CRS(WGS-84,4326,http://spatialreference.org/ref/epsg/4326/))</code>
1 row

Query

```
RETURN point({ x: 2.3, y: 4.5, crs: 'WGS-84' }) AS point
```

`x` and `y` coordinates may be used in the `WGS 84` CRS instead of `longitude` and `latitude`, respectively, providing `crs` is set to '`WGS-84`'.

Table 194. Result

<code>point</code>
<code>GeographicPoint(2.3,4.5,CRS(WGS-84,4326,http://spatialreference.org/ref/epsg/4326/))</code>
1 row

Query

```
MATCH (p:Office)
RETURN point({ longitude: p.longitude, latitude: p.latitude }) AS officePoint
```

A point representing the coordinates of the city of Malmo in the `WGS 84` CRS is returned.

Table 195. Result

<code>officePoint</code>
<code>GeographicPoint(12.994341,55.611784,CRS(WGS-84,4326,http://spatialreference.org/ref/epsg/4326/))</code>
1 row

Query

```
RETURN point(NULL) AS p
```

If `null` is provided as the argument, `null` is returned.

Table 196. Result

p
<null>
1 row

8.9.3. point() - cartesian 2D

`point()` returns a value of type 'Point', which represents a point in the *cartesian* coordinate system.

Syntax: `point({x, y [, crs]})`

Arguments:

Name	Description
A single map consisting of the following	
x	A numeric expression
y	A numeric expression
crs	The string 'cartesian'

Query

```
RETURN point({ x: 2.3, y: 4.5 }) AS point
```

A point with an `x` coordinate of `2.3` and a `y` coordinate of `4.5` in the *cartesian* CRS is returned

Table 197. Result

point
<code>CartesianPoint(2.3,4.5,CRS(cartesian,7203,http://spatialreference.org/ref/sr-org/7203/))</code>
1 row

8.10. User-defined functions

User-defined functions are written in Java, deployed into the database and are called in the same way as any other Cypher function.

This example shows how you invoke a user-defined function called `join` from Cypher.

8.10.1. Call a user-defined function

This calls the user-defined function `org.neo4j.procedure.example.join()`.

Query

```
MATCH (n:Member)
RETURN org.neo4j.function.example.join(collect(n.name)) AS members
```

Result

```
+-----+  
| members |  
+-----+  
| "John,Paul,George,Ringo" |  
+-----+  
1 row
```

For developing and deploying user-defined functions in Neo4j, see [Extending Neo4j □ User-defined functions](#).

8.10.2. User-defined aggregation functions

User-defined aggregation functions are written in Java, deployed into the database and are called in the same way as any other Cypher function.

This example shows how you invoke a user-defined aggregation function called `longestString` from Cypher.

8.10.3. Call a user-defined aggregation function

This calls the user-defined function `org.neo4j.procedure.example.longestString()`.

Query

```
MATCH (n:Member)  
RETURN org.neo4j.function.example.longestString(n.name) AS member
```

Result

```
+-----+  
| member |  
+-----+  
| "George" |  
+-----+  
1 row
```

For developing and deploying user-defined aggregation functions in Neo4j, see [Extending Neo4j □ User-defined aggregation functions](#).

Chapter 9. Schema

This section explains how to work with an optional schema in Neo4j in the Cypher query language.

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.

- [Indexes](#)
 - Create a single-property index
 - Create a composite index
 - Drop a single-property index
 - Drop a composite index
 - Use index
 - Use a single-property index with WHERE using equality
 - Use a composite index with WHERE using equality
 - Use index with WHERE using range comparisons
 - Use index with IN
 - Use index with STARTS WITH
 - Use index when checking for the existence of a property
- [Constraints](#)
 - Unique node property constraints
 - Node property existence constraints
 - Relationship property existence constraints
 - Node Keys
- [Statistics](#)
 - Configuration options
 - Manual index sampling

9.1. Indexes

This section explains how to work with indexes in Neo4j and Cypher.

- [Introduction](#)
- [Create a single-property index](#)
- [Create a composite index](#)
- [Drop a single-property index](#)
- [Drop a composite index](#)
- [Use index](#)

- Use a single-property index with WHERE using equality
- Use a composite index with WHERE using equality
- Use index with WHERE using range comparisons
- Use index with IN
- Use index with STARTS WITH
- Use index when checking for the existence of a property

9.1.1. Introduction

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 enables the creation of indexes on one or more properties for all nodes that have a given label:

- An index that is created on a single property for any given label is called a *single-property index*.
- An index that is created on more than one property for any given label is called a *composite index*. Differences in the usage patterns between composite and single-property indexes are detailed in the examples below.

Once an index has been created, it will automatically be managed and kept up to date by the database when the graph is changed. Neo4j will automatically pick up and start using the index once it has been created and brought online.

9.1.2. Create a single-property index

An index on a single property for all nodes that have a particular label can be created with `CREATE INDEX ON :Label(property)`. Note that the index is not immediately available, but will be created in the background.

Query

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

Result

```
+-----+
| No data returned, and nothing was changed. |
+-----+
```

9.1.3. Create a composite index

An index on multiple properties for all nodes that have a particular label — i.e. a composite index — can be created with `CREATE INDEX ON :Label(prop1, ..., propN)`. Only nodes labeled with the specified label and which contain all the properties in the index definition will be added to the index.

The following statement will create a composite index on all nodes labeled with `Person` and which have both a `firstname` and `surname` property:

```
CREATE INDEX ON :Person(firstname, surname)
```

Now, assume the following query is run:

```
CREATE (a:Person {firstname: 'Bill', surname: 'Johnson', age: 34}),  
(b:Person {firstname: 'Sue', age: 39})
```

Node **a** has both a `firstname` and a `surname` property, and so it will be added to the composite index. However, as node **b** has no `surname` property, it will not be added to the composite index.

Note that the composite index is not immediately available, but will be created in the background.

9.1.4. Drop a single-property index

An index on all nodes that have a label and single property combination can be dropped with `DROP INDEX ON :Label(property)`.

Query

```
DROP INDEX ON :Person(firstname)
```

Result

```
+-----+  
| No data returned. |  
+-----+  
Indexes removed: 1
```

9.1.5. Drop a composite index

A composite index on all nodes that have a label and multiple property combination can be dropped with `DROP INDEX ON :Label(prop1, ..., propN)`.

The following statement will drop a composite index on all nodes labeled with `Person` and which have both a `firstname` and `surname` property:

```
DROP INDEX ON :Person(firstname, surname)
```

9.1.6. 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(firstname)` index, if it exists. If you want Cypher to use specific indexes, you can enforce it using hints. See [Planner hints and the USING keyword](#).

Query

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

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
Other	1	1	0	0	0	person
+ProduceResults	1	1	0	0	0	person
+NodeIndexSeek	1	1	2	1	0	:Person(firstname)
Total database accesses: 2						

9.1.7. Use a single-property index with WHERE using equality

A query containing equality comparisons of a single indexed property in the `WHERE` clause is backed automatically by the index. If you want Cypher to use specific indexes, you can enforce it using hints. See [Planner hints and the USING keyword](#).

Query

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

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
Other	1	1	0	0	0	person
+ProduceResults	1	1	0	0	0	person
+NodeIndexSeek	1	1	2	1	0	:Person(firstname)
Total database accesses: 2						

9.1.8. Use a composite index with WHERE using equality

A query containing equality comparisons for all the properties of a composite index will automatically be backed by the same index.

For instance, assume the following composite index has been created:

```
CREATE INDEX ON :Person(firstname, surname)
```

The following query will use the composite index:

```
MATCH (n:Person)
WHERE n.firstname = 'Bill' AND n.surname = 'Johnson'
RETURN n
```

However, this query will not be backed by the composite index, as the query does not contain an equality predicate on the `surname` property:

```
MATCH (n:Person)
WHERE n.firstname = 'Bill'
RETURN n
```

The query above will only be backed by an index on the `Person` label and `firstname` property defined thus: `:Person(firstname)`; i.e. a single-property index.

Moreover, unlike single-property indexes, composite indexes currently do not support queries containing the following types of predicates on properties in the index:

- Existence: `exists(n.prop)`
- Range: `n.prop > value`
- `STARTS WITH`
- `ENDS WITH`
- `CONTAINS`

Therefore, the following queries will not be able to be backed by the composite index defined earlier:

```
MATCH (n:Person)
WHERE n.firstname = 'Bill' AND exists(n.surname)
RETURN n
```

```
MATCH (n:Person)
WHERE n.firstname = 'Bill' AND n.surname STARTS WITH 'Jo'
RETURN n
```

If you want Cypher to use specific indexes, you can enforce it using hints. See [Planner hints and the USING keyword](#).

9.1.9. Use index with `WHERE` using range comparisons

Single-property indexes are also automatically used for inequality (range) comparisons of an indexed property in the `WHERE` clause. Composite indexes are currently not able to support range comparisons. If you want Cypher to use specific indexes, you can enforce it using hints. See [Planner hints and the USING keyword](#).

Query

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

Query Plan

Operator Variables Other	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	
+ProduceResults	10	1	0	0	0	person
+NodeIndexSeekByRange	10	1	2	1	0	person
:Person(firstname) > { AUTOSTRING0}						
Total database accesses:	2					

9.1.10. Use index with IN

The `IN` predicate on `person.firstname` in the following query will use the `Person(firstname)` index, if it exists. If you want Cypher to use specific indexes, you can enforce it using hints. See [Planner hints and the USING keyword](#).

Query

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

Query Plan

Operator Other	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
+ProduceResults	24	2	0	0	0	person
+NodeIndexSeek	24	2	4	2	0	person
:Person(firstname)						
Total database accesses:	4					

9.1.11. Use index with STARTS WITH

The `STARTS WITH` predicate on `person.firstname` in the following query will use the `Person(firstname)` index, if it exists. Composite indexes are currently not able to support `STARTS WITH`, `ENDS WITH` and `CONTAINS`.

Query

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

Query Plan

Operator Variables Other	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	
+ProduceResults	26	1	0	0	0	person
+NodeIndexSeekByRange :Person(firstname STARTS WITH { AUTOSTRING0})	26	1	2	1	0	person
Total database accesses: 2						

9.1.12. Use index when checking for the existence of a property

The `exists(p.firstname)` predicate in the following query will use the `Person(firstname)` index, if it exists. Composite indexes are currently not able to support the `exists` predicate.

Query

```
MATCH (p:Person)
WHERE exists(p.firstname)
RETURN p
```

Query Plan

Operator Other	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
+ProduceResults	2	2	0	1	0	p
+NodeIndexScan :Person(firstname)	2	2	3	2	0	p
Total database accesses: 3						

9.2. 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, along with node and relationship property existence constraints, and Node Keys, which guarantee both existence and uniqueness.

- [Introduction](#)
- [Unique node property constraints](#)
 - [Create uniqueness constraint](#)
 - [Drop uniqueness constraint](#)

- Create a node that complies with unique property constraints
- Create a node that violates a unique property constraint
- Failure to create a unique property constraint due to conflicting nodes
- Node property existence constraints
 - Create node property existence constraint
 - Drop node property existence constraint
 - Create a node that complies with property existence constraints
 - Create a node that violates a property existence constraint
 - Removing an existence constrained node property
 - Failure to create a node property existence constraint due to existing node
- Relationship property existence constraints
 - Create relationship property existence constraint
 - Drop relationship property existence constraint
 - Create a relationship that complies with property existence constraints
 - Create a relationship that violates a property existence constraint
 - Removing an existence constrained relationship property
 - Failure to create a relationship property existence constraint due to existing relationship
- Node Keys
 - Create a Node Key
 - Drop a Node Key
 - Create a node that complies with a Node Key
 - Create a node that violates a Node Key
 - Failure to create a Node Key due to existing node

9.2.1. Introduction

Unique property constraints 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.

Property existence constraints 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.

Node Keys ensure that, for a given label and set of properties:

- i. All the properties exist on all the nodes with that label.
- ii. The combination of the property values is unique.

Queries attempting to do any of the following will fail:

- Create new nodes without all the properties or where the combination of property values is not unique.
- Remove one of the mandatory properties.
- Update the properties so that the combination of property values is no longer unique.



Property existence constraints and Node Keys are only available in Neo4j Enterprise Edition. Note that databases with property existence constraints and/or Node Keys cannot be opened using Neo4j Community Edition.

A given label can have multiple constraints, and unique and property existence constraints can be combined on the same property.

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'.

Creating a constraint has the following implications on indexes:

- Adding a unique property constraint on a property will also add a [single-property index](#) on that property, so such an index cannot be added separately.
- Adding a Node Key for a set of properties will also add a [composite index](#) on those properties, so such an index cannot be added separately.
- Cypher will use these indexes for lookups just like other indexes; see the [Indexes](#) section for more details on the rules governing their behavior.
- If a unique property constraint is dropped and the single-property index on the property is still required, the index will need to be created explicitly.
- If a Node Key is dropped and the composite-property index on the properties is still required, the index will need to be created explicitly.

9.2.2. 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

```
+-----+  
| No data returned. |  
+-----+  
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

```
+-----+  
| No data returned. |  
+-----+  
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

```
+-----+
| No data returned. |
+-----+
Nodes created: 1
Properties set: 2
Labels added: 1
```

Create a node that violates 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 [Indexes](#) 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:
Both Node(0) and Node(1) have the label `Book` and property `isbn` = '1449356265'
```

9.2.3. 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(variable.propertyName)` syntax.

Query

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

Result

```
+-----+  
| No data returned. |  
+-----+  
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

```
+-----+  
| No data returned. |  
+-----+  
Property existence constraints removed: 1
```

Create a node that complies with property existence constraints

Create a `Book` node with an `isbn` property.

Query

```
CREATE (book:Book { isbn: '1449356265', title: 'Graph Databases' })
```

Result

```
+-----+  
| No data returned. |  
+-----+  
Nodes created: 1  
Properties set: 2  
Labels added: 1
```

Create a node that violates 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`
```

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`
```

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` must have the property `isbn`
```

9.2.4. 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(variable.propertyName)` syntax.

Query

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

Result

```
+-----+
| No data returned. |
+-----+
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

```
+-----+  
| No data returned. |  
+-----+  
Property existence constraints removed: 1
```

Create a relationship that complies with property existence constraints

Create a **LIKED** relationship with a **day** property.

Query

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

Result

```
+-----+  
| No data returned. |  
+-----+  
Nodes created: 2  
Relationships created: 1  
Properties set: 1  
Labels added: 2
```

Create a relationship that violates 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`
```

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`
```

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

Create a constraint on the property `day` on relationships with the `LICKED` 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` must have the property `day`
```

9.2.5. Node Keys

Create a Node Key

To create a Node Key ensuring that all nodes with a particular label have a set of defined properties whose combined value is unique, and where all properties in the set are present, use the `ASSERT (variable.propertyName_1, ..., variable.propertyName_n) IS NODE KEY` syntax.

Query

```
CREATE CONSTRAINT ON (n:Person) ASSERT (n.firstname, n.surname) IS NODE KEY
```

Result

```
+-----+  
| No data returned, and nothing was changed. |  
+-----+
```

Drop a Node Key

Use `DROP CONSTRAINT` to remove a Node Key from the database.

Query

```
DROP CONSTRAINT ON (n:Person) ASSERT (n.firstname, n.surname) IS NODE KEY
```

Result

```
+-----+  
| No data returned, and nothing was changed. |  
+-----+
```

Create a node that complies with a Node Key

Create a `Person` node with both a `firstname` and `surname` property.

Query

```
CREATE (p:Person { firstname: 'John', surname: 'Wood', age: 55 })
```

Result

```
+-----+
| No data returned. |
+-----+
Nodes created: 1
Properties set: 3
Labels added: 1
```

Create a node that violates a Node Key

Trying to create a `Person` node without a `surname` property, given a Node Key on `:Person(firstname, surname)`, will fail.

Query

```
CREATE (p:Person { firstname: 'Jane', age: 34 })
```

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

Error message

```
Node(1) with label `Person` must have the properties `firstname, surname`
```

Failure to create a Node Key due to existing node

Trying to create a Node Key on the property `surname` on nodes with the `Person` label will fail when a node without a `surname` already exists in the database.

Query

```
CREATE CONSTRAINT ON (n:Person) ASSERT (n.firstname, n.surname) IS NODE KEY
```

In this case the Node Key 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 ( person:Person ) ASSERT exists(person.firstname,
person.surname):
Node(0) with label `Person` must have the properties `firstname, surname`
```

9.3. Statistics

- [Introduction](#)
- [Configuration options](#)
- [Manual index sampling](#)

9.3.1. Introduction

When you issue a Cypher query, it gets compiled to an execution plan (see [Execution plans](#)) 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.

9.3.2. 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.

`dbms.index_sampling.background_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.



If background sampling is turned off, make sure to trigger manual sampling when data has been updated.

`dbms.index_sampling.update_percentage`

Controls how large portion of the index has to have been updated before a new sampling run is triggered.

`cypher.statistics_divergence_threshold`

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*.

9.3.3. Manual index sampling

Index resampling can be triggered using two built-in procedures `db.resampleIndex()` and `db.resampleOutdatedIndexes()`.

Here is an example of using `cypher-shell` to trigger resampling.

```
> cypher-shell 'CALL db.resampleIndex(":Person(name)");'  
> cypher-shell 'CALL db.resampleOutdatedIndexes();'
```

Chapter 10. Query tuning

This section describes query tuning for the Cypher query language.

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 [Execution plans](#).

- [How are queries executed?](#)
- [Profiling a query](#)
- [Basic query tuning example](#)
- [Planner hints and the USING keyword](#)
 - [Introduction](#)
 - [Index hints](#)
 - [Scan hints](#)
 - [Join hints](#)
 - [PERIODIC COMMIT query hint](#)

10.1. 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 3.2 will use the cost planner for all queries. You can force it to use a specific planner by using the `cypheplanner` configuration setting (see [Operations Manual Configuration Settings](#)), or by prepending your query with `CYPHER planner=cost` or `CYPHER planner=rule`.

You can see which planner was used by looking at the execution plan.



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.

10.2. Profiling a query

There are two options to choose from when you want to analyze a query by looking at its execution plan:

EXPLAIN

If you want to see the execution plan but not run the statement, prepend your Cypher statement with `EXPLAIN`. The statement will always return an empty result and make no changes to the database.

PROFILE

If you want to run the statement and see which operators are doing most of the work, use `PROFILE`. This will run your statement and keep track of how many rows pass through each operator, and how much each operator needs to interact with the storage layer to retrieve the necessary data. Please note that *profiling your query uses more resources*, so you should not profile unless you are actively working on a query.

See [Execution plans](#) for a detailed explanation of each of the operators contained in an execution plan.



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.

10.3. 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 '{csv-dir}/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 '{csv-dir}/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 '{csv-dir}/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 [Profiling a query](#) but in this case we're going to prefix our query with **PROFILE**:

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

Compiler CYPHER 3.2

Planner COST

Runtime INTERPRETED

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

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
```

Compiler CYPHER 3.2

Planner COST

Runtime INTERPRETED

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	13	1	0	0	0	p
+Filter	13	1	125	0	0	p
p.name == { AUTOSTRING0 }						
+NodeByLabelScan	125	125	126	0	0	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
```

```
Compiler CYPHER 3.2
```

```
Planner COST
```

```
Runtime INTERPRETED
```

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	1	1	0	0	0	p
+NodeIndexSeek	1	1	2	0	0	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 [Indexes](#)) to find the appropriate node.

10.4. Planner hints and the USING keyword

A planner hint is used to influence the decisions of the planner when building an execution plan for a query. Planner hints are specified in a query with the `USING` keyword.



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.

- [Introduction](#)
- [Index hints](#)
- [Scan hints](#)
- [Join hints](#)
- [PERIODIC COMMIT query hint](#)

10.4.1. Introduction

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, or other starting points.

However, the selected index might not always be the best choice. Sometimes multiple indexes are possible candidates, and the query planner picks the wrong one from a performance point of view. Moreover, in some circumstances (albeit rarely) it is better not to use an index at all.

Neo4j can be forced to use a specific starting point through the `USING` keyword. This is called giving a planner hint. There are four types of planner hints: index hints, scan hints, join hints, and the `PERIODIC COMMIT` query hint.



You cannot use planner hints if your query has a `START` clause.

The following graph is used for the examples below:

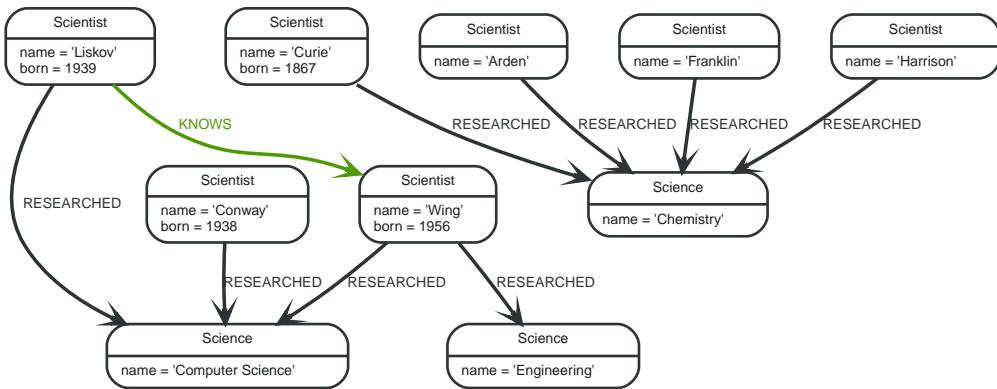


Figure 26. Graph

Query

```

MATCH (liskov:Scientist { name:'Liskov' })-[:KNOWS]->(wing:Scientist)-[:RESEARCHED]->(cs:Science {
  name: 'Computer Science' })<-[RESEARCHED]-(conway:Scientist { name: 'Conway' })
RETURN 1 AS column
  
```

The following query will be used in some of the examples on this page. It has intentionally been constructed in such a way that the statistical information will be inaccurate for the particular subgraph that this query matches. For this reason, it can be improved by supplying planner hints.

Query plan

Operator Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Time (ms)	Other
+ProduceResults	0	1	0	3		0	0.064
anon[122], anon[41], anon[68], column, conway, cs, liskov, wing							
+Projection	0	1	0	3		0	0.070
column -- anon[122], anon[41], anon[68], conway, cs, liskov, wing {column : { AUTOINT3}}							
+Filter	0	5	4	9		0	0.117
anon[122], anon[41], anon[68], conway, cs, liskov, wing NOT(anon[122] == anon[68]); conway:Scientist; conway.name == { AUTOSTRING2}							
+Expand(All)	0	3	4	3		0	0.180
anon[122], conway -- anon[41], anon[68], cs, liskov, wing (cs)<-[RESEARCHED]-(conway)							
+Filter	0	3	4	6		0	0.251
anon[41], anon[68], cs, liskov, wing cs:Science; cs.name == { AUTOSTRING1}							
+Expand(All)	0	2	3	3		0	0.330
anon[68], cs -- anon[41], liskov, wing (wing)-[:RESEARCHED]->(cs)							
+Filter	0	1	1	3		0	0.346
anon[41], liskov, wing wing:Scientist							
+Expand(All)	0	1	2	3		0	0.475
anon[41], wing -- liskov (liskov)-[:KNOWS]->(wing)							
+NodeIndexSeek	1	1	2	3		0	0.857
liskov :Scientist(name)							

Total database accesses: 20

10.4.2. Index hints

Index hints are used to specify which index, if any, the planner should use as a starting point. This can be beneficial in cases where the index statistics are not accurate for the specific values that the query at hand is known to use, which would result in the planner picking a non-optimal index. To supply an index hint, use `USING INDEX variable:Label(property)` after the applicable `MATCH` clause.

It is possible to supply several index hints, but keep in mind that several starting points will require the use of a potentially expensive join later in the query plan.

Query using an index hint

The query above will not naturally pick an index to solve the plan. This is because the graph is very small, and label scans are faster for small databases. In general, however, query performance is ranked by the dbhit metric, and we see that using an index is slightly better for this query.

Query

```
MATCH (liskov:Scientist { name:'Liskov' })-[:KNOWS]->(wing:Scientist)-[:RESEARCHED]->(cs:Science {  
name: 'Computer Science' })-<[:RESEARCHED]->(conway:Scientist { name: 'Conway' })  
USING INDEX liskov:Scientist(name)  
RETURN liskov.born AS column
```

Returns the year '**Barbara Liskov**' was born.

Query plan

Operator Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Time (ms)	Other
+ProduceResults	0	1	0	0	0	0.081	anon[122], anon[41], anon[68], column, conway, cs, liskov, wing
+Projection	0	1	1	0	0	0.109	column -- anon[122], anon[41], anon[68], conway, cs, liskov, wing {column : liskov.born}
+Filter	0	5	4	0	0	0.159	anon[122], anon[41], anon[68], conway, cs, liskov, wing NOT(anon[122] == anon[68]); conway:Scientist; conway.name == { AUTOSTRING2}
+Expand(All)	0	3	4	0	0	0.222	anon[122], conway -- anon[41], anon[68], cs, liskov, wing (cs)<-[RESEARCHED]-(conway)
+Filter	0	3	4	0	0	0.293	anon[41], anon[68], cs, liskov, wing cs:Science; cs.name == { AUTOSTRING1}
+Expand(All)	0	2	3	0	0	0.395	anon[68], cs -- anon[41], liskov, wing (wing)-[RESEARCHED]->(cs)
+Filter	0	1	1	0	0	0.411	anon[41], liskov, wing wing:Scientist
+Expand(All)	0	1	2	0	0	0.508	anon[41], wing -- liskov (liskov)-[KNOWS]->(wing)
+NodeIndexSeek	1	1	2	0	0	0.828	liskov :Scientist(name)

Total database accesses: 21

Query using multiple index hints

Supplying one index hint changed the starting point of the query, but the plan is still linear, meaning it only has one starting point. If we give the planner yet another index hint, we force it to use two starting points, one at each end of the match. It will then join these two branches using a join operator.

Query

```
MATCH (liskov:Scientist { name:'Liskov' })-[:KNOWS]->(wing:Scientist)-[:RESEARCHED]->(cs:Science {  
name: 'Computer Science' })-<[:RESEARCHED]->(conway:Scientist { name: 'Conway' })  
USING INDEX liskov:Scientist(name)  
USING INDEX conway:Scientist(name)  
RETURN liskov.born AS column
```

Returns the year 'Barbara Liskov' was born, using a slightly better plan.

Query plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Time (ms)	Variables
							Other
+ProduceResults	0	1	0	0	0	0.062	anon[122], anon[41], anon[68], column, conway, cs, liskov, wing
+Projection	0	1	1	0	0	0.094	column -- anon[122], anon[41], anon[68], conway, cs, liskov, wing {column : liskov.born}
+Filter	0	1	0	0	0	0.099	anon[122], anon[41], anon[68], conway, cs, liskov, wing NOT(anon[122] == anon[68])
+NodeHashJoin	0	1	0	0	0	0.494	anon[41], anon[68], liskov, wing -- anon[122], conway, cs cs
+Filter	1	1	1	0	0	0.163	anon[122], conway, cs cs.name == { AUTOSTRING1 }
+Expand(All)	1	1	2	0	0	0.191	anon[122], cs -- conway (conway)-[:RESEARCHED]->(cs)
+NodeIndexSeek	1	1	2	0	0	0.336	conway :Scientist(name)
+Filter	0	3	4	0	0	0.431	anon[41], anon[68], cs, liskov, wing cs:Science; cs.name == { AUTOSTRING1 }
+Expand(All)	0	2	3	0	0	0.490	anon[68], cs -- anon[41], liskov, wing (wing)-[:RESEARCHED]->(cs)
+Filter	0	1	1	0	0	0.505	

```

anon[41], liskov, wing | wing:Scientist
| |
| | +-----+-----+-----+
+-----+
| +Expand(All) | 0 | 1 | 2 | 0 | 0 | 0.661 |
anon[41], wing -- liskov | (liskov)-[:KNOWS]->(wing)
| |
| | +-----+-----+-----+
+-----+
| +NodeIndexSeek | 1 | 1 | 2 | 0 | 0 | 0.985 |
liskov | :Scientist(name)
| |
+-----+
+-----+
Total database accesses: 18

```

10.4.3. Scan hints

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 variable:Label` after the applicable `MATCH` clause. This will force Cypher to not use an index that could have been used, and instead do a label scan.

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 (s:Scientist)
USING SCAN s:Scientist
WHERE s.born < 1939
RETURN s.born AS column

```

Returns all scientists born before '1939'.

Query plan

```

+-----+-----+-----+-----+-----+
| Operator | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Variables |
Other | |
+-----+-----+-----+-----+-----+
| +ProduceResults | 0 | 2 | 0 | 6 | 0 | column, s |
| |
| | +-----+-----+-----+
+-----+
| +Projection | 0 | 2 | 2 | 6 | 0 | column -- s |
{column : s.born}
| |
| | +-----+-----+-----+
+-----+
| +Filter | 0 | 2 | 7 | 6 | 0 | s |
AndedPropertyComparablePredicates(s,s.born,s.born < { AUTOINT0 }) |
| |
| | +-----+-----+-----+
+-----+
| +NodeByLabelScan | 7 | 7 | 8 | 8 | 0 | s |
:Scientist |
+-----+
+-----+
Total database accesses: 17

```

10.4.4. Join hints

Join hints are the most advanced type of hints, and are not used to find starting points for the query execution plan, but to enforce that joins are made at specified points. This implies that there has to be more than one starting point (leaf) in the plan, in order for the query to be able to join the two branches ascending from these leaves. Due to this nature, joins, and subsequently join hints, will force the planner to look for additional starting points, and in the case where there are no more good ones, potentially pick a very bad starting point. This will negatively affect query performance. In other cases, the hint might force the planner to pick a *seemingly* bad starting point, which in reality proves to be a very good one.

Hinting a join on a single node

In the example above using multiple index hints, we saw that the planner chose to do a join on the `cs` node. This means that the relationship between `wing` and `cs` was traversed in the outgoing direction, which is better statistically because the pattern `(:RESEARCHED)->(:Science)` is more common than the pattern `(:Scientist)-[:RESEARCHED]->()`. However, in the actual graph, the `cs` node only has two such relationships, so expanding from it will be beneficial to expanding from the `wing` node. We can force the join to happen on `wing` instead with a join hint.

Query

```
MATCH (liskov:Scientist { name:'Liskov' })-[:KNOWS]->(wing:Scientist)-[:RESEARCHED]->(cs:Science { name:'Computer Science' })<-[:RESEARCHED]-(conway:Scientist { name: 'Conway' })
USING INDEX liskov:Scientist(name)
USING INDEX conway:Scientist(name)
USING JOIN ON wing
RETURN wing.born AS column
```

Returns the birth date of 'Jeanette Wing', using a slightly better plan.

Query plan

Operator Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Time (ms)	Other
+ProduceResults anon[122], anon[41], anon[68], column, conway, cs, liskov, wing	0	1	0	0	0	0.206	
+Projection column -- anon[122], anon[41], anon[68], conway, cs, liskov, wing {column : wing.born}	0	1	1	0	0	0.322	
+NodeHashJoin anon[41], liskov -- anon[122], anon[68], conway, cs, wing wing	0	1	0	0	0	0.758	
+Filter anon[122], anon[68], conway, cs, wing NOT(anon[122] == anon[68])	1	2	0	0	0	0.468	
+Expand(All) anon[68], wing -- anon[122], conway, cs (cs)<-[:RESEARCHED]-(wing)	1	3	4	0	0	0.599	

```

+-----+
| | +Filter | 0 | 2 | 2 | 0 | 0 | 0.727 |
anon[122], conway, cs
| |
| | | +-----+
+-----+
| | +Expand(All) | 1 | 1 | 2 | 0 | 0 | 0.831 |
anon[122], cs -- conway
| |
| | | +-----+
+-----+
| | +NodeIndexSeek | 1 | 1 | 2 | 0 | 0 | 1.221 |
conway
| |
| | | +-----+
+-----+
| +Filter | 0 | 1 | 1 | 0 | 0 | 0.425 |
anon[41], liskov, wing
| |
| | | +-----+
+-----+
| +Expand(All) | 0 | 1 | 2 | 0 | 0 | 0.699 |
anon[41], wing -- liskov
| |
| | | +-----+
+-----+
| +NodeIndexSeek | 1 | 1 | 2 | 0 | 0 | 1.999 |
liskov
| |
+-----+
+-----+

```

Total database accesses: 16

Hinting a join on multiple nodes

The query

```
Query

MATCH (liskov:Scientist { name:'Liskov' })-[:KNOWS]->(wing:Scientist { name:'Wing' })-[:RESEARCHED]->(cs:Science { name:'Computer Science' })<-[:RESEARCHED]-(liskov)
USING INDEX liskov:Scientist(name)
USING JOIN ON liskov, cs
```

Returns the birth date of 'Janette Wing'.

Query plan

Operator Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits Other	Page Cache Misses	Time (ms)
+ProduceResults anon[136], anon[41], anon[82], column, cs, liskov, wing	0	1	0	0	0	0.061
+Projection column -- anon[136], anon[41], anon[82], cs, liskov, wing {column : wing.born}	0	1	1	0	0	0.092
+Filter anon[136], anon[41], anon[82], cs, liskov, wing NOT(anon[136] == anon[82])	0	1	0	0	0	0.097
+NodeHashJoin anon[41], anon[82], wing -- anon[136], cs, liskov liskov, cs	0	1	0	0	0	0.524
+Filter anon[136], cs, liskov cs:Science; cs.name == { AUTOSTRING2}	1	2	2	0	0	0.175
+Expand(All) anon[136], cs -- liskov (liskov)-[:RESEARCHED]->(cs)	1	1	2	0	0	0.209
+NodeIndexSeek liskov :Scientist(name)	1	1	2	0	0	0.365
+Filter anon[41], anon[82], cs, liskov, wing cs:Science; cs.name == { AUTOSTRING2}	0	3	4	0	0	0.463
+Expand(All) anon[82], cs -- anon[41], liskov, wing (wing)-[:RESEARCHED]->(cs)	0	2	3	0	0	0.533
+Filter anon[41], liskov, wing wing:Scientist; wing.name == { AUTOSTRING1}	0	2	2	0	0	0.584
+Expand(All) anon[41], wing -- liskov (liskov)-[:KNOWS]->(wing)	0	1	2	0	0	0.691
+NodeIndexSeek liskov :Scientist(name)	1	1	2	0	0	1.022

Total database accesses: 20

10.4.5. PERIODIC COMMIT query hint



See [Importing CSV files with Cypher](#) 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`. If required, the limit for the number of rows per commit may be set as follows: `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 [Importing large amounts of data in LOAD CSV](#) for examples of `USING PERIODIC COMMIT` with and without setting the number of rows per commit.



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 11. Execution plans

This section describes operators used as part of an execution plan to execute a query in the Cypher query language.

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 *database hit* is an abstract unit of this storage engine work.

See [Profiling a query](#) 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.

- [Execution plan operators at a glance](#)
- [Starting point operators](#)
- [Expand operators](#)
- [Combining operators](#)
- [Row operators](#)
- [Update operators](#)
- [Shortest path planning](#)

Execution plan operators at a glance

This table comprises all the execution plan operators ordered lexicographically.

Name	Operator type	Description
AllNodesScan	Starting point	Reads all nodes from the node store.
AntiConditionalApply	Combining	Checks whether a variable is <code>null</code> , and if so the right-hand side will be executed.
AntiSemiApply	Combining	Tests for the existence of a pattern predicate.
Apply	Combining	Apply works by performing a nested loop.
AssertSameNode	Combining	This operator is used to ensure that no uniqueness constraints are violated.

Name	Operator type	Description
ConditionalApply	Combining	Checks whether a variable is not <code>null</code> , and if so the right-hand side will be executed.
Constraint operation	Update	Creates a constraint on a (label,property) pair.
DirectedRelationshipByIdSeek	Starting point	Reads one or more relationships by id from the relationship store.
Distinct	Row	Removes duplicate rows from the incoming stream of rows.
Eager	Row	For isolation purposes, <code>Eager</code> ensures that operations affecting subsequent operations are executed fully for the whole dataset before continuing execution.
EagerAggregation	Row	Eagerly loads underlying results and stores it in a hashmap, using the grouping keys as the keys for the map.
EmptyResult	Update	Eagerly loads everything coming in to the <code>EmptyResult</code> operator and discards it.
Expand(All)	Expand	Given a start node, <code>Expand(All)</code> will follow incoming or outgoing relationships, depending on the pattern relationship.
Expand(Into)	Expand	When both the start and end node have already been found, <code>Expand(Into)</code> is used to find all connecting relationships between the two nodes.
Filter	Row	Filters each row coming from the child operator, only passing through rows that evaluate the predicates to <code>true</code> .
LetAntiSemiApply	Combining	Tests for the absence of a pattern predicate in queries containing multiple pattern predicates.
LetSemiApply	Combining	Tests for the existence of a pattern predicate in queries containing multiple pattern predicates.
Limit	Row	Returns the first 'n' rows from the incoming input.
Merge(Into)	Update	When both the start and end node have already been found, <code>Merge(Into)</code> is used to find all connecting relationships or creating a new relationship between the two nodes.
NodeByIdSeek	Starting point	Reads one or more nodes by id from the node store.
NodeByLabelScan	Starting point	Using the label index, fetches all nodes with a specific label on them from the node label index.
NodeCountFromCountStore	Row	Uses the count store to answer questions about node counts.
NodeHashJoin	Combining	Using a hash table, a <code>NodeHashJoin</code> joins the input coming from the left with the input coming from the right.
NodeIndexContainsScan	Starting point	An index contains scan goes through all values stored in an index, and searches for entries containing a specific string.

Name	Operator type	Description
NodeIndexScan	Starting point	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.
NodeIndexSeek	Starting point	Finds nodes using an index seek.
NodeIndexSeekByRange	Starting point	Finds nodes using an index seek where the value of the property matches a given prefix string.
NodeUniqueIndexSeek	Starting point	Finds nodes using an index seek within a unique index.
OptionalExpand(All)	Expand	Traverses relationships from a given node, and ensures that predicates are evaluated before producing rows.
ProcedureCall	Row	Calls a procedure.
Projection	Row	For each row from its input, projection evaluates a set of expressions and produces a row with the results of the expressions.
RelationshipCountFromCountStore	Row	Uses the count store to answer questions about relationship counts.
SelectOrAntiSemiApply	Combining	Tests for the absence of a pattern predicate and evaluates a predicate.
SelectOrSemiApply	Combining	Tests for the existence of a pattern predicate and evaluates a predicate.
SemiApply	Combining	Tests for the existence of a pattern predicate.
Skip	Row	Skips 'n' rows from the incoming rows.
Sort	Row	Sorts rows by a provided key.
Top	Row	Returns the first 'n' rows sorted by a provided key.
Triadic	Combining	Triadic is used to solve triangular queries, such as the very common 'find my friend-of-friends that are not already my friend'.
UndirectedRelationshipByIdSeek	Starting point	Reads one or more relationships by id from the relationship store.
Union	Row	Concatenates the results from the right plan after the results of the left plan.
Unwind	Row	Takes a list of values and returns one row per item in the list.
UpdateGraph	Update	Applies updates to the graph.

11.1. Starting point operators

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

Operators:

- [AllNodesScan](#)
- [DirectedRelationshipByIdSeek](#)
- [NodeByIdSeek](#)

- [NodeByLabelScan](#)
- [NodeIndexSeek](#)
- [NodeUniqueIndexSeek](#)
- [NodeIndexSeekByRange](#)
- [NodeIndexContainsScan](#)
- [NodeIndexScan](#)
- [UndirectedRelationshipByIdSeek](#)

11.1.1. All Nodes Scan

Reads all nodes from the node store. The variable 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	Page Cache Hits	Page Cache Misses	Variables
+ProduceResults	35	35	0	68	0	n
+AllNodesScan	35	35	36	70	0	n

Total database accesses: 36

11.1.2. 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	Page Cache Hits	Page Cache Misses	Variables
+ProduceResults	1	1	0	0	0	r, n1
+DirectedRelationshipByIdSeekPipe	1	1	1	1	1	anon[17], n1, r EntityByIdRhs(SingleSeekArg({ AUTOINT0}))

Total database accesses: 1

11.1.3. 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	Page Cache Hits	Page Cache Misses	Variables
+ProduceResults	1	1	0	0	0	n
+NodeByIdSeek	1	1	1	1	0	n

Total database accesses: 1

11.1.4. 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	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	14	14	0	13	0	person
+NodeByLabelScan	14	14	15	15	0	person

Total database accesses: 15

11.1.5. Node index seek

Finds nodes using an index seek. The node variable 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	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	1	1	0	0	0	location
+NodeIndexSeek	1	1	2	1	0	location
:Location(name)						
Total database accesses: 2						

11.1.6. 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 l
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
Variables Other						
+ProduceResults	26	1	0	0	0	1
+NodeIndexSeekByRange	26	1	2	1	0	1
:Location(name STARTS WITH { AUTOSTRING0})						
Total database accesses: 2						

11.1.7. Node index contains scan

An index contains scan goes through all values stored in an index, and searches for entries containing a specific string. This is slower than an index seek, since all entries need to be examined, but still faster than the indirection needed by a label scan and then a property store filter.

Query

```
MATCH (l:Location)
WHERE l.name CONTAINS 'al'
RETURN l
```

Query Plan

Operator Variables Other	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses
+ProduceResults	10	2	0	1	0 1
+NodeIndexContainsScan	10	2	3	2	0 1
:Location(name); Parameter(AUTOSTRING0, String)					
Total database accesses: 3					

11.1.8. 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 exists(l.name)
RETURN l
```

Query Plan

Operator Other	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
+ProduceResults	10	10	0	9	0 1	
+NodeIndexScan	10	10	11	10	0 1	
:Location(name)						
Total database accesses: 11						

11.1.9. 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	Page Cache Hits	Page Cache Misses	Variables
+ProduceResults	1	2	0	0	0	anon[16], n1, r
+UndirectedRelationshipByIdSeek	1	2	1	1	0	anon[16], n1, r
Total database accesses:	1					

11.2. Expand operators

These operators explore the graph by expanding graph patterns.

Operators:

- [Expand\(All\)](#)
- [Expand\(Into\)](#)
- [OptionalExpand\(All\)](#)

11.2.1. Expand All

Given a start node, expand-all will follow incoming or outgoing relationships, 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	Page Cache Hits	Page Cache Misses	Variables
+ProduceResults	0	1	0	2	0	anon[30], fof,
+Expand(All)	0	1	2	2	0	anon[30], fof
-- p (p)-[:FRIENDS_WITH]->(fof)						
+NodeIndexSeek	1	1	2	3	0	p
:Person(name)						
Total database accesses:	4					

11.2.2. 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	Page Cache Hits	Page Cache Misses	Variables
Operator						
Other						
+ProduceResults	0	0	0	2	0	anon[30],
anon[53], fof, p						
+Filter	0	0	0	2	0	anon[30],
anon[53], fof, p						
NOT(anon[30] == anon[53])						
+Expand(Into)	0	0	0	2	0	anon[30] --
anon[53], fof, p (p)-[:FRIENDS_WITH]->(fof)						
+Expand(All)	0	0	1	2	0	anon[53], fof
-- p (p)<--(fof)						
+NodeIndexSeek	1	1	2	3	0	p
:Person(name)						
Total database accesses: 3						

11.2.3. 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 variable 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	Page Cache Hits	Page Cache Misses	Variables
Other						
+-----						
+ProduceResults works_in	14	15	0	16	0	l, p,
+-----						
+OptionalExpand(All) works_in -- p AndedPropertyInequalities(Property(Variable(works_in), PropertyKeyName(duration)), GreaterThan(Parameter(AUTOINT0, Integer))); (p)-[works_in:WORKS_IN]->(1)	14	15	44	16	0	l,
+-----						
+NodeByLabelScan :Person	14	14	15	18	0	p
+-----						
+-----						
Total database accesses: 59						

11.3. Combining operators

The combining operators are used to piece together other operators.

Operators:

- [Apply](#)
- [SemiApply](#)
- [AntiSemiApply](#)
- [LetSemiApply](#)
- [LetAntiSemiApply](#)
- [SelectOrSemiApply](#)
- [SelectOrAntiSemiApply](#)
- [ConditionalApply](#)
- [AntiConditionalApply](#)
- [AssertSameNode](#)
- [NodeHashJoin](#)
- [Triadic](#)

The following graph is used for the examples below:

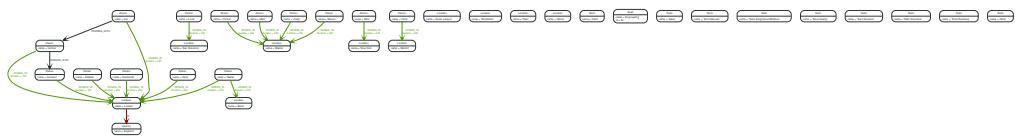


Figure 27. Graph

11.3.1. 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 leaf operator on the right-hand side, and then **Apply** will yield the combined results. **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 > 2
OPTIONAL MATCH (p)-[:WORKS_IN]->(city)
RETURN city.name
```

Finds all the people with more than two friends and returns the city they work in.

Query plan

Operator	Estimated Rows	Variables	Other
+ProduceResults	1 anon[70], anon[93], city, city.name, fs, p		
+Projection	1 city.name -- anon[70], anon[93], city, fs, p {city.name : city.name}		
+OptionalExpand(All)	1 anon[93], city -- anon[70], fs, p (p)-[:WORKS_IN]->(city)		
+Filter	1 anon[70], fs, p anon[70]		
+Projection	1 anon[70] -- fs, p {p : p, fs : fs, : fs > { AUTOINT0 }}		
+EagerAggregation	1 fs -- p p		
+Expand(All)	2 anon[17], f -- p (p)-[:FRIENDS_WITH]->(f)		
+NodeByLabelScan	14 p :Person		

Total database accesses: ?

11.3.2. SemiApply

Tests for the existence of a pattern predicate. [SemiApply](#) takes a row from its child operator and feeds it to the leaf operator on the right-hand side. 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 (p:Person)
WHERE (p)-[:FRIENDS_WITH]->()
RETURN p.name
```

Finds all the people who have friends.

Query plan

Operator	Estimated Rows	Variables	Other
+ProduceResults	4	p, p.name	
+Projection	4	p.name -- p {p.name : p.name}	
+Filter	4	p GetDegree(p, Some(FRIENDS_WITH), OUTGOING) > Literal(0)	
+NodeByLabelScan	14	p :Person	

Total database accesses: ?

11.3.3. AntiSemiApply

Tests for the absence of a pattern. [AntiSemiApply](#) takes a row from its child operator and feeds it to the leaf operator on the right-hand side. If the right-hand side operator tree yields at least one row, the row from the left-hand side is yielded by the [AntiSemiApply](#) operator. This makes [AntiSemiApply](#) a filtering operator, used for pattern predicates in queries.

Query

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

Finds the names of all the people who are not my friends.

Query plan

Operator	Estimated Rows	Variables	Other
+ProduceResults	4	me, other, other.name	
+Projection	4	other.name -- me, other	{other.name : other.name}
+AntiSemiApply	4	me, other	
\+Expand(Into)	0	anon[62] -- me, other	(me)-[:FRIENDS_WITH]->(other)
+Argument	14	me, other	
+CartesianProduct	14	me -- other	
\+NodeByLabelScan	14	other	:Person
+NodeIndexSeek	1	me	:Person(name)

Total database accesses: ?

11.3.4. LetSemiApply

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 another operator.

Query

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

Finds the names of 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 plan

```
+-----+-----+
+-----+
| Operator      | Estimated Rows | Variables           | Other
|-----|
+-----+-----+
| +ProduceResults |             7 | other, other.name   |
| |               +-----+
+-----+
| +Projection    |             7 | other.name -- other | {other.name : other.name}
| |               +-----+
+-----+
| +Filter        |             7 | other
| Ors(GetDegree(other,Some(FRIENDS_WITH),OUTGOING) > Literal(0), GetDegree(other,Some(WORKS_IN),OUTGOING) >
| Literal(0)) |
| |               +-----+
+-----+
| +NodeByLabelScan |            14 | other                | :Person
| |               +-----+
+-----+
-----+
Total database accesses: ?
```

11.3.5. LetAntiSemiApply

Tests for the absence of a pattern. When a query contains multiple pattern predicates [LetAntiSemiApply](#) 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 friends or who work somewhere.

Query

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

Finds all the people who don't have any friends or who work somewhere. The [LetAntiSemiApply](#) operator will be used to check for the absence of the [FRIENDS_WITH](#) relationship from each person.

Query plan

```
+-----+-----+
+-----+
| Operator      | Estimated Rows | Variables           | Other
|-----|
+-----+-----+
| +ProduceResults |             7 | other, other.name   |
| |               +-----+
+-----+
| +Projection    |             7 | other.name -- other | {other.name : other.name}
| |               +-----+
+-----+
| +Filter         |             7 | other
| Ors(GetDegree(other,Some(FRIENDS_WITH),OUTGOING) <= Literal(0), GetDegree(other,Some(WORKS_IN),OUTGOING) > Literal(0)) |
| |               +-----+
+-----+
| +NodeByLabelScan |            14 | other                | :Person
| |               +-----+
+-----+
-----+
Total database accesses: ?
```

11.3.6. SelectOrSemiApply

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 existence 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.name
```

Finds the names of all people who have friends, or are older than '25'.

Query plan

```
+-----+-----+
| Operator      | Estimated Rows | Variables           | Other
|
+-----+-----+
| +ProduceResults |          4 | other, other.name   |
|
| |           +-----+
+-----+
| +Projection    |          4 | other.name -- other | {other.name : other.name}
|
| |           +-----+
+-----+
| +Filter        |          4 | other               | Ors(other.age > { AUTOINT0},
GetDegree(other,Some(FRIENDS_WITH),OUTGOING) > Literal(0)) |
|
| |           +-----+
+-----+
| +NodeByLabelScan |         14 | other               | :Person
|
+-----+-----+
+
Total database accesses: ?
```

11.3.7. SelectOrAntiSemiApply

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.name
```

Finds the names of all people who do not have friends, or are older than '25'.

Query plan

```
+-----+-----+
| Operator      | Estimated Rows | Variables           | Other
|
+-----+-----+
| +ProduceResults |          4 | other, other.name   |
|
| |           +-----+
+-----+
| +Projection    |          4 | other.name -- other | {other.name : other.name}
|
| |           +-----+
+-----+
| +Filter        |          4 | other               | Ors(other.age > { AUTOINT0},
GetDegree(other,Some(FRIENDS_WITH),OUTGOING) <= Literal(0)) |
|
| |           +-----+
+-----+
| +NodeByLabelScan |         14 | other               | :Person
|
+-----+-----+
+
Total database accesses: ?
```

11.3.8. ConditionalApply

Checks whether a variable is not `null`, and if so the right-hand side will be executed.

Query

```
MERGE (p:Person { name: 'Andres' })
ON MATCH SET p.exists = TRUE
```

Looks for the existence of a person called '**Andres**', and if found sets the `exists` property to `true`.

Query plan

Operator	Estimated Rows	Variables	Other
+ProduceResults	1	p	
+EmptyResult	1	p	
+AntiConditionalApply	1	p	
\ \			
+MergeCreateNode	1	p	
+ConditionalApply	1	p	
\ \			
+SetProperty	1	p	
+Argument	1	p	
+Optional	1	p	
+NodeIndexSeek	1	p	:Person(name)

Total database accesses: ?

11.3.9. AntiConditionalApply

Checks whether a variable is `null`, and if so the right-hand side will be executed.

Query

```
MERGE (p:Person { name: 'Andres' })
ON CREATE SET p.exists = TRUE
```

Looks for the existence of a person called '**Andres**', and if not found, creates one and sets the `exists` property to `true`.

Query plan

Operator	Estimated Rows	Variables	Other
+ProduceResults	1	p	
+EmptyResult	1	p	
+AntiConditionalApply	1	p	
\ \			
+SetProperty	1	p	
+MergeCreateNode	1	p	
+Optional	1	p	
+NodeIndexSeek	1	p	:Person(name)

Total database accesses: ?

11.3.10. AssertSameNode

This operator is used to ensure that no uniqueness constraints are violated.

Query

```
MERGE (t:Team { name: 'Engineering', id: 42 })
```

Looks for the existence of a team with the supplied name and id, and if one does not exist, it will be created. Owing to the existence of two uniqueness constraints on `:Team(name)` and `:Team(id)`, any node that would be found by the `UniqueIndexSeek` must be the very same node, or the constraints would be violated.

Query plan

Operator	Estimated Rows	Variables	Other
+ProduceResults	1	t	
+EmptyResult	1	t	
+AntiConditionalApply	1	t	
+MergeCreateNode	1	t	
+Optional	1	t	
+AssertSameNode	0	t	
+NodeUniqueIndexSeek(Locking)	1	t	:Team(name)
+NodeUniqueIndexSeek(Locking)	1	t	:Team(id)

Total database accesses: ?

11.3.11. NodeHashJoin

Using a hash table, a `NodeHashJoin` joins the input coming from the left with the input coming from the right.

Query

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

Returns the name of the location where the matched persons both work.

Query plan

```
+-----+-----+
| Operator      | Estimated Rows | Variables                                | Other
|
+-----+-----+
| +ProduceResults |          10 | anon[37], anon[56], andy, loc, loc.name, matt |
|
| |           +-----+
+-----+
| +Projection   |          10 | loc.name -- anon[37], anon[56], andy, loc, matt | {loc.name : loc.name}
| |           +-----+
+-----+
| +Filter       |          10 | anon[37], anon[56], andy, loc, matt          | NOT(anon[37] == anon[56])
| |           +-----+
+-----+
| +NodeHashJoin |          10 | anon[37], andy -- anon[56], loc, matt        | loc
|
| | \           +-----+
+-----+
| | | +Expand(All) |          19 | anon[56], loc -- matt                      | (matt)-[:WORKS_IN]->(loc)
| | |           +-----+
+-----+
| | | +NodeIndexSeek |          1 | matt                                         | :Person(name)
|
| | |           +-----+
+-----+
| | | +Expand(All) |          19 | anon[37], loc -- andy                      | (andy)-[:WORKS_IN]->(loc)
| | |           +-----+
+-----+
| | | +NodeIndexSeek |          1 | andy                                         | :Person(name)
|
+-----+
+-----+
```

Total database accesses: ?

11.3.12. 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.name
```

Finds the names of all friends of my friends that are not already my friends.

Query plan

Operator	Estimated Rows	Variables	Other
+ProduceResults	0	anon[18], anon[35], anon[37], me, other, other.name	
+Projection	0	other.name -- anon[18], anon[35], anon[37], me, other {other.name}	
: other.name}			
+TriadicSelection	0	anon[18], anon[35], anon[37], me, other	me,
anon[35], other			
+Filter	0	anon[18], anon[35], anon[37], me, other	
NOT(anon[18] == anon[37])			
+Expand(All)	0	anon[37], other -- anon[18], anon[35], me	()-
[:FRIENDS_WITH]-(other)			
+Argument	4	anon[18], anon[35], me	
+Expand(All)	4	anon[18], anon[35] -- me	(me)-
[:FRIENDS_WITH]-()			
+NodeByLabelScan	14	me	:Person

Total database accesses: ?

11.4. Row operators

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

Operators:

- [Eager](#)
- [Distinct](#)
- [EagerAggregation](#)
- [NodeCountFromCountStore](#)
- [RelationshipCountFromCountStore](#)
- [Filter](#)
- [Limit](#)
- [Projection](#)
- [Skip](#)
- [Sort](#)
- [Top](#)

- [Union](#)
- [Unwind](#)
- [ProcedureCall](#)

11.4.1. 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 (a)-[r]-(b)
DELETE r,a,b
MERGE ()
```

Query Plan

Operator Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses
Other					
+ProduceResults anon[38], a, b, r	1	0	0	0	0
+EmptyResult anon[38], a, b, r	1	0	0	1764	0
+Apply r -- anon[38]	1	504	0	1764	0 a, b,
+AntiConditionalApply anon[38]	1	504	0	28	0
+MergeCreateNode anon[38]	1	0	0	0	0
+Optional anon[38]	35	504	0	28	0
+AllNodesScan anon[38]	35	504	540	49	0
+Eager r	36	36	0	1727	0 a, b,
+Delete(3) r	36	36	39	-111	0 a, b,
+Eager r	36	36	0	33	0 a, b,
+Expand(All) -- b (b)-[r:]->(a)	36	36	71	70	0 a, r
+AllNodesScan	35	35	36	72	0 b

Total database accesses: 686

11.4.2. Distinct

Removes duplicate rows from the incoming stream of rows.

Query

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

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	14	6	0	11	0	1
+Distinct	14	6	0	11	0	1
+Filter p p:Person	15	15	15	11	0	anon[19], 1,
+Expand(All) -- l (l)<-[>:WORKS_IN]-(p)	15	15	25	11	0	anon[19], p
+NodeByLabelScan :Location	10	10	11	13	0	1

Total database accesses: 51

11.4.3. 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 (l:Location)-[:WORKS_IN]-(p:Person)
RETURN l.name AS location, collect(p.name) AS people
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	4	6	0	0	0	location, people
+EagerAggregation	4	6	15	12	0	people -- location
	location					
+Projection	15	15	15	12	0	location -- anon[19], 1, p {location : l.name, p : p}
+Filter	15	15	15	12	0	anon[19], 1, p p:Person
+Expand(All)	15	15	25	12	0	anon[19], p -- 1 (1)<-[:WORKS_IN]-(p)
+NodeByLabelScan	10	10	11	14	0	1 :Location

Total database accesses: 81

11.4.4. Node Count From Count Store

Use the count store to answer questions about node counts. This is much faster than eager aggregation which achieves the same result by actually counting. However the count store only saves a limited range of combinations, so eager aggregation will still be used for more complex queries. For example, we can get counts for all nodes, and nodes with a label, but not nodes with more than one label.

Query

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

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	4	1	0	0	0	people
+NodeCountFromCountStore	4	1	1	0	0	people count((:Person)) AS people

Total database accesses: 1

11.4.5. Relationship Count From Count Store

Use the count store to answer questions about relationship counts. This is much faster than eager aggregation which achieves the same result by actually counting. However the count store only saves a limited range of combinations, so eager aggregation will still be used for more complex queries. For example, we can get counts for all relationships, relationships with a type, relationships with a label on one end, but not relationships with labels on both end nodes.

Query

```
MATCH (p:Person)-[r:WORKS_IN]->()
RETURN count(r) AS jobs
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
+ProduceResults	4	1	0	0	0	jobs
+RelationshipCountFromCountStore	4	1	2	0	0	jobs
count((:Person)-[:WORKS_IN]->()) AS jobs						

Total database accesses: 2

11.4.6. 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	Page Cache Hits	Page Cache Misses	Variables
+ProduceResults	14	0	0	15	0	p
+Filter	14	0	14	15	0	p
p.name =~ /{ AUTOSTRING0}/						
+NodeIndexScan	14	14	15	16	0	p
:Person(name)						

Total database accesses: 29

11.4.7. 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	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	3	3	0	3	0	p
+Limit	3	3	0	3	0	SignedDecimalIntegerLiteral(3)
+NodeByLabelScan	14	3	4	0	0	:Person

Total database accesses: 4

11.4.8. 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	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	1	1	0	0	0	greeting
+Projection	1	1	0	0	0	{greeting : { AUTOSTRING0}}

Total database accesses: 0

11.4.9. 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	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	14	13	0	0	0	anon[35], anon[59], p, p
+Projection	14	13	0	0	0	0 p -- anon[35], anon[59], p {p : }
+Skip	14	13	0	0	0	0 anon[35], anon[59], p Parameter(AUTOINT0, Integer)
+Sort	14	14	0	13	0	0 anon[35], anon[59], p anon[59]
+Projection	14	14	14	13	0	0 anon[59] -- anon[35], p { : , : .id}
+Projection	14	14	0	13	0	0 anon[35] -- p { : p@7}
+NodeByLabelScan	14	14	15	15	0	0 p :Person

Total database accesses: 29

11.4.10. 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	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	14	14	0	0	0	anon[24], anon[37], p, p
+Projection	14	14	0	0	0	anon[24], anon[37], p {p : }
+Sort	14	14	0	15	0	anon[24], anon[37], p anon[37]
+Projection	14	14	14	15	0	anon[24], p { : , : .name}
+Projection	14	14	0	15	0	{ : p@7} anon[24] -- p
+NodeByLabelScan	14	14	15	17	0	:Person p
:Person						

Total database accesses: 29

11.4.11. 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	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	2	2	0	0	0	anon[24], anon[37], p, p
+Projection	2	2	0	0	0	anon[24], anon[37], p {p : }
+Top	2	2	0	15	0	anon[24], anon[37], p anon[37]; SignedDecimalIntegerLiteral(2)
+Projection	14	14	14	15	0	anon[24], p { : , : .name}
+Projection	14	14	0	15	0	{ : p@7}
+NodeByLabelScan	14	14	15	17	0	:Person

Total database accesses: 29

11.4.12. 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	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	10	11	0	13	0	p.name
+Union	10	11	0	15	0	p.name
+Projection	1	1	1	0	0	p.name -- p
{p.name : p@83.name}						
+NodeByLabelScan	1	1	2	2	0	p
:Country						
+Projection	10	10	10	11	0	p.name -- p
{p.name : p@7.name}						
+NodeByLabelScan	10	10	11	13	0	p
:Location						

Total database accesses: 24

11.4.13. Unwind

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

Query

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

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	10	5	0	0	0	value
+Unwind	10	5	0	0	0	value
FunctionInvocation(Namespace(List()),FunctionName(range),false,Vector(SignedDecimalIntegerLiteral(1),SignedDecimalIntegerLiteral(5)))						

Total database accesses: 0

11.4.14. Call Procedure

Return all labels sorted by name

Query

```
CALL db.labels() YIELD label
RETURN *
ORDER BY label
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
Other			1			
+ProduceResults	10000	4	0	0	0	label
+Sort	10000	4	0	0	0	label
+ProcedureCall	10000	4	1	0	0	label
db.labels() :: (label :: String)						
Total database accesses: 1						

11.5. Update operators

These operators are used in queries that update the graph.

Operators:

- [Constraint operation](#)
- [EmptyResult](#)
- [UpdateGraph](#)
- [Merge\(Into\)](#)

11.5.1. 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
+CreateUniqueConstraint

Total database accesses: ?

11.5.2. Empty Result

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

Query

```
CREATE (:Person)
```

Query Plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
+ProduceResults	1	0	0	0	0	anon[8]
+EmptyResult	1	0	0	1	0	anon[8]
+CreateNode	1	1	2	1	0	anon[8]

Total database accesses: 2

11.5.3. Update Graph

Applies updates to the graph.

Query

```
CYPHER planner=rule
CREATE (:Person { name: 'Alistair' })
```

Query Plan

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

Total database accesses: 4

11.5.4. 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

```
CYPHER planner=rule
MATCH (p:Person { name: 'me' }),(f:Person { name: 'Andres' })
MERGE (p)-[:FRIENDS_WITH]->(f)
```

Query Plan

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

Total database accesses: 9

11.6. Shortest path planning

Shortest path finding in Cypher and how it is planned.

Planning shortest paths in Cypher can lead to different query plans depending on the predicates that need to be evaluated. Internally, Neo4j will use a fast bidirectional breadth-first search algorithm if the predicates can be evaluated whilst searching for the path. Therefore, this fast algorithm will always be certain to return the right answer when there are universal predicates on the path; for example, when searching for the shortest path where all nodes have the `Person` label, or where there are no nodes with a `name` property.

If the predicates need to inspect the whole path before deciding on whether it is valid or not, this fast algorithm cannot be relied on to find the shortest path, and Neo4j may have to resort to using a slower exhaustive depth-first search algorithm to find the path. This means that query plans for shortest path queries with non-universal predicates will include a fallback to running the exhaustive search to find the path should the fast algorithm not succeed. For example, depending on the data, an answer to a shortest path query with existential predicates — such as the requirement that at least one node contains the property `name='Charlie Sheen'` — may not be able to be found by the fast algorithm. In this case, Neo4j will fall back to using the exhaustive search to enumerate all paths and potentially return an answer.

The running times of these two algorithms may differ by orders of magnitude, so it is important to ensure that the fast approach is used for time-critical queries.

When the exhaustive search is planned, it is still only executed when the fast algorithm fails to find any matching paths. The fast algorithm is always executed first, since it is possible that it can find a valid path even though that could not be guaranteed at planning time.

Please note that falling back to the exhaustive search may prove to be a very time consuming strategy in some cases; such as when there is no shortest path between two nodes. Therefore, in these cases, it is recommended to set `cypher.forbid_exhaustive_shortestpath` to `true`, as explained in [Operations Manual Configuration settings](#)

11.6.1. Shortest path with fast algorithm

Query

```
MATCH (ms:Person { name: 'Martin Sheen' }), (cs:Person { name: 'Charlie Sheen' }), p = shortestPath((ms)-[rels:ACTED_IN*]-(cs))
WHERE ALL (r IN rels WHERE exists(r.role))
RETURN p
```

This query can be evaluated with the fast algorithm — there are no predicates that need to see the whole path before being evaluated.

Query plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
+-----+ Other +-----+						
+-----+ +ProduceResults rels +-----+	1	1	0	8	0	cs, ms, p,
+-----+ +ShortestPath cs, ms {p0 : AllIterablePredicate(FilterScope(Variable(r), Some(FunctionInvocation(Namespace(List())), FunctionName(exists), false, Vector(Property(Variable(r), PropertyKeyName(role))))), Variable(rels))} +-----+	1	1	9	8	0	p, rels -- cs, ms {p0 : AllIterablePredicate(FilterScope(Variable(r), Some(FunctionInvocation(Namespace(List())), FunctionName(exists), false, Vector(Property(Variable(r), PropertyKeyName(role))))), Variable(rels))}
+-----+ +CartesianProduct \ +-----+	1	1	0	9	0	ms -- cs
+-----+ +NodeIndexSeek :Person(name) +-----+	1	1	2	8	0	cs
+-----+ +NodeIndexSeek :Person(name) +-----+	1	1	2	9	0	ms
Total database accesses: 13						

11.6.2. Shortest path with additional predicate checks on the paths

Consider using the exhaustive search as a fallback

Predicates used in the `WHERE` clause that apply to the shortest path pattern are evaluated before deciding what the shortest matching path is.

Query

```
MATCH (cs:Person { name: 'Charlie Sheen' }),(ms:Person { name: 'Martin Sheen' }), p = shortestPath((cs)-[*]-(ms))
WHERE length(p)> 1
RETURN p
```

This query, in contrast with the one above, needs to check that the whole path follows the predicate before we know if it is valid or not, and so the query plan will also include the fallback to the slower exhaustive search algorithm

Query plan

Operator Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Other
+ProduceResults anon[95], anon[113], cs, ms, p	0	1	0	5	0	
+AntiConditionalApply anon[95], anon[113], cs, ms, p	0	1	0	5	0	
+Top anon[95], anon[113], cs, ms, p anon[95]; SignedDecimalIntegerLiteral(1)	0	0	0	0	0	
+Projection anon[95] -- anon[113], cs, ms, p { : length(p)}	0	0	0	0	0	
+Filter anon[113], cs, ms, p length(p) > { AUTOINT2}	0	0	0	0	0	
+Projection anon[113], cs, ms, p {p : ProjectedPath(Set(), cs),<function2>)}	0	0	0	0	0	
+VarLengthExpand(Into) anon[113], cs, ms, p (cs)-[:*]-(ms)	0	0	0	0	0	
+Argument anon[113], cs, ms, p	1	0	0	0	0	

```

+-----+
| +Apply
anon[113], cs, ms, p | 1 | 1 | 0 | 5 | 0 |
| | \
+-----+
+-----+
| | +Optional
anon[113], cs, ms, p | 1 | 1 | 0 | 4 | 0 |
| | |
+-----+
+-----+
| | +ShortestPath
anon[113], p -- cs, ms | {p0 : GreaterThan(FunctionInvocation(Namespace(List()),FunctionName(length),false,Vector(Variable(p))),Parameter(AUTOINT2,Integer)))} | 0 | 1 | 1 | 4 | 0 |
| | |
+-----+
+-----+
| | +Argument
ms | 1 | 1 | 0 | 4 | 0 | cs,
| |
| |
+-----+
+-----+
| +CartesianProduct
ms | 1 | 1 | 0 | 6 | 0 | cs --
| | \
+-----+
+-----+
| | +NodeIndexSeek
| :Person(name) | 1 | 1 | 2 | 5 | 0 | ms
| |
| |
+-----+
+-----+
| +NodeIndexSeek
| :Person(name) | 1 | 1 | 2 | 6 | 0 | cs
| |
+-----+
+-----+

```

The way the bigger exhaustive query plan works is by using `Apply/Optional` to ensure that when the fast algorithm does not find any results, a `null` result is generated instead of simply stopping the result stream. On top of this, the planner will issue an `AntiConditionalApply`, which will run the exhaustive search if the path variable is pointing to `null` instead of a path.

Prevent the exhaustive search from being used as a fallback

Query

```
MATCH (cs:Person { name: 'Charlie Sheen' }), (ms:Person { name: 'Martin Sheen' }), p = shortestPath((cs)-[*]->(ms))
WITH p
WHERE length(p) > 1
RETURN p
```

This query, just like the one above, needs to check that the whole path follows the predicate before we

know if it is valid or not. However, the inclusion of the `WITH` clause means that the query plan will not include the fallback to the slower exhaustive search algorithm. Instead, any paths found by the fast algorithm will subsequently be filtered, which may result in no answers being returned.

Query plan

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Variables
Other						
+ProduceResults	1	1	0	5	0	anon[146], anon[113], cs, ms, p
+Filter	1	1	0	5	0	anon[146], anon[113], cs, ms, p anon[146]
+Projection	1	1	0	5	0	anon[146] -- anon[113], cs, ms, p {p : p, : length(p) > { AUTOINT2}}
+ShortestPath	1	1	1	5	0	anon[113], p -- cs, ms {}
+CartesianProduct	1	1	0	6	0	cs -- ms
\						
+NodeIndexSeek	1	1	2	5	0	ms :Person(name)
+NodeIndexSeek	1	1	2	6	0	cs :Person(name)

Total database accesses: 5

Chapter 12. Glossary of keywords

This section comprises a glossary of all the keywords — grouped by category and thence ordered lexicographically — in the Cypher query language.

- [Clauses](#)
- [Operators](#)
- [Functions](#)
- [Expressions](#)

12.1. Clauses

Clause	Type	Description
<code>CALL [...YIELD]</code>	Reading/Writing	Invoke a procedure deployed in the database.
<code>CREATE</code>	Writing	Create nodes and relationships.
<code>CREATE CONSTRAINT ON (n:Label) ASSERT exists(n.property)</code>	Schema	Create a constraint ensuring that all nodes with a particular label have a certain property.
<code>CREATE CONSTRAINT ON ()-[r:REL_TYPE]-() ASSERT exists(r.property)</code>	Schema	Create a constraint ensuring that all relationship with a particular type have a certain property.
<code>CREATE CONSTRAINT ON (n:Label) ASSERT n.property IS UNIQUE</code>	Schema	Create a constraint ensuring the uniqueness of the combination of node label and property value for a particular property key across all nodes.
<code>CREATE CONSTRAINT ON (n:Label) ASSERT (n.prop1, ..., n.propN) IS NODE KEY</code>	Schema	Create a constraint ensuring all nodes with a particular label have all the specified properties and that the combination of property values is unique; i.e. ensures existence and uniqueness.
<code>CREATE INDEX ON :Label(property)</code>	Schema	Create an index on all nodes with a particular label and a single property; i.e. create a single-property index.
<code>CREATE INDEX ON :Label(prop1, ..., propN)</code>	Schema	Create an index on all nodes with a particular label and multiple properties; i.e. create a composite index.
<code>CREATE UNIQUE</code>	Reading/Writing	A mixture of <code>MATCH</code> and <code>CREATE</code> , matching what it can, and creating what is missing.
<code>DELETE</code>	Writing	Delete graph elements — nodes, relationships or paths. Any node to be deleted must also have all associated relationships explicitly deleted.
<code>DETACH DELETE</code>	Writing	Delete a node or set of nodes. All associated relationships will automatically be deleted.
<code>DROP CONSTRAINT ON (n:Label) ASSERT exists(n.property)</code>	Schema	Drop a constraint ensuring that all nodes with a particular label have a certain property.
<code>DROP CONSTRAINT ON ()-[r:REL_TYPE]-() ASSERT exists(r.property)</code>	Schema	Drop a constraint ensuring that all relationship with a particular type have a certain property.

Clause	Type	Description
DROP CONSTRAINT ON (n:Label) ASSERT n.property IS UNIQUE	Schema	Drop a constraint ensuring the uniqueness of the combination of node label and property value for a particular property key across all nodes.
DROP CONSTRAINT ON (n:Label) ASSERT (n.prop1, ..., n.propN) IS NODE KEY	Schema	Drop a constraint ensuring all nodes with a particular label have all the specified properties and that the combination of property values is unique.
DROP INDEX ON :Label(property)	Schema	Drop an index on all nodes with a particular label and a single property; i.e. drop a single-property index.
DROP INDEX ON :Label(prop1, ..., propN)	Schema	Drop an index on all nodes with a particular label and multiple properties; i.e. drop a composite index.
FOREACH	Writing	Update data within a list, whether components of a path, or the result of aggregation.
LIMIT	Reading sub-clause	A sub-clause used to constrain the number of rows in the output.
LOAD CSV	Importing data	Use when importing data from CSV files.
MATCH	Reading	Specify the patterns to search for in the database.
MERGE	Reading/Writing	Ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.
ON CREATE	Reading/Writing	Used in conjunction with MERGE , specifying the actions to take if the pattern needs to be created.
ON MATCH	Reading/Writing	Used in conjunction with MERGE , specifying the actions to take if the pattern already exists.
OPTIONAL MATCH	Reading	Specify the patterns to search for in the database while using nulls for missing parts of the pattern.
ORDER BY	Reading sub-clause	A sub-clause following RETURN or WITH , specifying that the output should be sorted in particular way.
REMOVE	Writing	Remove properties and labels from nodes and relationships.
RETURN	Projecting	Defines what to include in the query result set.
SET	Writing	Update labels on nodes and properties on nodes and relationships.
SKIP	Reading/Writing	A sub-clause defining from which row to start including the rows in the output.
START	Reading	Find starting points through legacy indexes.
UNION	Set operations	Combines the result of multiple queries. Duplicates are removed.
UNION ALL	Set operations	Combines the result of multiple queries. Duplicates are retained.
UNWIND	Projecting	Expands a list into a sequence of rows.

Clause	Type	Description
USING INDEX variable:Label(property)	Hint	Index hints are used to specify which index, if any, the planner should use as a starting point.
USING JOIN ON variable	Hint	Join hints are used to enforce a join operation at specified points.
USING PERIODIC COMMIT	Hint	This query hint may be used to prevent an out-of-memory error from occurring when importing large amounts of data using <code>LOAD CSV</code> .
USING SCAN variable:Label	Hint	Scan hints are used to force the planner to do a label scan (followed by a filtering operation) instead of using an index.
WITH	Projecting	Allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.
WHERE	Reading sub-clause	A sub-clause used to add constraints to the patterns in a <code>MATCH</code> or <code>OPTIONAL MATCH</code> clause, or to filter the results of a <code>WITH</code> clause.

12.2. Operators

Operator	Type	Description
%	Mathematical	Modulo division
*	Mathematical	Multiplication
+	Mathematical	Addition
+	String	Concatenation
+	List	Concatenation
-	Mathematical	Subtraction or unary minus
.	General	Property access
/	Mathematical	Division
<	Comparison	Less than
<=	Comparison	Less than or equal to
<>	Comparison	Inequality
=	Comparison	Equality
=~	String	Regular expression match
>	Comparison	Greater than
>=	Comparison	Greater than or equal to
AND	Boolean	Conjunction
CONTAINS	String comparison	Case-sensitive inclusion search
DISTINCT	General	Duplicate removal
ENDS WITH	String comparison	Case-sensitive suffix search
IN	List	List element existence check
IS NOT NULL	Comparison	Non- <code>null</code> check
IS NULL	Comparison	<code>null</code> check

Operator	Type	Description
NOT	Boolean	Negation
OR	Boolean	Disjunction
STARTS WITH	String comparison	Case-sensitive prefix search
XOR	Boolean	Exclusive disjunction
<code>[]</code>	General	Subscript (dynamic property access)
<code>[]</code>	List	Subscript (accessing element(s) in a list)
<code>^</code>	Mathematical	Exponentiation

12.3. Functions

Function	Type	Description
<code>abs()</code>	Numeric	Returns the absolute value of a number.
<code>acos()</code>	Trigonometric	Returns the arccosine of the expression.
<code>all()</code>	Predicate	Tests whether a predicate holds for all elements in the list.
<code>any()</code>	Predicate	Tests whether a predicate holds for at least one element in the list.
<code>asin()</code>	Trigonometric	Returns the arcsine of the expression.
<code>atan()</code>	Trigonometric	Returns the arctangent of the expression.
<code>atan2()</code>	Trigonometric	Returns the arctangent2 of a set of coordinates.
<code>avg()</code>	Aggregating	Returns the average of a numeric column.
<code>ceil()</code>	Numeric	Returns the smallest integer greater than or equal to the argument.
<code>coalesce()</code>	Scalar	Returns the first non- <code>null</code> value in the list of expressions passed to it.
<code>collect()</code>	Aggregating	Returns a list containing all collected values.
<code>cos()</code>	Trigonometric	Returns the cosine of the expression.
<code>cot()</code>	Trigonometric	Returns the cotangent of the expression.
<code>count()</code>	Aggregating	Returns the number of rows.
<code>degrees()</code>	Trigonometric	Converts radians to degrees.
<code>distance()</code>	Spatial	Returns a floating point number representing the geodesic distance between any two given points.
<code>e()</code>	Logarithmic	Returns the base of the natural logarithm, <code>e</code> .
<code>endNode()</code>	Scalar	Returns the last node of a relationship.
<code>exists()</code>	Predicate	Returns true if a match for the pattern exists in the graph, or the property exists in the node, relationship or map.

Function	Type	Description
exp()	Logarithmic	Returns e^n , where e is the base of the natural logarithm, and n is the value of the argument expression.
extract()	List	Returns a single property, or the value of a function from a list of nodes or relationships.
filter()	List	Returns all the elements in a list complying with a predicate.
floor()	Numeric	Returns the greatest integer less than or equal to the expression.
haversin()	Trigonometric	Returns half the versine of the expression.
head()	Scalar	Returns the first element in a list.
id()	Scalar	Returns the id of the relationship or node.
keys()	List	Returns a list of string representations for the property names of a node, relationship, or map.
labels()	List	Returns a list of string representations for the labels attached to a node.
last()	Scalar	Returns the last element in a list.
left()	String	Returns a string containing the left n characters of the original string.
length()	Scalar	Returns the length of a path.
log()	Logarithmic	Returns the natural logarithm of the expression.
log10()	Logarithmic	Returns the common logarithm (base 10) of the expression.
lTrim()	String	Returns the original string with whitespace removed from the left side.
max()	Aggregating	Returns the highest value in a numeric column.
min()	Aggregating	Returns the lowest value in a numeric column.
nodes()	List	Returns all nodes in a path.
none()	Predicate	Returns true if the predicate holds for no element in the list.
percentileCont()	Aggregating	Returns the percentile of a given value over a group using linear interpolation.
percentileDisc()	Aggregating	Returns the nearest value to a given percentile over a group using a rounding method.
pi()	Trigonometric	Returns the mathematical constant π .
point() - cartesian 2D	Spatial	Returns a point object, given two coordinate values in the cartesian coordinate system.
point() - WGS 84	Spatial	Returns a point object, given two coordinate values in the WGS 84 coordinate system.

Function	Type	Description
<code>properties()</code>	Scalar	If the argument is a node or a relationship, the returned map is a map of its properties.
<code>radians()</code>	Trigonometric	Converts degrees to radians.
<code>rand()</code>	Numeric	Returns a random number in the range from 0 (inclusive) to 1 (exclusive), <code>[0, 1)</code> .
<code>range()</code>	List	Returns numerical values in a range.
<code>reduce()</code>	List	Runs an expression against individual elements of a list, storing the result of the expression in an accumulator.
<code>relationships()/rels()</code>	List	Returns all relationships in a path.
<code>replace()</code>	String	Returns a string with the search string replaced by the replace string, replacing all occurrences.
<code>reverse()</code>	String	Returns the original string reversed.
<code>right()</code>	String	Returns a string containing the right n characters of the original string.
<code>round()</code>	Numeric	Returns the numerical expression, rounded to the nearest integer.
<code>rTrim()</code>	String	Returns the original string with whitespace removed from the right side.
<code>sign()</code>	Numeric	Returns the signum of a number — zero if the expression is zero, <code>-1</code> for any negative number, and <code>1</code> for any positive number.
<code>sin()</code>	Trigonometric	Returns the sine of the expression.
<code>single()</code>	Predicate	Returns true if the predicate holds for exactly one of the elements in the list.
<code>size()</code>	Scalar	Returns the number of items in a list.
<code>Size of pattern expression</code>	Scalar	Returns the number of sub-graphs matching the pattern expression.
<code>Size of string</code>	Scalar	Returns the size of a string.
<code>split()</code>	String	Returns the sequence of strings which are delimited by split patterns.
<code>sqrt()</code>	Logarithmic	Returns the square root of a number.
<code>startNode()</code>	Scalar	Returns the first node of a relationship.
<code>stDev()</code>	Aggregating	Returns the standard deviation for a given value over a group for a sample of a population.
<code>stDevP()</code>	Aggregating	Returns the standard deviation for a given value over a group for an entire population.
<code>substring()</code>	String	Returns a substring of the original, with a 0-based index start and length.
<code>sum()</code>	Aggregating	Returns the sum of a numeric column.
<code>tail()</code>	List	Returns all but the first element in a list.
<code>tan()</code>	Trigonometric	Returns the tangent of the expression.

Function	Type	Description
<code>timestamp()</code>	Scalar	Returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.
<code>toBoolean()</code>	Scalar	Converts the argument to a boolean and returns the result.
<code>toFloat()</code>	Scalar	Converts the argument to a float and returns the result.
<code>toInteger()</code>	Scalar	Converts the argument to an integer and returns the result.
<code>toLowerCase()</code>	String	Returns the original string in lowercase.
<code>toString()</code>	String	Converts the argument to a string.
<code>toUpperCase()</code>	String	Returns the original string in uppercase.
<code>trim()</code>	String	Returns the original string with whitespace removed from both sides.
<code>type()</code>	Scalar	Returns a string representation of the relationship type.

12.4. Expressions

Name	Description
<code>CASE Expression</code>	A generic conditional expression, similar to if/else statements available in other languages.

Drivers

This chapter contains the complete documentation of the official Neo4j drivers.

Driver version: 1.4.

Neo4j drivers provide application access to a Neo4j database. These official drivers use the Bolt protocol. They have been designed to strike a balance between an idiomatic API for each language, and a uniform surface across all supported languages.

The chapter describes the following:

- [Get started](#) — An overview of the official Neo4j drivers and how to connect to a Neo4j database.
- [Client applications](#) — How to manage database connections within an application.
- [Sessions and transactions](#) — How to create units of work and provide a logical context for that work.
- [Working with Cypher values](#) — The types and values used by Cypher and how they map to native language types.

Chapter 13. Get started

This section gives an overview of the official Neo4j drivers and how to connect to a Neo4j database with a "Hello World" example.

13.1. About the official drivers

Both official and community database drivers are available to provide access to Neo4j for applications. The official drivers have been built for Neo4j 3.0 and above, and use the [Bolt protocol](#) for communication. Official drivers exist for:

- C# — works with any .NET language
- Java — works with any JVM language
- JavaScript
- Python

The Bolt protocol has been designed by Neo Technology, specifically for use with Neo4j, and is fully aligned with the Cypher type system.

The driver API is intended to be topologically agnostic. By this, we mean that the underlying database topology — single instance, causal cluster, etc. — can be altered without requiring a corresponding alteration to application code. In the general case, only the [connection URI](#) needs to be modified when changes are made to the topology.

Table 198. Driver compatibility per Neo4j version

	Driver version					Neo4j version
1.0	1.1	1.2	1.3	1.4	3.0	
Upgrade recommended (*)	Upgrade recommended (*)	Upgrade recommended (*)	Upgrade recommended (*)	Recommended	3.1	
Upgrade recommended (*)	Upgrade recommended (*)	Upgrade recommended (*)	Upgrade recommended (*)	Recommended	3.2	

(*) Regardless of which version of Neo4j you are using, it is strongly recommended to upgrade to the latest driver version. All 1.x drivers are compatible with all 3.x servers but earlier versions of the driver may not provide all features.



The official drivers do not support HTTP communication. If you need an HTTP driver, there are a number of community drivers to choose from. See also [HTTP API](#).

13.2. Driver installation

Example 1. Acquire the driver

To install the latest version of the driver:

```
pip install neo4j-driver
```

To install a certain version of the driver:

```
pip install neo4j-driver==1.4.0
```

13.3. A "Hello World" example

The example below shows the minimal configuration necessary to interact with Neo4j through a driver.

Example 2. Hello World

```
class HelloWorldExample(object):

    def __init__(self, uri, user, password):
        self._driver = GraphDatabase.driver(uri, auth=(user, password))

    def close(self):
        self._driver.close()

    def print_greeting(self, message):
        with self._driver.session() as session:
            greeting = session.write_transaction(self._create_and_return_greeting, message)
            print(greeting)

    @staticmethod
    def _create_and_return_greeting(tx, message):
        result = tx.run("CREATE (a:Greeting)"
                       "SET a.message = $message"
                       "RETURN a.message + ' from node ' + id(a)", message=message)
        return result.single()[0]
```

13.4. Driver API docs

For a comprehensive listing of all driver functionality, refer to the API documentation for the specific language driver.

Example 3. API docs

```
https://neo4j.com/docs/api/python-driver/1.4/
```

Chapter 14. Client applications

This section describes how to manage database connections within an application.

14.1. The driver object

A Neo4j client application will require a driver object instance in order to provide access to the database. This object instance should be made available to all parts of the application that need to interact with Neo4j. In languages where [thread safety](#) is an issue, the driver can be considered thread-safe.

A note on lifecycle



Applications will typically construct a driver instance on startup and destroy it on exit. Destroying a driver instance will immediately shut down any connections previously opened via that driver; for drivers that contain a connection pool, the entire pool will be shut down.

To construct a driver instance, a [connection URI](#) and [authentication information](#) must be supplied. Additional configuration details can be supplied if required. All of these details are immutable for the lifetime of the driver. Therefore, if multiple configurations are required (such as when working with multiple database users) then multiple driver objects must be used.

An example of driver construction and destruction can be seen below:

Example 4. The driver lifecycle

```
class DriverLifecycleExample:  
    def __init__(self, uri, user, password):  
        self._driver = GraphDatabase.driver(uri, auth=(user, password))  
  
    def close(self):  
        self._driver.close()
```

14.2. Connection URIs

A connection URI identifies a graph database and how to connect to it. The official Neo4j drivers currently support the following URI schemes and driver object types:

Table 199. Available URI schemes

URI Scheme	Driver object type
bolt	Direct driver
bolt+routing	Routing driver

14.2.1. Direct drivers (bolt)

A Direct driver is created via a `bolt` URI, for example: `bolt://localhost:7687`. This kind of driver is used to maintain connections to a single database server and is typically used when working with a single Neo4j instance or when targeting a specific member of a cluster. Note that a Routing driver is preferable when working with a causal cluster.

14.2.2. Routing drivers (bolt+routing)

A Routing driver is created via a `bolt+routing` URI, for example:

`bolt+routing://graph.example.com:7687`. The address in the URI must be that of a core server. This kind of driver uses the [Bolt Routing Protocol](#) and works in tandem with the cluster to route transactions to available cluster members.

14.2.3. Routing drivers with routing context

Routing drivers with routing context are an available option when using drivers of version 1.3 or above together with a Neo4j cluster of version 3.2 or above. In such a setup, a routing driver can include a preferred routing context via the query part of the `bolt+routing` URI. Below is an example using a routing context of `{region: europe, country: sw}`:

```
bolt+routing://graph.example.com:7687/?region=europe&country=sw
```

The driver communicates the routing context to the cluster. It then obtains refined routing information back from the cluster, based on information suggested in the routing context. In the example above, the driver communicates that it prefers to route queries to the servers located in the country Sweden in the region Europe.

14.3. Authentication

Authentication details are provided as an auth token which contains the user names, passwords or other credentials required to access the database. Neo4j supports multiple authentication standards but uses basic authentication by default.

14.3.1. Basic authentication

The basic authentication scheme is backed by a password file stored within the server and requires applications to provide a user name and password. For this, use the basic auth helper:

Example 5. Basic authentication

```
def __init__(self, uri, user, password):
    self._driver = GraphDatabase.driver(uri, auth=(user, password))
```



The basic authentication scheme can also be used to authenticate against an LDAP server.

14.3.2. Kerberos authentication

The Kerberos authentication scheme provides a simple way to create a Kerberos authentication token with a base64 encoded server authentication ticket. The best way to create a Kerberos authentication token is shown below:

Example 6. Kerberos authentication

```
def __init__(self, uri, ticket):
    self._driver = GraphDatabase.driver(uri, auth=kerberos_auth(ticket))
```



The Kerberos authentication token can only be understood by the server if the server has the [Kerberos Add-on](#) (<https://neo4j.com/docs/add-on/kerberos/1.0>) installed.

14.3.3. Custom authentication

For advanced deployments, where a custom security provider has been built, the custom authentication helper can be used.

Example 7. Custom authentication

```
def __init__(self, uri, principal, credentials, realm, scheme, **parameters):
    self._driver = GraphDatabase.driver(uri, auth=custom_auth(principal, credentials, realm,
    scheme, **parameters))
```

14.4. Configuration

14.4.1. Encryption

TLS encryption is enabled for all connections by default. This can be disabled through configuration as follows:

Example 8. Unencrypted

```
def __init__(self, uri, user, password):
    self._driver = GraphDatabase.driver(uri, auth=(user, password), encrypted=False)
```

The server can be modified to require encryption for all connections. Please see the [Operations Manual](#) [Configure Neo4j connectors](#) for more information.

An attempt to connect to a server using an encryption setting not allowed by that server will result in a [Service unavailable](#) status.

14.4.2. Trust

During a TLS handshake, the server provides a certificate to the client application. The application can choose to accept or reject this certificate based on one of the following trust strategies:

Table 200. Trust strategies

Trust strategy	Description
TRUST_ALL_CERTIFICATES (default)	Accept any certificate provided by the server
TRUST_CUSTOM_CA_SIGNED_CERTIFICATES	Accept any certificate that can be verified against a custom CA
TRUST_SYSTEM_CA_SIGNED_CERTIFICATES	Accept any certificate that can be verified against the system store

Example 9. Trust

```
def __init__(self, uri, user, password):
    self._driver = GraphDatabase.driver(uri, auth=(user, password), trust=
    TRUST_ALL_CERTIFICATES)
```

14.4.3. Connection timeout

To configure the maximum time allowed to establish a connection, pass a duration value to the driver configuration. For example:

Example 10. Connection timeout

```
def __init__(self, uri, user, password):
    self._driver = GraphDatabase.driver(uri, auth=(user, password), connection_timeout=15)
```

14.4.4. Max retry time

To configure retry behavior, supply a value for the maximum time in which to keep attempting retries of transaction functions. For example:

Example 11. Max retry time

```
def __init__(self, uri, user, password):
    self._driver = GraphDatabase.driver(uri, auth=(user, password), max_retry_time=15)
```

Note that the time specified here does not take into account the running time of the unit of work itself, merely a limit after which retries will no longer be attempted.

14.5. Service unavailable

A Service unavailable status will be signalled when the driver is no longer able to establish communication with the server, even after retries. Encountering this condition usually indicates a fundamental networking or database problem. Applications should be designed to cater for this eventuality.

Example 12. Service unavailable

```
def add_item(self):
    with self._driver.session() as session:
        try:
            session.write_transaction(lambda tx: tx.run("CREATE (a:Item)"))
            return True
        except ServiceUnavailable:
            return False
```

Chapter 15. Sessions and transactions

This section describes how to create units of work and provide a logical context for that work.

15.1. Sessions

A session is a container for a sequence of transactions. Sessions borrow connections from a pool as required and so should be considered lightweight and disposable. In languages where [thread safety](#) is an issue, a session should *not* be considered thread-safe.

In languages that support them, sessions are usually scoped within a context block. This ensures that they are properly closed and that any underlying connections are released and not leaked.

Example 13. Session

```
def add_person(self, name):
    with self._driver.session() as session:
        session.run("CREATE (a:Person {name: $name})", name=name)
```

15.2. Transactions

Transactions are atomic units of work consisting of one or more Cypher statement executions. A transaction is executed within a session.

To execute a Cypher statement, two pieces of information are required: the statement template and a keyed set of parameters. The template is a string containing placeholders that are substituted with parameter values at runtime. While it is possible to run non-parameterized Cypher, good programming practice is to use parameters in Cypher statements. This allows for caching of statements within the Cypher engine, which is beneficial for performance. Parameter values should adhere to [Values](#).

The Neo4j driver API provides for three forms of transaction:

- Auto-commit transactions
- Transaction functions
- Explicit transactions

Of these, only [transaction function](#) can be automatically replayed on failure.

15.2.1. Auto-commit Transactions

An auto-commit transaction is a simple but limited form of transaction. Such a transaction consists of only one Cypher statement, cannot be automatically replayed on failure, and cannot take part in a [causal chain](#).

An auto-commit transaction is invoked using the `session.run` method:

Example 14. Auto-commit transaction

```
def add_person(self, name):
    with self._driver.session() as session:
        session.run("CREATE (a:Person {name: $name})", name=name)
```

Auto-commit transactions are sent to the network and acknowledged immediately. This means that multiple transactions cannot share network packets, thereby exhibiting a lesser network efficiency than other forms of transaction.

Auto-commit transactions are intended to be used for simple use cases such as when learning Cypher or writing one-off scripts. It is not recommended to use auto-commit transactions in production environments or when performance or resilience are a primary concern.

However, Auto-commit transactions are the only way to execute `USING PERIODIC COMMIT` Cypher statements.

15.2.2. Transaction functions

Transaction functions are the recommended form for containing transactional units of work. This form requires minimal boilerplate code and allows for a clear separation of database queries and application logic.

Example 15. Transaction function

```
def add_person(self, name):
    with self._driver.session() as session:
        session.write_transaction(self.create_person_node, name)

@staticmethod
def create_person_node(tx, name):
    tx.run("CREATE (a:Person {name: $name})", name=name)
```

Transaction functions are also able to handle connection problems and transient errors using an automatic retry mechanism. This retry capability can be [configured](#) on Driver construction.

Any query results obtained within a transaction function should be consumed within that function. Transaction functions can return values but these should be derived values rather than raw results.

15.2.3. Explicit transactions

Explicit transactions are the longhand form of transaction functions, providing access to explicit `BEGIN`, `COMMIT` and `ROLLBACK` operations. While this form is useful for a handful of use cases, it is recommended to use transaction functions wherever possible.

15.2.4. Cypher errors

When executing Cypher, it is possible for an exception to be thrown by the Cypher engine. Each such exception is associated with a [status code](#) that describes the nature of the error and a message that provides more detail.

The error classifications are listed in the table below.

Table 201. Error classifications

Classification	Description
ClientError	The client application has caused an error. The application should amend and retry the operation.
DatabaseError	The server has caused an error. Retrying the operation will generally be unsuccessful.
TransientError	A temporary error has occurred. The application should retry the operation.

15.3. Causal chaining

When working with a causal cluster, the transactions in a session will automatically be causally chained. This means that for any two transactions, it is guaranteed that the second transaction will begin only after the first has been successfully committed. This is true even if the transactions are physically carried out on different cluster members.

Causal chaining is carried out by passing [bookmarks](#) between transactions in a session. Each bookmark records a point in transactional history and is used to instruct the cluster to carry out work in a particular sequence. Bookmarks are returned from the server to the client on the successful commit of a transaction and may be supplied as parameters to new transactions. Passing bookmarks into a transaction causes the executing server to block until it has caught up with the latest of these bookmarks.

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

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



If you try to extract a bookmark from a database which is not running in Causal Cluster mode, you will receive a `null` result.

15.4. Access modes

Transactions can be executed in either `read` or `write` mode. In a causal cluster, each transaction will be routed to an appropriate server based on the mode. When using a single instance, all transactions will be passed to that one server. Routing Cypher by identifying reads and writes can improve the utilization of available cluster resources: as read servers are typically more plentiful than write servers, it is beneficial to direct as much as possible of read transactions to read servers. Doing so helps keeping write servers available for write transactions.

Note that the driver does not parse Cypher and cannot determine whether a transaction is intended to carry out read or write operations. As a result of this, a `write` transaction tagged for `read` will be sent to a read server, but will fail on execution.

Example 16. Read-write transaction

```
def add_person(self, name):
    with self._driver.session() as session:
        session.write_transaction(self.create_person_node, name)
        return session.read_transaction(self.match_person_node, name)

@staticmethod
def create_person_node(tx, name):
    tx.run("CREATE (a:Person {name: $name})", name=name)
    return None

@staticmethod
def match_person_node(tx, name):
    result = tx.run("MATCH (a:Person {name: $name}) RETURN count(a)", name=name)
    return result.single()[0]
```

Chapter 16. Working with Cypher values

This section describes the types and values used by Cypher and how they map to native language types.

16.1. The Cypher type system

Drivers translate between application language types and the Cypher type system. To pass parameters and process results, it is important to know the basics of the Cypher type system and to understand how the Cypher types are mapped in the driver.

Cypher types are split into two groups: basic types and graph structure types. While query results can contain any of these, only basic types can be used as parameters.

16.1.1. Basic types

The basic types are:

- Boolean
- Integer
- Float
- String
- List
- Map



For information on how to work with `null` in Cypher, please refer to [Working with null](#).

16.1.2. Graph structure types

In addition to the basic types which can be used both for parameters and results, the following Cypher types are valid for results only:

- Node
- Relationship
- Path

16.1.3. Type mappings

The Neo4j driver maps Cypher types to native language types as depicted in the table below.

Example 17. Map Neo4j types to native language types

Neo4j	Python
Null	None
Boolean	bool
Integer	int/long (†)
Float	float
Bytes	bytes
String	str/unicode (†)
List	list
Map	dict
Node	Node
Relationship	Relationship
Path	Path

(†) Python versions

In Python 2, a Cypher `Integer` is mapped to `int` and a Cypher `String` to `str`.

In Python 3, a Cypher `Integer` is mapped to `long` and a Cypher `String` to `unicode`.

16.2. Statement results

A statement result is comprised of a stream of records. The result is typically handled by the receiving application as it arrives, although it is also possible to retain all or part of a result for future consumption.

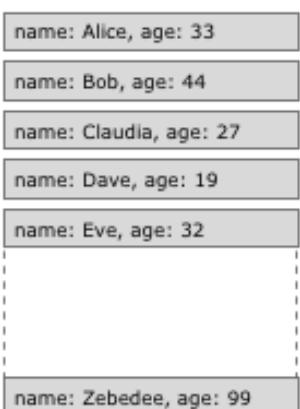


Figure 28. Result stream

16.2.1. Records

A record provides an immutable view of part of a result. It is an ordered map of keys and values. As values within a record are both keyed and ordered, that value can be accessed either by position (0-based integer) or by key (string).

16.2.2. The buffer

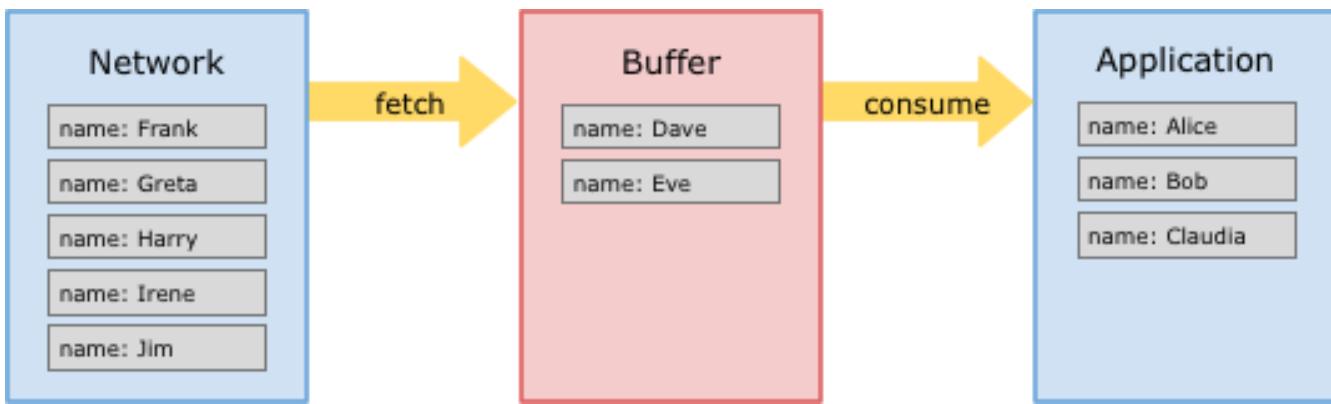


Figure 29. Result buffer

Most drivers contain a result buffer. This provides a staging point for results, and divides result handling into *fetching* (moving from the network to the buffer) and *consuming* (moving from the buffer to the application).

If results are consumed in the same order as they are produced, records merely pass through the buffer; if they are consumed out of order, the buffer will be utilized to retain records until they are consumed by the application. For large results, this may require a significant amount of memory and impact performance. For this reason, it is recommended to consume results in order wherever possible.

16.2.3. Consuming the stream

Query results will often be consumed as a stream. Drivers provide a language-idiomatic way to iterate through a result stream.

Example 18. Consuming the stream

```
def get_people(self):
    with self._driver.session() as session:
        return session.read_transaction(self.match_person_nodes)

@staticmethod
def match_person_nodes(tx):
    result = tx.run("MATCH (a:Person) RETURN a.name ORDER BY a.name")
    return [record["a.name"] for record in result]
```

16.2.4. Retaining results

The result record stream is available until another statement is run in the session, or until the current transaction is closed. To hold on to the results beyond this scope, the results need to be explicitly retained. For retaining results, each driver offers methods that collect the result stream and translate it into standard data structures for that language. Retained results can be processed while the session is free to take on the next workload. The saved results can also be used directly to run a new statement.

Example 19. Retain results for further processing

```
def add_employees(self, company_name):
    with self._driver.session() as session:
        employees = []
        persons = session.read_transaction(self.match_person_nodes)

        for person in persons:
            employees += session.write_transaction(self.add_employee_to_company, person,
company_name)

    return employees

@staticmethod
def add_employee_to_company(tx, person, company_name):
    tx.run("MATCH (emp:Person {name: $person_name}) "
          "MERGE (com:Company {name: $company_name}) "
          "MERGE (emp)-[:WORKS_FOR]->(com)",
          person_name=person["name"], company_name=company_name)

@staticmethod
def match_person_nodes(tx):
    return list(tx.run("MATCH (a:Person) RETURN a.name AS name"))
```

16.3. Statement result summaries

Supplementary information such as query statistics, timings and server information can be obtained from the statement result summary. If this detail is accessed before the entire result has been consumed, the remainder of the result will be buffered.

See also the [language-specific driver API documentation](#).

HTTP API

This chapter covers the HTTP API for operating and querying Neo4j.

Chapter 17. Transactional Cypher HTTP 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 [Operations Manual → dbms.rest.transaction.idle_timeout](#) to the number of seconds before timeout. The default timeout is 60 seconds.

- 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 [PERIODIC COMMIT query hint](#)) may only be executed when creating a new transaction and immediately committing it with a single HTTP request (see [Begin and commit a transaction in one request](#) for how to do that).
- When a request fails the transaction will be rolled back. By checking the result for the presence/absence of the `transaction` key you can figure out if the transaction is still open.



17.1. Streaming

Responses from the HTTP 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.



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 [Parameters](#) for more information.

17.2. 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" : [ 6 ],  
      "meta" : [ null ]  
    } ]  
  },  
  "errors" : [ ]  
}
```

17.3. 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" : [ 2 ],
      "meta" : [ null ]
    } ],
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ {
        "name" : "My Node"
      } ],
      "meta" : [ {
        "id" : 3,
        "type" : "node",
        "deleted" : false
      } ]
    } ],
    "errors" : [ ]
  }
}
```

17.4. 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/10`

```
{
  "commit" : "http://localhost:7474/db/data/transaction/10/commit",
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ {
        "name" : "My Node"
      } ],
      "meta" : [ {
        "id" : 10,
        "type" : "node",
        "deleted" : false
      } ]
    } ],
    "transaction" : {
      "expires" : "Fri, 14 Jul 2017 18:04:09 +0000"
    },
    "errors" : [ ]
  }
}
```

17.5. 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/12
- **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/12/commit",
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ {
        "name" : "My Node"
      } ],
      "meta" : [ {
        "id" : 11,
        "type" : "node",
        "deleted" : false
      } ]
    } ],
    "transaction" : {
      "expires" : "Fri, 14 Jul 2017 18:04:10 +0000"
    },
    "errors" : [ ]
  }
}
```

17.6. 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" : "Fri, 14 Jul 2017 18:04:07 +0000"  
  },  
  "errors" : [ ]  
}
```

17.7. 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" : [ 5 ],  
      "meta" : [ null ]  
    } ]  
  },  
  "errors" : [ ]  
}
```

17.8. 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" : [ ]  
}
```

17.9. 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" : [ 4 ],
      "meta" : [ null ]
    } ],
    "stats" : {
      "contains_updates" : true,
      "nodes_created" : 1,
      "nodes_deleted" : 0,
      "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" : [ ]
}
```

17.10. 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 visualize 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
} ] ],
"meta" : [ {
  "id" : 7,
  "type" : "node",
  "deleted" : false
}, [ {
  "id" : 7,
  "type" : "node",
  "deleted" : false
}, {
  "id" : 0,
  "type" : "relationship",
  "deleted" : false
}, {
  "id" : 8,
  "type" : "node",
  "deleted" : false
} ], [ {
  "id" : 7,
  "type" : "node",
  "deleted" : false
}, {
  "id" : 1,
  "type" : "relationship",
  "deleted" : false
}, {
  "id" : 9,
  "type" : "node",
  "deleted" : false
} ] ],
"graph" : {
  "nodes" : [ {
    "id" : "7",
    "labels" : [ "Bike" ],
    "properties" : {
      "weight" : 10
    }
  },
  {
    "id" : "8",
    "labels" : [ "Wheel" ],
    "properties" : {
      "spokes" : 3
    }
  },
  {
    "id" : "9",
    "labels" : [ "Wheel" ],
    "properties" : {
      "spokes" : 32
    }
  } ],
  "relationships" : [ {
    "id" : "0",
    "type" : "HAS",
    "startNode" : "7",
    "endNode" : "8",
    "properties" : {
      "position" : 1
    }
  },
  {
    "id" : "1",
    "type" : "HAS",
    "startNode" : "7",
    "endNode" : "9",
    "properties" : {
      "position" : 2
    }
  } ]
},
"errors" : [ ]
}

```

17.11. 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 [Neo4j Status Codes](#).

Example request

- **POST** `http://localhost:7474/db/data/transaction/11/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.SyntaxError",  
    "message" : "Invalid input 'T': expected <init> (line 1, column 1 (offset: 0))\n\\\"This is not a valid  
Cypher Statement.\\\"\n ^"  
  } ]  
}
```

17.12. Handling errors in an open transaction

Whenever there is an error in a request the server will rollback the transaction. By inspecting the response for the presence/absence of the `transaction` key you can tell if the transaction is still open

Example request

- **POST** `http://localhost:7474/db/data/transaction/9`
- **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

```
{  
  "commit" : "http://localhost:7474/db/data/transaction/9/commit",  
  "results" : [ ],  
  "errors" : [ {  
    "code" : "Neo.ClientError.Statement.SyntaxError",  
    "message" : "Invalid input 'T': expected <init> (line 1, column 1 (offset: 0))\n\\\"This is not a valid  
Cypher Statement.\\\"\n ^"  
  } ]  
}
```

Chapter 18. Authentication and authorization

This section describes authentication and authorization using the Neo4j HTTP API.

The HTTP API supports authentication and authorization so that requests to the HTTP API must be authorized using the username and password of a valid user. Authentication and authorization are enabled by default. Refer to [Operations Manual □ Enabling authentication and authorization](#) for a description on how to enable and disable authentication and authorization.

When Neo4j is first installed you can authenticate with the default user `neo4j` and the default password `neo4j`. The default password must be changed before access to resources will be permitted. This is done either using Neo4j Browser or via direct HTTP calls (see [User status and password changing](#)).

18.1. Authenticating

18.1.1. 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: Basic realm="Neo4j"`

```
{  
  "errors" : [ {  
    "code" : "Neo.ClientError.Security.Unauthorized",  
    "message" : "No authentication header supplied."  
  } ]  
}
```

18.1.2. 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

```
{
  "password_change_required" : false,
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "username" : "neo4j"
}
```

18.1.3. 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:** Basic realm="Neo4j"

```
{
  "errors" : [ {
    "code" : "Neo.ClientError.Security.Unauthorized",
    "message" : "Invalid username or password."
  } ]
}
```

18.1.4. 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 [User status and password changing](#) 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" : [ {
    "code" : "Neo.ClientError.Security.Forbidden",
    "message" : "User is required to change their password."
  } ]
}
```

18.2. User status and password changing

18.2.1. 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

```
{  
  "password_change_required" : false,  
  "password_change" : "http://localhost:7474/user/neo4j/password",  
  "username" : "neo4j"  
}
```

18.2.2. 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

```
{  
  "password_change_required" : true,  
  "password_change" : "http://localhost:7474/user/neo4j/password",  
  "username" : "neo4j"  
}
```

18.2.3. 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

18.3. Access when authentication and authorization are disabled

When authentication and authorization have been disabled, HTTP API requests can be sent without an [Authorization](#) header.

18.4. Copying security configuration from one instance to another

The username and password combination is local to each Neo4j instance. In many cases you want to start a Neo4j instance with preconfigured authentication and authorization. For instructions on how to do this, refer to [Operation Manual □ Propagate users and roles](#).

Extending Neo4j

This chapter introduces three methods for extending Neo4j: procedures, functions, and plugins.

- [Procedures](#)
- [User-defined functions](#)
- [Authentication and authorization plugins](#)

Chapter 19. Procedures

User-defined procedures are written in Java, deployed into the database, and called from Cypher.

A *procedure* is a mechanism that allows Neo4j to be extended by writing custom code which can be invoked directly from Cypher. Procedures can take arguments, perform operations on the database, and return results.

Procedures are written in Java and compiled into *jar* files. They can be deployed to the database by dropping a *jar* file into the `$NEO4J_HOME/plugins` directory on each standalone or clustered server. The database must be re-started on each server to pick up new procedures.

Procedures are the preferred means for extending Neo4j. Examples of use cases for procedures are:

1. To provide access to functionality that is not available in Cypher, such as manual indexes and schema introspection.
2. To provide access to third party systems.
3. To perform graph-global operations, such as counting connected components or finding dense nodes.
4. To express a procedural operation that is difficult to express declaratively with Cypher.

19.1. Calling procedures

To call a stored procedure, use a Cypher `CALL` clause. The procedure name must be fully qualified, so a procedure named `findDenseNodes` defined in the package `org.neo4j.examples` could be called using:

```
CALL org.neo4j.examples.findDenseNodes(1000)
```

A `CALL` may be the only clause within a Cypher statement or may be combined with other clauses. Arguments can be supplied directly within the query or taken from the associated parameter set. For full details, see the Cypher documentation on [the CALL clause](#).

19.2. Built-in procedures

Neo4j comes bundled with a number of built-in procedures. These can be used to:

- Inspect schema.
- Inspect meta data.
- Explore procedures and components.
- Monitor management data.
- Set user password.

A subset of these are described in [Operations Manual □ Built-in procedures](#). Running `CALL dbms.procedures()` will display the full list of all the procedures.

19.3. User-defined procedures

This section covers how to write, test and deploy a procedure for Neo4j.

Custom procedures are written in the Java programming language. Procedures are deployed via a *jar* file that contains the code itself along with any dependencies (excluding Neo4j). These files should be placed into the *plugin* directory of each standalone database or cluster member and will become available following the next database restart.

The example that follows shows the steps to create and deploy a new procedure.



The example discussed below is available as [a repository on GitHub](https://github.com/neo4j-examples/neo4j-procedure-template) (<https://github.com/neo4j-examples/neo4j-procedure-template>). To get started quickly you can fork the repository and work with the code as you follow along in the guide below.

19.3.1. Set up a new project

A project can be set up in any way that allows for compiling a procedure and producing a *jar* file. Below is an example configuration using the [Maven](https://maven.apache.org/) (<https://maven.apache.org/>) build system. For readability, only excerpts from the Maven *pom.xml* file are shown here, the whole file is available from the [Neo4j Procedure Template](https://github.com/neo4j-examples/neo4j-procedure-template) (<https://github.com/neo4j-examples/neo4j-procedure-template>) repository.

Setting up a project with Maven

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.neo4j.example</groupId>
  <artifactId>procedure-template</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <packaging>jar</packaging>
  <name>Neo4j Procedure Template</name>
  <description>A template project for building a Neo4j Procedure</description>

  <properties>
    <neo4j.version>3.2.3</neo4j.version>
  </properties>
```

Next, the build dependencies are defined. The following two sections are included in the *pom.xml* between *<dependencies></dependencies>* tags.

The first dependency section includes the procedure API that procedures use at runtime. The scope is set to [provided](#), because once the procedure is deployed to a Neo4j instance, this dependency is provided by Neo4j. If non-Neo4j dependencies are added to the project, their scope should normally be [compile](#).

```
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j</artifactId>
  <version>${neo4j.version}</version>
  <scope>provided</scope>
</dependency>
```

Next, the dependencies necessary for testing the procedure are added:

- Neo4j Harness, a utility that allows for starting a lightweight Neo4j instance. It is used to start Neo4j with a specific procedure deployed, which greatly simplifies testing.
- The Neo4j Java driver, used to send cypher statements that call the procedure.
- JUnit, a common Java test framework.

```

<dependency>
  <groupId>org.neo4j.test</groupId>
  <artifactId>neo4j-harness</artifactId>
  <version>${neo4j.version}</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.neo4j.driver</groupId>
  <artifactId>neo4j-java-driver</artifactId>
  <version>1.4.0</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>

```

Along with declaring the dependencies used by the procedure it is also necessary to define the steps that Maven will go through to build the project. The goal is first to *compile* the source, then to *package* it in a *jar* that can be deployed to a Neo4j instance.



Procedures require at least Java 8, so the version **1.8** should be defined as the *source* and *target version* in the configuration for the Maven compiler plugin.

The [Maven Shade](https://maven.apache.org/plugins/maven-shade-plugin/) (<https://maven.apache.org/plugins/maven-shade-plugin/>) plugin is used to package the compiled procedure. It also includes all dependencies in the package, unless the dependency scope is set to *test* or *provided*.

Once the procedure has been deployed to the *plugins* directory of each Neo4j instance and the instances have restarted, the procedure is available for use.

```

<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
    <plugin>
      <artifactId>maven-shade-plugin</artifactId>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

19.3.2. Writing integration tests

The test dependencies include *Neo4j Harness* and *JUnit*. These can be used to write integration tests for procedures.

First, we decide what the procedure should do, then we write a test that proves that it does it right. Finally we write a procedure that passes the test.

Below is a template for testing a procedure that accesses Neo4j's full-text indexes from Cypher.

Writing tests for procedures

```
package example;

import org.junit.Rule;
import org.junit.Test;
import org.neo4j.driver.v1.*;
import org.neo4j.graphdb.factory.GraphDatabaseSettings;
import org.neo4j.harness.Neo4jRule;

import static org.hamcrest.core.IsEqual.equalTo;
import static org.junit.Assert.assertThat;
import static org.neo4j.driver.v1.Values.parameters;

public class ManualFullTextIndexTest
{
    // This rule starts a Neo4j instance
    @Rule
    public Neo4jRule neo4j = new Neo4jRule()

        // This is the Procedure we want to test
        .withProcedure( FullTextIndex.class );

    @Test
    public void shouldAllowIndexingAndFindingANode() throws Throwable
    {
        // In a try-block, to make sure we close the driver after the test
        try( Driver driver = GraphDatabase.driver( neo4j.boltURI(), Config.build().withEncryptionLevel(
        Config.EncryptionLevel.NONE ).toConfig() ) )
        {

            // Given I've started Neo4j with the FullTextIndex procedure class
            // which my 'neo4j' rule above does.
            Session session = driver.session();

            // And given I have a node in the database
            long nodeId = session.run( "CREATE (p:User {name:'Brookreson'}) RETURN id(p)" )
                .single()
                .get( 0 ).asLong();

            // When I use the index procedure to index a node
            session.run( "CALL example.index({id}, ['name'])", parameters( "id", nodeId ) );

            // Then I can search for that node with lucene query syntax
            StatementResult result = session.run( "CALL example.search('User', 'name:Brook*')" );
            assertThat( result.single().get( "nodeId" ).asLong(), equalTo( nodeId ) );
        }
    }
}
```

19.3.3. Writing a procedure

With the test in place, we write a procedure procedure that fulfils the expectations of the test. The full example is available in the [Neo4j Procedure Template](https://github.com/neo4j-examples/neo4j-procedure-template) (<https://github.com/neo4j-examples/neo4j-procedure-template>) repository.

Particular things to note:

- All procedures are annotated `@Procedure`. Procedures that write to the database specify `mode = WRITE`.
- The *context* of the procedure, which is the same as each resource that the procedure wants to use, is annotated `@Context`.
- The *input* and *output*.

For more details, see the [API documentation for procedures](https://neo4j.com/docs/java-reference/3.2/javadocs/index.html?org/neo4j/procedure/Procedure.html) (<https://neo4j.com/docs/java-reference/3.2/javadocs/index.html?org/neo4j/procedure/Procedure.html>).



The correct way to signal an error from within a procedure is to throw a `RuntimeException`.

```
package example;

import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Stream;

import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Label;
import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.index.Index;
import org.neo4j.graphdb.index.IndexManager;
import org.neo4j.logging.Log;
import org.neo4j.procedure.Context;
import org.neo4j.procedure.Name;
import org.neo4j.procedure.PerformsWrites;
import org.neo4j.procedure.Procedure;

import static org.neo4j.helpers.collection.MapUtil.stringMap;
import static org.neo4j.procedure.Mode.SCHEMA;
import static org.neo4j.procedure.Mode.WRITE;

/**
 * This is an example showing how you could expose Neo4j's full text indexes as
 * two procedures - one for updating indexes, and one for querying by label and
 * the lucene query language.
 */
public class FullTextIndex
{
    // Only static fields and @Context-annotated fields are allowed in
    // Procedure classes. This static field is the configuration we use
    // to create full-text indexes.
    private static final Map<String, String> FULL_TEXT =
        stringMap( IndexManager.PROVIDER, "lucene", "type", "fulltext" );

    // This field declares that we need a GraphDatabaseService
    // as context when any procedure in this class is invoked
    @Context
    public GraphDatabaseService db;

    // This gives us a log instance that outputs messages to the
    // standard log, `neo4j.log`
    @Context
    public Log log;

    /**
     * This declares the first of two procedures in this class - a
     * procedure that performs queries in a manual index.
     *
     * It returns a Stream of Records, where records are
     * specified per procedure. This particular procedure returns
     * a stream of {@link SearchHit} records.
     *
     * The arguments to this procedure are annotated with the
     * {@link Name} annotation and define the position, name
     * and type of arguments required to invoke this procedure.
     * There is a limited set of types you can use for arguments,
     * these are as follows:
     *
     * <ul>
     *   <li>{@link String}</li>
     *   <li>{@link Long} or {@code long}</li>
     *   <li>{@link Double} or {@code double}</li>
     *   <li>{@link Number}</li>
     *   <li>{@link Boolean} or {@code boolean}</li>
     *   <li>{@link java.util.Map} with key {@link String} and value {@link Object}</li>
     *   <li>{@link java.util.List} of elements of any valid argument type, including {@link
     java.util.List}</li>
     *   <li>{@link Object}, meaning any of the valid argument types</li>
     * </ul>
     *
     * @param label the label name to query by
     * @param query the lucene query, for instance `name:Brook` to
     *              search by property `name` and find any value starting
}
```

```

*           with `Brook`. Please refer to the Lucene Query Parser
*           documentation for full available syntax.
* @return the nodes found by the query
*/
@Procedure( name = "example.search", mode = WRITE )
public Stream<SearchHit> search( @Name("label") String label,
                                  @Name("query") String query )
{
    String index = indexName( label );

    // Avoid creating the index, if it's not there we won't be
    // finding anything anyway!
    if( !db.index().existsForNodes( index ) )
    {
        // Just to show how you'd do logging
        log.debug( "Skipping index query since index does not exist: '%s'", index );
        return Stream.empty();
    }

    // If there is an index, do a lookup and convert the result
    // to our output record.
    return db.index()
        .forNodes( index )
        .query( query )
        .stream()
        .map( SearchHit::new );
}

/**
 * This is the second procedure defined in this class, it is used to update the
 * index with nodes that should be queryable. You can send the same node multiple
 * times, if it already exists in the index the index will be updated to match
 * the current state of the node.
 *
 * This procedure works largely the same as {@link #search(String, String)},
 * with three notable differences. One, it is annotated with `mode = SCHEMA`,
 * which is <i>required</i> if you want to perform updates to the graph in your
 * procedure.
 *
 * Two, it returns {@code void} rather than a stream. This is simply a short-hand
 * for saying our procedure always returns an empty stream of empty records.
 *
 * Three, it uses a default value for the property list, in this way you can call
 * the procedure by simply invoking {@code CALL index(nodeId)}. Default values are
 * are provided as the Cypher string representation of the given type, e.g.
 * {@code {default: true}}, {@code null}, or {@code -1}.
 *
 * @param nodeId the id of the node to index
 * @param propKeys a list of property keys to index, only the ones the node
 *                 actually contains will be added
 */
@Procedure( name = "example.index", mode = SCHEMA )
public void index( @Name("nodeId") long nodeId,
                   @Name("value = "properties", defaultValue = "[]") List<String> propKeys )
{
    Node node = db.getNodeById( nodeId );

    // Load all properties for the node once and in bulk,
    // the resulting set will only contain those properties in 'propKeys'
    // that the node actually contains.
    Set<Map.Entry<String, Object>> properties =
        node.getProperties( propKeys.toArray( new String[0] ) ).entrySet();

    // Index every label (this is just as an example, we could filter which labels to index)
    for ( Label label : node.getLabels() )
    {
        Index<Node> index = db.index().forNodes( indexName( label.name() ), FULL_TEXT );

        // In case the node is indexed before, remove all occurrences of it so
        // we don't get old or duplicated data
        index.remove( node );

        // And then index all the properties
        for ( Map.Entry<String, Object> property : properties )
        {
            index.add( node, property.getKey(), property.getValue() );
        }
    }
}

```

```

/**
 * This is the output record for our search procedure. All procedures
 * that return results return them as a Stream of Records, where the
 * records are defined like this one - customized to fit what the procedure
 * is returning.
 *
 * The fields must be one of the following types:
 *
 * <ul>
 *   <li>{@link String}</li>
 *   <li>{@link Long} or {@code long}</li>
 *   <li>{@link Double} or {@code double}</li>
 *   <li>{@link Number}</li>
 *   <li>{@link Boolean} or {@code boolean}</li>
 *   <li>{@link org.neo4j.graphdb.Node}</li>
 *   <li>{@link org.neo4j.graphdb.Relationship}</li>
 *   <li>{@link org.neo4j.graphdb.Path}</li>
 *   <li>{@link java.util.Map} with key {@link String} and value {@link Object}</li>
 *   <li>{@link java.util.List} of elements of any valid field type, including {@link
java.util.List}</li>
 *   <li>{@link Object}, meaning any of the valid field types</li>
 * </ul>
 */
public static class SearchHit
{
    // This records contain a single field named 'nodeId'
    public long nodeId;

    public SearchHit( Node node )
    {
        this.nodeId = node.getId();
    }
}

private String indexName( String label )
{
    return "label-" + label;
}
}

```

19.3.4. Injectable resources

When writing procedures, some resources can be injected into the procedure from the database. To inject these, use the `@Context` annotation. The classes that can be injected are:

- `Log`
- `TerminationGuard`
- `GraphDatabaseService`

All of the above classes are considered safe and future-proof, and will not compromise the security of the database. There are also several classes that can be injected that are unsupported restricted and can be changed with little or no notice. The database will not load all classes by default but can be toggled with `dbms.security.procedures.unrestricted` to load unsafe procedures.

Chapter 20. User-defined functions

This section covers how to write, test and deploy a user-defined function for Neo4j.

User-defined functions are a simpler form of procedures that are read-only and always return a single value. Although they are not as powerful in capability, they are often easier to use and more efficient than procedures for many common tasks.

20.1. Calling a user-defined function

User-defined functions are called in the same way as any other Cypher function. The function name must be fully qualified, so a function named `join` defined in the package `org.neo4j.examples` could be called using:

```
MATCH (p: Person) WHERE p.age = 36 RETURN org.neo4j.examples.join(collect(p.names))
```

20.2. Writing a user-defined function

User-defined functions are created similarly to how procedures are created, but are instead annotated with `@UserFunction` and instead of returning a stream of values it returns a single value. Valid output types are `long`, `Long`, `double`, `Double`, `boolean`, `Boolean`, `String`, `Node`, `Relationship`, `Path`, `Map<String, Object>`, or `List<T>`, where `T` can be any of the supported types.

For more details, see the [API documentation for user-defined functions](https://neo4j.com/docs/java-reference/3.2/javadocs/org/neo4j/procedure/UserFunction.html) (<https://neo4j.com/docs/java-reference/3.2/javadocs/org/neo4j/procedure/UserFunction.html>).



The correct way to signal an error from within a function is to throw a `RuntimeException`.

```
package example;

import org.neo4j.procedure.Name;
import org.neo4j.procedure.Procedure;
import org.neo4j.procedure.UserFunction;

public class Join
{
    @UserFunction
    @Description("example.join(['s1','s2',...], delimiter) - join the given strings with the given
    delimiter.")
    public String join(
        @Name("strings") List<String> strings,
        @Name(value = "delimiter", defaultValue = ",") String delimiter) {
        if (strings == null || delimiter == null) {
            return null;
        }
        return String.join(delimiter, strings);
    }
}
```

20.2.1. Writing integration tests

Tests for user-defined functions are created in the same way as those for procedures.

Below is a template for testing a user-defined function that joins a list of strings.

Writing tests for the `join` user-defined function

```
package example;

import org.junit.Rule;
import org.junit.Test;
import org.neo4j.driver.v1.*;
import org.neo4j.harness.junit.Neo4jRule;

import static org.hamcrest.core.IsEqual.equalTo;
import static org.junit.Assert.assertThat;

public class JoinTest
{
    // This rule starts a Neo4j instance
    @Rule
    public Neo4jRule neo4j = new Neo4jRule()

        // This is the function we want to test
        .withFunction( Join.class );

    @Test
    public void shouldAllowIndexingAndFindingANode() throws Throwable
    {
        // This is in a try-block, to make sure we close the driver after the test
        try( Driver driver = GraphDatabase.driver( neo4j.boltURI(), Config.build().withEncryptionLevel(
        Config.EncryptionLevel.NONE ).toConfig() ) )
        {
            // Given
            Session session = driver.session();

            // When
            String result = session.run( "RETURN example.join(['Hello', 'World']) AS result" ).single().
            get("result").asString();

            // Then
            assertThat( result, equalTo( "Hello,World" ) );
        }
    }
}
```

20.3. User-defined aggregation functions

This section covers how to write, test and deploy a user-defined aggregation function for Neo4j.

User-defined aggregation functions are functions that aggregate data and return a single result.

20.3.1. Calling a user-defined aggregation function

User-defined aggregation functions are called in the same way as any other Cypher aggregation function. The function name must be fully qualified, so a function named `longestString` defined in the package `org.neo4j.examples` could be called using:

```
MATCH (p: Person) WHERE p.age = 36 RETURN org.neo4j.examples.longestString(p.name)
```

20.3.2. Writing a user-defined aggregation function

User-defined aggregation functions are annotated with `@UserAggregationFunction`. The annotated function must return an instance of an aggregator class. An aggregator class contains one method annotated with `@UserAggregationUpdate` and one method annotated with `@UserAggregationResult`. The method annotated with `@UserAggregationUpdate` will be called multiple times and allows the class to aggregate data. When the aggregation is done the method annotated with `@UserAggregationResult` is called once and the result of the aggregation will be returned. Valid output types are `long`, `Long`,

`double, Double, boolean, Boolean, String, Node, Relationship, Path, Map<String, Object, or List<T>,`
where `T` can be any of the supported types.

For more details, see the [API documentation for user-defined aggregation functions](https://neo4j.com/docs/java-reference/3.2/javadocs/org/neo4j/procedure/UserAggregationFunction.html) (<https://neo4j.com/docs/java-reference/3.2/javadocs/org/neo4j/procedure/UserAggregationFunction.html>).



The correct way to signal an error from within an aggregation function is to throw a `RuntimeException`.

```
package example;

import org.neo4j.procedure.Description;
import org.neo4j.procedure.Name;
import org.neo4j.procedure.UserAggregationFunction;
import org.neo4j.procedure.UserAggregationResult;
import org.neo4j.procedure.UserAggregationUpdate;

public class LongestString
{
    @UserAggregationFunction
    @Description( "org.neo4j.function.example.longestString(string) - aggregates the longest string found" )
    public LongStringAggregator longestString()
    {
        return new LongStringAggregator();
    }

    public static class LongStringAggregator
    {
        private int longest;
        private String longestString;

        @UserAggregationUpdate
        public void findLongest(
            @Name( "string" ) String string )
        {
            if ( string != null && string.length() > longest)
            {
                longest = string.length();
                longestString = string;
            }
        }

        @UserAggregationResult
        public String result()
        {
            return longestString;
        }
    }
}
```

Writing integration tests

Tests for user-defined aggregation functions are created in the same way as those for normal user-defined functions.

Below is a template for testing a user-defined aggregation function that finds the longest string.

Writing tests for the `longestString` user-defined function

```
package example;

import org.junit.Rule;
import org.junit.Test;
import org.neo4j.driver.v1.*;
import org.neo4j.harness.junit.Neo4jRule;

import static org.hamcrest.core.AreEqual;
import static org.junit.Assert.assertThat;

public class LongestStringTest
{
    // This rule starts a Neo4j instance
    @Rule
    public Neo4jRule neo4j = new Neo4jRule()

        // This is the function we want to test
        .withAggregationFunction( LongestString.class );

    @Test
    public void shouldAllowIndexingAndFindingANode() throws Throwable
    {
        // This is in a try-block, to make sure we close the driver after the test
        try( Driver driver = GraphDatabase.driver( neo4j.boltURI() , Config.build().withEncryptionLevel(
Config.EncryptionLevel.NONE ).toConfig() ) )
        {
            // Given
            Session session = driver.session();

            // When
            String result = session.run( "UNWIND [\"abc\", \"abcd\", \"ab\"] AS string RETURN
example.longestString(string) AS result").single().get("result").asString();

            // Then
            assertThat( result, equalTo( "abcd" ) );
        }
    }
}
```

Chapter 21. Authentication and authorization plugins

This chapter describes Neo4j support for custom-built authentication and authorization plugins.

Neo4j provides authentication and authorization plugin interfaces to support real-world deployment scenarios not covered by native users or the built-in configuration-based LDAP connector.

The SPI lives in the org.neo4j.server.security.enterprise.auth.plugin.spi package. Custom built plugins have access to the `neo4j.conf` configuration file in case you want to load any custom settings. Plugins can also write to the security event log.

The authentication plugin implements the AuthPlugin interface with the authenticate method. The example below shows a minimal authentication plugin that checks for neo4j user with neo4j password.

```
@Override
public AuthenticationInfo authenticate( AuthToken authToken )
{
    String principal = authToken.principal();
    char[] credentials = authToken.credentials();

    if ( principal.equals( "neo4j" ) && Arrays.equals( credentials, "neo4j".toCharArray() ) )
    {
        return (AuthenticationInfo) () -> "neo4j";
    }
    return null;
}
```

The authorization plugin implements the AuthPlugin interface with the authorize method. The example below shows a minimal authorization plugin that assigns the reader role to a user named neo4j. Note the usage of the helper class PredefinedRole.

```
@Override
public AuthorizationInfo authorize( Collection<PrincipalAndProvider> principals )
{
    if ( principals.stream().anyMatch( p -> "neo4j".equals( p.principal() ) ) )
    {
        return (AuthorizationInfo) () -> Collections.singleton( PredefinedRoles.READER );
    }
    return null;
}
```

There is also a simplified combined plugin interface that provides both authentication and authorization in a single method called authenticateAndAuthorize. The example below shows a combined plugin verifying neo4j/neo4j credentials and returning reader role authorization:

```
@Override
public AuthInfo authenticateAndAuthorize( AuthToken authToken )
{
    String principal = authToken.principal();
    char[] credentials = authToken.credentials();

    if ( principal.equals( "neo4j" ) && Arrays.equals( credentials, "neo4j".toCharArray() ) )
    {
        return AuthInfo.of( "neo4j", Collections.singleton( PredefinedRoles.READER ) );
    }
    return null;
}
```

Neo4j provides an extendable platform as some user deployment scenarios may not be easily configured through standard LDAP connector. One known complexity is integrating with LDAP user directory where groups have users as a member and the not other way around. The example below first searches for a group that the user is member of, and then maps that group to the Neo4j role by calling the custom-built `getNeo4jRoleForGroupId` method:

```

@Override
public AuthInfo authenticateAndAuthorize( AuthToken authToken ) throws AuthenticationException
{
    try
    {
        String username = authToken.principal();
        char[] password = authToken.credentials();

        LdapContext ctx = authenticate( username, password );
        Set<String> roles = authorize( ctx, username );

        return AuthInfo.of( username, roles );
    }
    catch ( NamingException e )
    {
        throw new AuthenticationException( e.getMessage() );
    }
}

private LdapContext authenticate( String username, char[] password ) throws NamingException
{
    Hashtable<String, Object> env = new Hashtable<>();
    env.put( Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory" );
    env.put( Context.PROVIDER_URL, "ldap://0.0.0.0:10389" );

    env.put( Context.SECURITY_PRINCIPAL, String.format( "cn=%s,ou=users,dc=example,dc=com", username ) );
    env.put( Context.SECURITY_CREDENTIALS, password );

    return new InitialLdapContext( env, null );
}

private Set<String> authorize( LdapContext ctx, String username ) throws NamingException
{
    Set<String> roleNames = new LinkedHashSet<>();

    // Set up our search controls
    SearchControls searchCtl = new SearchControls();
    searchCtl.setSearchScope( SearchControls.SUBTREE_SCOPE );
    searchCtl.setReturningAttributes( new String[]{GROUP_ID} );

    // Use a search argument to prevent potential code injection
    Object[] searchArguments = new Object[]{username};

    // Search for groups that has the user as a member
    NamingEnumeration result = ctx.search( GROUP_SEARCH_BASE, GROUP_SEARCH_FILTER, searchArguments, searchCtl );

    if ( result.hasMoreElements() )
    {
        SearchResult searchResult = (SearchResult) result.next();

        Attributes attributes = searchResult.getAttributes();
        if ( attributes != null )
        {
            NamingEnumeration attributeEnumeration = attributes.getAll();
            while ( attributeEnumeration.hasMore() )
            {
                Attribute attribute = (Attribute) attributeEnumeration.next();
                String attributeId = attribute.getId();
                if ( attributeId.equalsIgnoreCase( GROUP_ID ) )
                {
                    // We found a group that the user is a member of. See if it has a role mapped to it
                    String groupId = (String) attribute.get();
                    String neo4jGroup = getNeo4jRoleForGroupId( groupId );
                    if ( neo4jGroup != null )
                    {
                        // Yay! Add it to our set of roles
                        roleNames.add( neo4jGroup );
                    }
                }
            }
        }
    }
    return roleNames;
}

```

Read more about this and other plugin examples at <https://github.com/neo-technology/neo4j->

Appendix A: Reference

This appendix contains a complete reference of Neo4j status codes.

Chapter 22. 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.



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.

22.1. 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
TransientError	The database cannot service the request right now, retrying later might yield a successful outcome.	Rollback
DatabaseError	The database failed to service the request.	Rollback

22.2. Status codes

This is a complete list of all status codes Neo4j may return, and what they mean.

Status Code	Description
Neo.ClientError.Cluster.NotALeader	The request cannot be processed by this server. Write requests can only be processed by the leader.
Neo.ClientError.General.ForbiddenOnReadOnlyDatabase	This is a read only database, writing or modifying the database is not allowed.
Neo.ClientError.General.InvalidArguments	The request contained fields that were empty or are not allowed.
Neo.ClientError.LegacyIndex.LegacyIndexNotFound	The request (directly or indirectly) referred to a legacy index that does not exist.
Neo.ClientError.Procedure.ProcedureCallFailed	Failed to invoke a procedure. See the detailed error description for exact cause.
Neo.ClientError.Procedure.ProcedureNotFound	A request referred to a procedure that is not registered with this database instance. If you are deploying custom procedures in a cluster setup, ensure all instances in the cluster have the procedure jar file deployed.
Neo.ClientError.Procedure.ProcedureRegistrationFailed	The database failed to register a procedure, refer to the associated error message for details.
Neo.ClientError.Procedure.ProcedureTimedOut	The procedure has not completed within the specified timeout. You may want to retry with a longer timeout.
Neo.ClientError.Procedure.TypeError	A procedure is using or receiving a value of an invalid type.
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.

Status Code	Description
<code>Neo.ClientError.Request.TransactionRequired</code>	The request cannot be performed outside of a transaction, and there is no transaction present to use. Wrap your request in a transaction and retry.
<code>Neo.ClientError.Schema.ConstraintAlreadyExists</code>	Unable to perform operation because it would clash with a pre-existing constraint.
<code>Neo.ClientError.Schema.ConstraintNotFound</code>	The request (directly or indirectly) referred to a constraint that does not exist.
<code>Neo.ClientError.Schema.ConstraintValidationFailed</code>	A constraint imposed by the database was violated.
<code>Neo.ClientError.Schema.ConstraintVerificationFailed</code>	Unable to create constraint because data that exists in the database violates it.
<code>Neo.ClientError.Schema.ForbiddenOnConstraintIndex</code>	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.
<code>Neo.ClientError.Schema.IndexAlreadyExists</code>	Unable to perform operation because it would clash with a pre-existing index.
<code>Neo.ClientError.Schema.IndexNotApplicable</code>	The request did not contain the properties required by the index.
<code>Neo.ClientError.Schema.IndexNotFound</code>	The request (directly or indirectly) referred to an index that does not exist.
<code>Neo.ClientError.Schema.RepeatedPropertyInCompositeSchema</code>	Unable to create composite index or constraint because a property was specified in several positions.
<code>Neo.ClientError.Schema.TokenNameError</code>	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.
<code>Neo.ClientError.Security.AuthenticationRateLimit</code>	The client has provided incorrect authentication details too many times in a row.
<code>Neo.ClientError.Security.AuthorizationExpired</code>	The stored authorization info has expired. Please reconnect.
<code>Neo.ClientError.Security.CredentialsExpired</code>	The credentials have expired and need to be updated.
<code>Neo.ClientError.Security.EncryptionRequired</code>	A TLS encrypted connection is required.
<code>Neo.ClientError.Security.Forbidden</code>	An attempt was made to perform an unauthorized action.
<code>Neo.ClientError.Security.Unauthorized</code>	The client is unauthorized due to authentication failure.
<code>Neo.ClientError.Statement.ArgumentError</code>	The statement is attempting to perform operations using invalid arguments
<code>Neo.ClientError.Statement.ArithmeticError</code>	Invalid use of arithmetic, such as dividing by zero.
<code>Neo.ClientError.Statement.ConstraintVerificationFailed</code>	A constraint imposed by the statement is violated by the data in the database.
<code>Neo.ClientError.Statement.EntityNotFound</code>	The statement refers to a non-existent entity.
<code>Neo.ClientError.Statement.ExternalResourceFailed</code>	Access to an external resource failed
<code>Neo.ClientError.Statement.LabelNotFound</code>	The statement is referring to a label that does not exist.
<code>Neo.ClientError.Statement.ParameterMissing</code>	The statement refers to a parameter that was not provided in the request.
<code>Neo.ClientError.Statement.PropertyNotFound</code>	The statement refers to a non-existent property.
<code>Neo.ClientError.Statement.SemanticError</code>	The statement is syntactically valid, but expresses something that the database cannot do.
<code>Neo.ClientError.Statement.SyntaxException</code>	The statement contains invalid or unsupported syntax.
<code>Neo.ClientError.Statement.TypeError</code>	The statement is attempting to perform operations on values with types that are not supported by the operation.

Status Code	Description
Neo.ClientError.Transaction.ForbiddenDueToTransactionType	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.InvalidBookmark	Supplied bookmark cannot be interpreted. You should only supply a bookmark previously that was previously generated by Neo4j. Maybe you have generated your own bookmark, or modified a bookmark since it was generated by Neo4j.
Neo.ClientError.Transaction.TransactionAccessedConcurrently	There were concurrent requests accessing the same transaction, which is not allowed.
Neo.ClientError.Transaction.TransactionEventHandlerFailed	A transaction event handler threw an exception. The transaction will be rolled back.
Neo.ClientError.Transaction.TransactionHookFailed	Transaction hook failure.
Neo.ClientError.Transaction.TransactionMarkedAsFailed	Transaction was marked as both successful and failed. Failure takes precedence and so this transaction was rolled back although it may have looked like it was going to be committed
Neo.ClientError.Transaction.TransactionNotFound	The request referred to a transaction that does not exist.
Neo.ClientError.Transaction.TransactionTerminated	The current transaction has been marked as terminated, meaning no more interactions with it are allowed. There are several reasons this happens - the client might have asked for the transaction to be terminated, an operator might have asked for the database to be shut down, or the current instance is about to go through a cluster role switch. Simply retry your operation in a new transaction.
Neo.ClientError.Transaction.TransactionTimedOut	The transaction has not completed within the specified timeout. You may want to retry with a longer timeout.
Neo.ClientError.Transaction.TransactionValidationFailed	Transaction changes did not pass validation checks
Neo.ClientNotification.Procedure.ProcedureWarning	The query used a procedure that generated a warning.
Neo.ClientNotification.Statement.CartesianProductWarning	This query builds a cartesian product between disconnected patterns.
Neo.ClientNotification.Statement.CreateUniqueUnavailableWarning	CREATE UNIQUE is not available in the current CYPHER version, the query has been run by an older CYPHER version.
Neo.ClientNotification.Statement.DynamicPropertyWarning	Queries using dynamic properties will use neither index seeks nor index scans for those properties
Neo.ClientNotification.Statement.EagerOperatorWarning	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.ExhaustiveShortestPathWarning	Exhaustive shortest path has been planned for your query that means that shortest path graph algorithm might not be used to find the shortest path. Hence an exhaustive enumeration of all paths might be used in order to find the requested shortest path.
Neo.ClientNotification.Statement.FeatureDeprecationWarning	This feature is deprecated and will be removed in future versions.
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.NoApplicableIndexWarning	Adding a schema index may speed up this query.
Neo.ClientNotification.Statement.PlannerUnavailableWarning	The RULE planner is not available in the current CYPHER version, the query has been run by an older CYPHER version.

Status Code	Description
Neo.ClientNotification.Statement.PlannerUnsupportedWarning	This query is not supported by the COST planner.
Neo.ClientNotification.Statement.RuntimeUnsupportedWarning	This query is not supported by the compiled runtime.
Neo.ClientNotification.Statement.UnboundedVariableLengthPatternWarning	The provided pattern is unbounded, consider adding an upper limit to the number of node hops.
Neo.ClientNotification.Statement.UnknownLabelWarning	The provided label is not in the database.
Neo.ClientNotification.Statement.UnknownPropertyKeyWarning	The provided property key is not in the database
Neo.ClientNotification.Statement.UnknownRelationshipTypeWarning	The provided relationship type is not in the database.
Neo.DatabaseError.General.IndexCorruptionDetected	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.SchemaCorruptionDetected	A malformed schema rule was encountered. Please contact your support representative.
Neo.DatabaseError.General.StorageDamageDetected	Expected set of files not found on disk. Please restore from backup.
Neo.DatabaseError.General.UnknownError	An unknown error occurred.
Neo.DatabaseError.Schema.ConstraintCreationFailed	Creating a requested constraint failed.
Neo.DatabaseError.Schema.ConstraintDropFailed	The database failed to drop a requested constraint.
Neo.DatabaseError.Schema.IndexCreationFailed	Failed to create an index.
Neo.DatabaseError.Schema.IndexDropFailed	The database failed to drop a requested index.
Neo.DatabaseError.Schema.LabelAccessFailed	The request accessed a label that did not exist.
Neo.DatabaseError.Schema.LabelLimitReached	The maximum number of labels supported has been reached, no more labels can be created.
Neo.DatabaseError.Schema.PropertyKeyAccessFailed	The request accessed a property that does not exist.
Neo.DatabaseError.Schema.RelationshipTypeAccessFailed	The request accessed a relationship type that does not exist.
Neo.DatabaseError.Schema.SchemaRuleAccessFailed	The request referred to a schema rule that does not exist.
Neo.DatabaseError.Schema.SchemaRuleDuplicateFound	The request referred to a schema rule that is defined multiple times.
Neo.DatabaseError.Statement.ExecutionFailed	The database was unable to execute the statement.
Neo.DatabaseError.Transaction.TransactionCommitFailed	The database was unable to commit the transaction.
Neo.DatabaseError.Transaction.TransactionLogError	The database was unable to write transaction to log.
Neo.DatabaseError.Transaction.TransactionRollbackFailed	The database was unable to roll back the transaction.
Neo.DatabaseError.Transaction.TransactionStartFailed	The database was unable to start the transaction.
Neo.TransientError.Cluster.NoLeaderAvailable	No leader available at the moment. Retrying your request at a later time may succeed.
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.General.OutOfMemoryError	There is not enough memory to perform the current task. Please try increasing 'dbms.memory.heap.max_size' in the neo4j configuration (normally in 'conf/neo4j.conf' or, if you are using Neo4j Desktop, found through the user interface) or if you are running an embedded installation increase the heap by using '-Xmx' command line flag, and then restart the database.

Status Code	Description
Neo.TransientError.General.StackOverFlowError	There is not enough stack size to perform the current task. This is generally considered to be a database error, so please contact Neo4j support. You could try increasing the stack size: for example to set the stack size to 2M, add `dbms.jvm.additional=-Xss2M` to in the neo4j configuration (normally in 'conf/neo4j.conf' or, if you are using Neo4j Desktop, found through the user interface) or if you are running an embedded installation just add -Xss2M as command line flag.
Neo.TransientError.Network.CommunicationError	An unknown network failure occurred, a retry may resolve the issue.
Neo.TransientError.Schema.SchemaModifiedConcurrently	The database schema was modified while this transaction was running, the transaction should be retried.
Neo.TransientError.Security.AuthProviderFailed	An auth provider request failed.
Neo.TransientError.Security.AuthProviderTimeout	An auth provider request timed out.
Neo.TransientError.Security.ModifiedConcurrently	The user was modified concurrently to this request.
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.
Neo.TransientError.Transaction.InstanceStateChanged	Transactions rely on assumptions around the state of the Neo4j instance they execute on. For instance, transactions in a cluster may expect that they are executing on an instance that can perform writes. However, instances may change state while the transaction is running. This causes assumptions the instance has made about how to execute the transaction to be violated - meaning the transaction must be rolled back. If you see this error, you should retry your operation in a new transaction.
Neo.TransientError.Transaction.Interrupted	Interrupted while waiting.
Neo.TransientError.Transaction.LockAcquisitionTimeout	Unable to acquire lock within configured timeout.
Neo.TransientError.Transaction.LockClientStopped	Transaction terminated, no more locks can be acquired.
Neo.TransientError.Transaction.LockSessionExpired	The lock session under which this transaction was started is no longer valid.
Neo.TransientError.Transaction.Outdated	Transaction has seen state which has been invalidated by applied updates while transaction was active. Transaction may succeed if retried.
Neo.TransientError.Transaction.Terminated	Explicitly terminated by the user.

Appendix B: Terminology

This section provides a cross-linked glossary of key terms for working with graph databases in general, and with Neo4j and Cypher in particular.

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.

acquire (connection)

To borrow a driver connection that is not currently in use from a connection pool.

acyclic

for a graph or subgraph: when there is no way to start at some node `n` and follow a sequence of adjacent relationships that eventually loops back to `n` again. The opposite of [cyclic](#).

adjacent

[nodes](#) sharing an [incident](#) (that is, directly-connected) [relationship](#) or [relationships](#) sharing an incident node.

attribute

Synonym for [property](#).

arc

graph theory: a synonym for a [directed relationship](#).

array

container that holds a number of elements. The element types can be the types supported by the underlying graph storage layer, but all elements must be of the same type.

aggregating expression

expression that summarizes a set of values, like computing their sum or their maximum.

Bolt

Bolt is a Neo4j proprietary, binary protocol used for communication between client applications and database servers. Bolt is versioned independently from the database and the drivers.

Bolt Routing Protocol

The steps required for a driver to obtain a routing table from a cluster member.

Bolt server

A Neo4j instance that can accept incoming Bolt connections.

bookmark

A marker for a point in the transactional history of Neo4j.

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`.

client application

A piece of software that interacts with a database server via a driver.

cluster member

A server that is part of a cluster.

co-incident

alternative term for [adjacent relationships](#), which share a common [node](#).

collection

container that holds a number of elements. The elements can have mixed types. This is the deprecated name of the Cypher type now called [List](#).

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.

connection

A persistent communication channel between a client application and a database server.

connection pool

A set of connections maintained for quick access, that can be acquired and released as required.

constraint

part of a database schema: defines a contract that the database will never break (for example, uniqueness of a [property](#) on all [nodes](#) that have a specific [label](#)).

cyclic

The opposite of [acyclic](#).

Cypher

a special-purpose programming language for describing [queries](#) and operations on a [graph database](#), with accompanying natural language concepts.

Cypher type system

The types used by Cypher.

DAG

a directed, [acyclic graph](#): there are no [cyclic paths](#) and all the [relationships](#) are directed.

data graph

[graph](#) stored in the database. See also [property graph](#).

data record

a unit of storage containing an arbitrary unordered collection of properties.

degree

of a node: is the number of relationships leaving or entering (if directed) the node; loops are counted twice.

direct driver

A driver that can connect to a single server address.

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.

driver (object)

A globally accessible controller for all database access.

driver (package)

A software library that provides access to Neo4j from a particular programming language. The Neo4j drivers implement the [Bolt](#) protocol.

edge

[graph theory](#): a synonym for undirected [relationship](#).

execution result

all statements return an execution result. For [queries](#), this can contain an iterator of [result rows](#).

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.

expression

produces values; may be used in [projections](#), as a [predicate](#), or when setting [properties](#) on [graph elements](#).

graph

1. [data graph](#),
2. [property graph](#),
3. [graph theory](#): set of [vertices](#) and [edges](#).

graph database

a database that uses [graph](#)-based structures (for example, [nodes](#), [relationships](#), [properties](#)) to represent and store data.

graph element

a [node](#), [relationship](#), or [path](#) which is part of a [graph](#).

variable

variables are named bindings to values (for example, lists, scalars) in a [statement](#). For example, in `MATCH (n) RETURN n`, `n` is a variable.

incident

[adjacent relationship](#) attached to a [node](#) or a node attached to a relationship.

incoming relationship

pertaining to a directed relationship: from the point of view of a [node](#) `n`, this is any [relationship](#) `r` arriving at `n`, exemplified by `(-)[:-r]-(n)`. The opposite of [outgoing](#).

index

data structure that improves performance of a database by redundantly storing the same information in a way that is faster to read.

intermediate result

set of variables and values (record) passed from one clause to another during query execution. This is internal to the execution of a given query.

label

marks a [node](#) as a member of a named subset. A node may be assigned zero or more labels. Labels are written as `:label` in Cypher (the actual label is prefixed by a colon). Note: *graph theory*: This differs from mathematical graphs, where a label applies uniquely to a single vertex.

list

container that holds a number of elements. The elements may have mixed types. This is the name of one of the types in the Cypher type system.

loop

a relationship that connects a node to itself.

neighbor

of node: another [node](#), connected by a common [relationship](#); *of relationship*: another relationship, connected to a common node.

*node**

[data record](#) within a [data graph](#); contains an arbitrary collection of [properties](#). Nodes may have zero, one, or more [labels](#) and are optionally connected by [relationships](#). Similar to [vertex](#).

null

[NULL](#) is a special marker, used to indicate that a data item does not exist in the [graph](#) or that the value of an [expression](#) is unknown or inapplicable.

operator

there are three categories of operators in Cypher:

1. *Arithmetic*, such as `+`, `/`, `%` etc.;
2. *Logical*, such as `OR`, `AND`, `NOT` etc.; and
3. *Comparison*, such as `<`, `>`, `=` etc.

outgoing relationship

pertaining to a directed relationship: from the point of view of a [node](#) `n`, this is any [relationship](#) `r` leaving `n`, exemplified by `(n)-[:r]>()`. The opposite of [incoming relationship](#).

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.

*path**

collection of alternating [nodes](#) and [relationships](#) that corresponds to a walk in the [data graph](#).

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.

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 [variables](#) provided in the input, a calculation based on variables in the input, or both. The relevant clauses are [WITH](#) and [RETURN](#).

*property**

named value stored in a [node](#) or [relationship](#). Synonym for [attribute](#).

property graph

a [graph](#) having [directed](#), [typed relationships](#). Each [node](#) or relationship may have zero or more associated [properties](#).

*query**

statement that reads or writes data from the database

*relationship**

[data record](#) in a [property graph](#) that associates an ordered pair of [nodes](#). Similar to [arc](#) and [edge](#).

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.

release (connection)

To return a connection back into a connection pool after use.

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).

role (clustering)

An operation that can be carried out by a cluster member (e.g. [read](#), [write](#)).

rollback

abort of the containing [transaction](#), effectively undoing any changes defined inside the transaction.

routing driver

A driver that can route traffic to multiple members of a cluster using the routing protocol.

routing table

A set of server addresses that identify cluster members associated with roles.

schema

persistent database state that describes available [indexes](#) and enabled [constraints](#) for the [data graph](#).

schema command

[statement](#) that updates the [schema](#).

server address

A combination of host name and port or IP address and port that targets a server.

session

A causally linked sequence of transactions.

statement

text string containing a [Cypher query](#) or [command](#).

statement result

The stream of records that are returned on execution of a statement.

thread safety

See https://en.wikipedia.org/wiki/Thread_safety.

type

types classify values. Each value in [Cypher](#) has a concrete type. Supported types are:

- string,
- boolean,
- the number types (double, integer, long),
- the map types (plain maps, nodes, and relationships),
- and lists of any concrete type.

The type hierarchy supports several other types (for example, any, scalar, derived map, list). These are used to classify values and [lists](#) of values having different concrete types.

transaction

A transaction comprises a unit of work performed against a database. It is treated in a coherent and reliable way, independent of other transactions. A transaction, by definition, must be atomic, consistent, isolated, and durable.

transitive closure

of a graph: is a [graph](#) which contains a [relationship](#) from node [x](#) to node [y](#) whenever there is a directed [path](#) from [x](#) to [y](#); For example, if there is a relationship from [a](#) to [b](#), and another from [b](#) to [c](#), then the transitive closure includes a relationship from [a](#) to [c](#).

undirected relationship

a [relationship](#) that doesn't have a direction. The opposite of [directed relationship](#).

value

A unit of data belonging to the Cypher type system.

vertex

graph theory: the fundamental unit used to form a mathematical graph (plural: vertices). See [node](#).

License

Creative Commons 3.0

You are free to

Share

copy and redistribute the material in any medium or format

Adapt

remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms

Attribution

You must give appropriate credit, provide a link to the license, and indicate if changes were made.
You may do so in any reasonable manner, but not in any way that suggests the licensor endorses

you or your use.

ShareAlike

If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions

You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

See <http://creativecommons.org/licenses/by-sa/3.0/> for further details. The full license text is available at <http://creativecommons.org/licenses/by-sa/3.0/legalcode>.