



The Neo4j Developer Manual

v3.4

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	31
7. Clauses	93
8. Functions	171
9. Schema	295
10. Query tuning	316
11. Execution plans	334
12. Deprecations, additions and compatibility	414
13. Glossary of keywords	418
Drivers	427
14. Get started	429
15. Client applications	432
16. Sessions and transactions	438
17. Working with Cypher values	444
HTTP API	448
18. Transactional Cypher HTTP endpoint	449
19. Authentication and authorization	459
Extending Neo4j	462
20. Procedures	463
21. User-defined functions	471
22. Authentication and authorization plugins	475
Appendix A: Reference	479
23. Neo4j Status Codes	480
Appendix B: Terminology	485

This is the developer manual for Neo4j version 3.4, 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

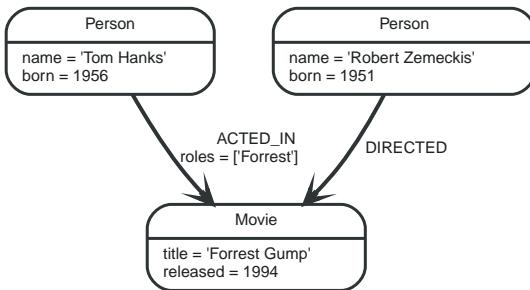
This chapter contains an introduction to the graph data model.

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:

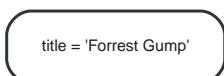


2.1.1. Nodes

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) (<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

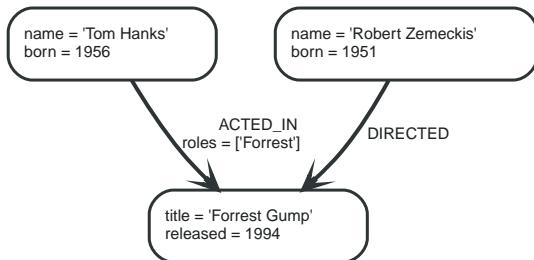
A relationship in Neo4j is a relationship as described in the [property graph model](https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc) (<https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc#pgm-definitions-relationship>), with a relationship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

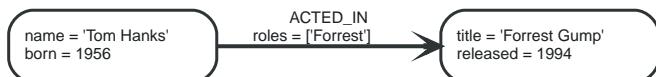
Relationships organize nodes into arbitrary structures, allowing a graph to resemble a list, a tree, a map, or a compound entity — any of which may be combined into yet more complex, richly interconnected structures.

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:

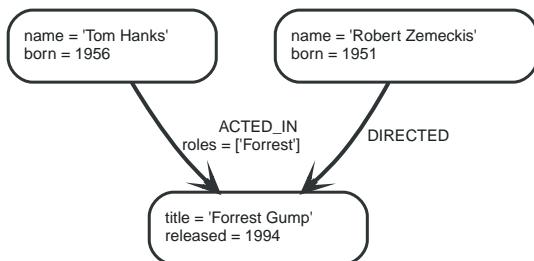


Table 1. Using relationship direction and type

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 types are:

- *Number*, an abstract type, which has the following subtypes:
 - Integer
 - Float
- String
- Boolean
- *Spatial types*:
 - Point
- *Temporal types*:
 - Date
 - Time
 - LocalTime
 - DateTime
 - LocalDateTime
 - Duration



`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.

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.

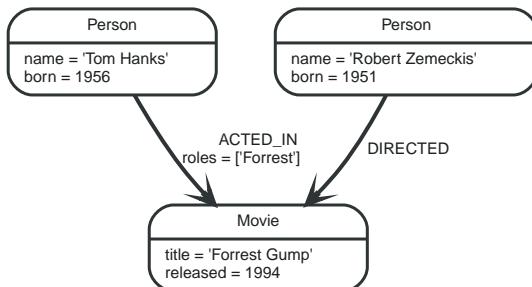
A label is a named graph construct that is used to group nodes into sets; all nodes labeled with the same label belongs to the same set. Many database queries can work with these sets instead of the whole graph, making queries easier to write and more efficient to execute. A node may be labeled with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

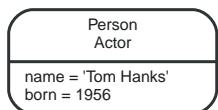
For example, all nodes representing users could be labeled with the label `:User`. With that in place, you can ask Neo4j to perform operations only on your user nodes, such as finding all users with a given name.

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://s3.amazonaws.com/artifacts.opencypher.org/M06/docs/style-guide.pdf) (<https://s3.amazonaws.com/artifacts.opencypher.org/M06/docs/style-guide.pdf>).

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

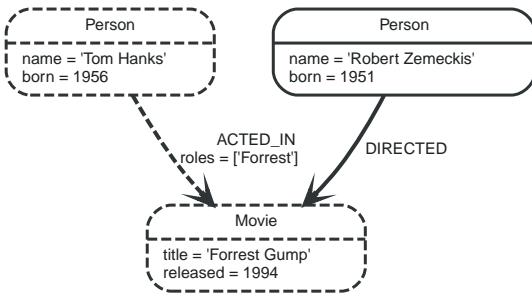
A traversal navigates through a graph to find paths.

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.

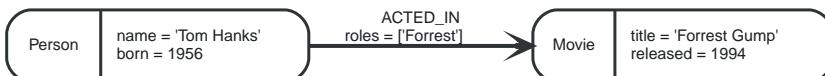
If we want to find out which movies Tom Hanks acted in according to our tiny example database, the traversal would start from the `Tom Hanks` node, follow any `:ACTED_IN` relationships connected to the node, and end up with `Forrest Gump` as the result (see the dashed lines):



2.1.6. Paths

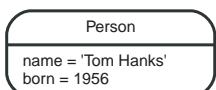
A path in Neo4j is a path as described in the [property graph model](#) (<https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc#pgm-definitions-path>). Paths are retrieved from a Cypher query or traversal.

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

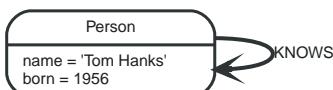


The path above has length one.

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.

The easiest way to set up an environment for developing an application with Neo4j and Cypher is to use Neo4j Desktop. Download Neo4j Desktop from <https://neo4j.com/download/> and follow the installation instructions for your operating system. Neo4j Desktop manages installation of the Neo4j database and provides access to many useful development tools.

Read more about deploying Neo4j for development or production in the [Neo4j Operations Manual ▾ Installation](#). To get started with Cypher, continue reading the [Get started with Cypher](#) guide. The full reference for Neo4j Cypher is found in the [Cypher query language](#) chapter.

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)`, `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.

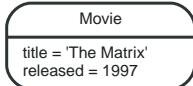
For the patterns we've looked at so far this could look like the following:

```
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:

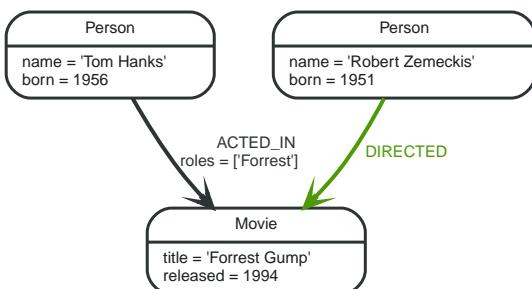
```
+-----+
| p |
+-----+
| Node[1]{name:"Keanu Reeves",born:1964} |
+-----+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

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.

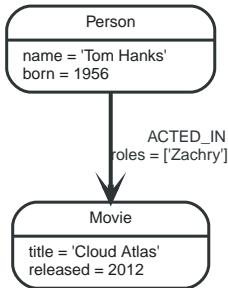
Now we know enough to connect new nodes to existing ones and can combine `MATCH` and `CREATE` to attach structures to the graph.

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.



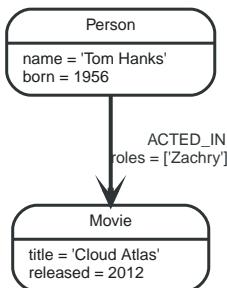
A **MERGE** clause without any previously assigned variables in it either matches the full pattern or creates the full pattern. It never produces a partial mix of matching and creating within a pattern. To achieve a partial match/create, make sure to use already defined variables for the parts that shouldn't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

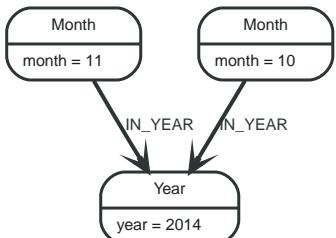


In case the direction of a relationship is arbitrary, you can leave off the arrowhead. **MERGE** will then check for the relationship in either direction, and create a new directed relationship if no matching relationship was found.

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.

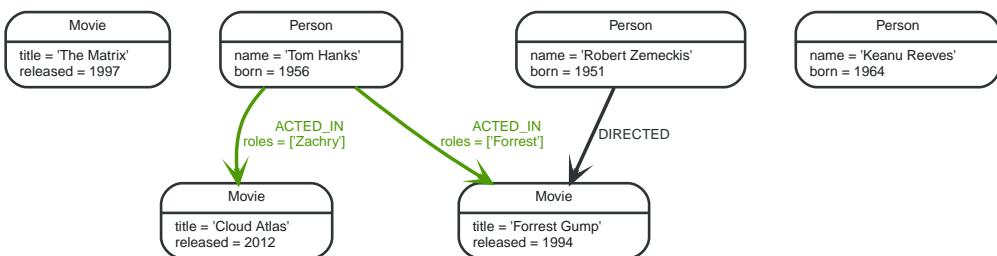
4.3. Getting correct results

- [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)`, `toInteger("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, toUpper(p.name), coalesce(p.nickname,"n/a") AS nickname, { name: p.name,
  label:head(labels(p))} AS person
```

p	name	toUpper(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: toInteger(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: toInteger(csvLine.id), title: csvLine.title, year:toInteger(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: toInteger(csvLine.personId)}),
      (movie:Movie { id: toInteger(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](#)

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:

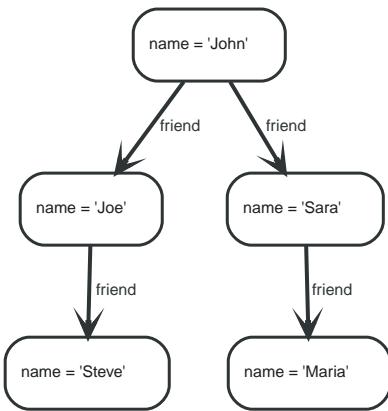


Figure 1. Example Graph

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 unique 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

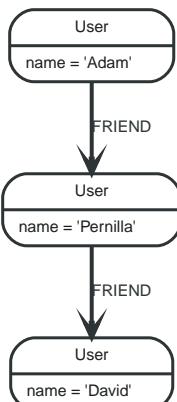
While pattern matching, Neo4j makes sure to not include matches where the same graph relationship is found multiple times in a single pattern. In most use cases, this is a sensible thing to do.

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
```

Chapter 6. Syntax

This section describes the syntax of the Cypher query language.

- [Values and types](#)
- [Naming rules and recommendations](#)
- [Expressions](#)
 - [Expressions in general](#)
 - [Note on string literals](#)
 - [CASE Expressions](#)
- [Variables](#)
- [Reserved keywords](#)
- [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 \(explicit indexes\)](#)
 - [Index query \(explicit indexes\)](#)
- [Operators](#)
 - [Operators at a glance](#)
 - [General operators](#)
 - [Mathematical operators](#)
 - [Comparison operators](#)
 - [Boolean operators](#)
 - [String operators](#)
 - [Temporal 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
- Temporal (Date/Time) values
 - Introduction
 - Time zones
 - Temporal instants
 - Specifying temporal instants
 - Specifying dates
 - Specifying times
 - Specifying time zones
 - Examples
 - Accessing components of temporal instants
 - Durations
 - Specifying durations
 - Examples
 - Accessing components of durations
 - Examples
 - Temporal indexing
 - Spatial values
 - Introduction
 - Coordinate Reference Systems
 - Geographic coordinate reference systems
 - Cartesian coordinate reference systems
 - Spatial instants
 - Creating points
 - Accessing components of points
 - Spatial index
- 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 `\[\]` operator and `null`](#)
 - [The `IN` operator and `null`](#)
 - [Expressions that return `null`](#)

6.1. Values and types

Cypher provides first class support for a number of data types.

These fall into several categories which will be described in detail in the following subsections:

- Property types
- Structural types
- Composite types

6.1.1. Property types

- Can be returned from Cypher queries
- Can be used as [parameters](#)
- Can be stored as properties
- Can be constructed with [Cypher literals](#)

Property types comprise:

- Number, an abstract type, which has the subtypes *Integer* and *Float*
- String
- Boolean
- The spatial type *Point*
- Temporal types: *Date*, *Time*, *LocalTime*, *DateTime*, *LocalDateTime* and *Duration*



The adjective 'numeric' — when used in the context of describing Cypher functions or expressions — indicates that any type of Number applies (Integer or Float).



Homogeneous lists of simple types can also be stored as properties, although lists in general (see [Composite types](#)) cannot be stored.



Cypher also provides pass-through support for byte arrays, which can be stored as property values. Byte arrays are *not* considered a first class data type by Cypher, so do not have a literal representation.

6.1.2. Structural types

- Can be returned from Cypher queries
- Cannot be used as [parameters](#)
- Cannot be stored as properties
- Cannot be constructed with [Cypher literals](#)

Structural types comprise:

- Nodes, comprising:
 - Id
 - Label(s)
 - Map (of properties)
- Relationships, comprising:
 - Id
 - Type
 - Map (of properties)
 - Id of the start and end nodes
- Paths
 - An alternating sequence of nodes and relationships



Nodes, relationships, and paths are returned as a result of pattern matching.



Labels are not values but are a form of pattern syntax.

6.1.3. Composite types

- Can be returned from Cypher queries
- Can be used as [parameters](#)
- Cannot be stored as properties
- Can be constructed with [Cypher literals](#)

Composite types comprise:

- **Lists** are heterogeneous, ordered collections of values, each of which has any property, structural or composite type.
- **Maps** are heterogeneous, unordered collections of (key, value) pairs, where:
 - the key is a String
 - the value has any property, structural or composite type



Composite values can also contain [null](#).

Special care must be taken when using [null](#) (see [Working with null](#)).

6.2. Naming rules and recommendations

We describe here rules and recommendations for the naming of node labels, relationship types, property names and [variables](#).

6.2.1. Naming rules

- Must begin with an alphabetic letter.
 - This includes "non-English" characters, such as å, ä, ö, ü etc.
 - If a leading non-alphabetic character is required, use backticks for escaping; e.g. `^n`.
- Can contain numbers, but not as the first character.
 - To illustrate, `1first` is not allowed, whereas `first1` is allowed.
 - If a leading numeric character is required, use backticks for escaping; e.g. `1first`.
- Cannot contain symbols.
 - An exception to this rule is using underscore, as given by `my_variable`.
 - An ostensible exception to this rule is using \$ as the first character to denote a [parameter](#), as given by `$myParam`.
 - If a leading symbolic character is required, use backticks for escaping; e.g. `\$\$n`.
- Can be very long, up to 65535 ($2^{16} - 1$) or 65534 characters, depending on the version of Neo4j.
- Are case-sensitive.
 - Thus, `:PERSON`, `:Person` and `:person` are three different labels, and `n` and `N` are two different variables.
- Whitespace characters:
 - Leading and trailing whitespace characters will be removed automatically. For example, `MATCH (a) RETURN a` is equivalent to `MATCH (a) RETURN a`.
 - If spaces are required within a name, use backticks for escaping; e.g. `my variable has spaces`.

6.2.2. Scoping and namespace rules

- Node labels, relationship types and property names may re-use names.
 - The following query — with `a` for the label, type and property name — is valid: `CREATE (a:a {a: 'a'})-[r:a]-(b:a {a: 'a'})`.
- Variables for nodes and relationships must not re-use names within the same query scope.
 - The following query is not valid as the node and relationship both have the name `a`: `CREATE (a)-[a]-(b)`.

6.2.3. Recommendations

Here are the naming conventions we recommend using:

Node labels	Camel case, beginning with an upper-case character	<code>:VehicleOwner</code> rather than <code>:vehice_owner</code> etc.
-------------	--	--

Relationship types	Upper case, using underscore to separate words	:OWNS_VEHICLE rather than :ownsVehicle etc
--------------------	--	--

6.3. 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](#)

6.3.1. 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.`
- A variable: `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.

6.3.2. Note on string literals

String literals can contain the following escape sequences:

Escape sequence	Character
<code>\t</code>	Tab

Escape sequence	Character
\b	Backspace
\n	Newline
\r	Carriage return
\f	Form feed
\'	Single quote
\"	Double quote
\\\	Backslash
\uxxxx	Unicode UTF-16 code point (4 hex digits must follow the \u)
\Uxxxxxxxxx	Unicode UTF-32 code point (8 hex digits must follow the \U)

6.3.3. 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:

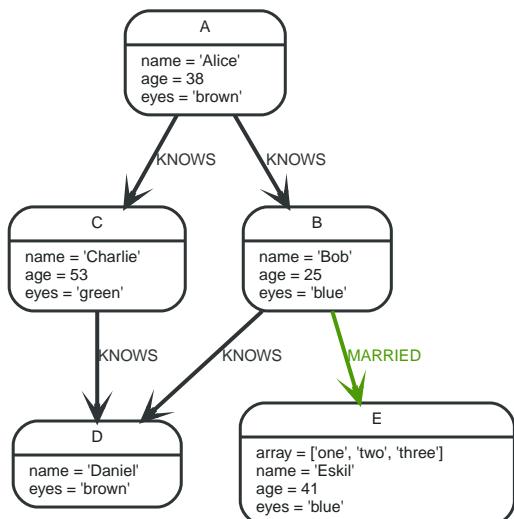


Figure 2. Graph

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
test	A valid expression.
value	An expression whose result will be compared to <code>test</code> .
result	This is the expression returned as output if <code>value</code> matches <code>test</code> .
default	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 2. 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
<code>predicate</code>	A predicate that is tested to find a valid alternative.
<code>result</code>	This is the expression returned as output if <code>predicate</code> evaluates to <code>true</code> .
<code>default</code>	If no match is found, <code>default</code> 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 3. 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 4. 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 5. Result

n.name	age_10_years_ago
"Alice"	28
"Bob"	15
"Charlie"	43
"Daniel"	-1
"Eskil"	31

5 rows

6.4. 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`.

Information regarding the naming of variables may be found [here](#).

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](#).

6.5. Reserved keywords

We provide here a listing of reserved words, grouped by the categories from which they are drawn, all of which have a special meaning in Cypher. In addition to this, we list a number of words that are reserved for future use.

These reserved words are not permitted to be used as identifiers in the following contexts:

- Variables
- Function names
- Parameters

If any reserved keyword is escaped — i.e. is encapsulated by backticks ` , such as `'AND'` — it would become a valid identifier in the above contexts.

6.5.1. Clauses

- CALL
- CREATE
- DELETE
- DETACH
- EXISTS
- FOREACH
- LOAD
- MATCH
- MERGE
- OPTIONAL
- REMOVE
- RETURN
- SET
- START
- UNION
- UNWIND
- WITH

6.5.2. Subclauses

- LIMIT
- ORDER
- SKIP
- WHERE
- YIELD

6.5.3. Modifiers

- ASC
- ASCENDING
- ASSERT
- BY
- CSV
- DESC
- DESCENDING
- ON

6.5.4. Expressions

- ALL
- CASE

- ELSE
- END
- THEN
- WHEN

6.5.5. Operators

- AND
- AS
- CONTAINS
- DISTINCT
- ENDS
- IN
- IS
- NOT
- OR
- STARTS
- XOR

6.5.6. Schema

- CONSTRAINT
- CREATE
- DROP
- EXISTS
- INDEX
- NODE
- KEY
- UNIQUE

6.5.7. Hints

- INDEX
- JOIN
- PERIODIC
- COMMIT
- SCAN
- USING

6.5.8. Literals

- false
- null

- `true`

6.5.9. Reserved for future use

- `ADD`
- `DO`
- `FOR`
- `MANDATORY`
- `OF`
- `REQUIRE`
- `SCALAR`

6.6. 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 \(explicit indexes\)](#)
- [Index query \(explicit indexes\)](#)

6.6.1. 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 explicit 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.

6.6.2. 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
```

6.6.3. Regular expression

Parameters

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

Query

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

6.6.4. Case-sensitive string pattern matching

Parameters

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

Query

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

6.6.5. Create node with properties

Parameters

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

Query

```
CREATE ($props)
```

6.6.6. 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
```

6.6.7. 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
```

6.6.8. SKIP and LIMIT

Parameters

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

Query

```
MATCH (n:Person)  
RETURN n.name  
SKIP $$  
LIMIT $1
```

6.6.9. Node id

Parameters

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

Query

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

6.6.10. Multiple node ids

Parameters

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

Query

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

6.6.11. Calling procedures

Parameters

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

Query

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

6.6.12. Index value (explicit indexes)

Parameters

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

Query

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

6.6.13. Index query (explicit indexes)

Parameters

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

Query

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

6.7. 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
- Temporal operators
 - Adding and subtracting a *Duration* to or from a temporal instant
 - Adding and subtracting a *Duration* to or from another *Duration*
 - Multiplying and dividing a *Duration* with or by a number
- List operators
 - Concatenating two lists using **+**
 - Using **IN** to check if a number is in a list
 - Using **IN** for more complex list membership operations
 - Accessing elements in a list using the **[]** operator
- Property operators
- Equality and comparison of values
- Ordering and comparison of values
- Chaining comparison operations

6.7.1. 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
Temporal operators	+ and - for operations between durations and temporal instants/durations, * and / for operations between durations and numbers
List operators	+ for concatenation, IN to check existence of an element in a list, [] for accessing element(s)

6.7.2. 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] AS ps
UNWIND ps AS p
RETURN DISTINCT p.eyeColor
```

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

Table 6. 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 7. 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 8. 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.



The behavior of the `[]` operator with respect to `null` is detailed [here](#).

6.7.3. 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 9. 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 10. Result

result
7
1 row

6.7.4. 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 11. 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 12. 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.

6.7.5. 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 13. Result

number
4
7
9
3 rows

6.7.6. 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 14. 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.

6.7.7. Temporal operators

Temporal operators comprise:

- adding a *Duration* to either a temporal instant or another *Duration*: `+`
- subtracting a *Duration* from either a temporal instant or another *Duration*: `-`
- multiplying a *Duration* with a number: `*`
- dividing a *Duration* by a number: `/`

The following table shows — for each combination of operation and operand type — the type of the value returned from the application of each temporal operator:

Operator	Left-hand operand	Right-hand operand	Type of result
<code>+</code>	Temporal instant	<i>Duration</i>	The type of the temporal instant
<code>+</code>	<i>Duration</i>	Temporal instant	The type of the temporal instant
<code>-</code>	Temporal instant	<i>Duration</i>	The type of the temporal instant
<code>+</code>	<i>Duration</i>	<i>Duration</i>	<i>Duration</i>
<code>-</code>	<i>Duration</i>	<i>Duration</i>	<i>Duration</i>
<code>*</code>	<i>Duration</i>	<i>Number</i>	<i>Duration</i>
<code>*</code>	<i>Number</i>	<i>Duration</i>	<i>Duration</i>
<code>/</code>	<i>Duration</i>	<i>Number</i>	<i>Duration</i>

Adding and subtracting a *Duration* to or from a temporal instant

Query

```
WITH localdatetime({ year:1984, month:10, day:11, hour:12, minute:31, second:14 }) AS aDateTime,
duration({ years: 12, nanoseconds: 2 }) AS aDuration
RETURN aDateTime + aDuration, aDateTime - aDuration
```

Table 15. Result

aDateTime + aDuration	aDateTime - aDuration
1996-10-11T12:31:14.000000002	1972-10-11T12:31:13.999999998
1 row	

Components of a *Duration* that do not apply to the temporal instant are ignored. For example, when adding a *Duration* to a *Date*, the *hours*, *minutes*, *seconds* and *nanoseconds* of the *Duration* are ignored (*Time* behaves in an analogous manner):

Query

```
WITH date({ year:1984, month:10, day:11 }) AS aDate, duration({ years: 12, nanoseconds: 2 }) AS aDuration
RETURN aDate + aDuration, aDate - aDuration
```

Table 16. Result

aDate + aDuration	aDate - aDuration
1996-10-11	1972-10-11
1 row	

Adding two durations to a temporal instant is not an associative operation. This is because non-existing dates are truncated to the nearest existing date:

Query

```
RETURN (date("2011-01-31")+ duration("P1M"))+ duration("P12M") AS date1, date("2011-01-31")+(duration("P1M")+ duration("P12M")) AS date2
```

Table 17. Result

date1	date2
2012-02-28	2012-02-29
1 row	

Adding and subtracting a *Duration* to or from another *Duration*

Query

```
WITH duration({ years: 12, months: 5, days: 14, hours: 16, minutes: 12, seconds: 70, nanoseconds: 1 }) AS duration1, duration({ months:1, days: -14, hours: 16, minutes: -12, seconds: 70 }) AS duration2
RETURN duration1, duration2, duration1 + duration2, duration1 - duration2
```

Table 18. Result

duration1	duration2	duration1 + duration2	duration1 - duration2
P12Y5M14DT16H13M10.000000001S	P1M-14DT15H49M10S	P12Y6MT32H2M20.000000001S	P12Y4M28DT24M0.000000001S
1 row			

Multiplying and dividing a *Duration* with or by a number

These operations are interpreted simply as component-wise operations with overflow to smaller units based on an average length of units in the case of division (and multiplication with fractions).

Query

```
WITH duration({ days: 14, minutes: 12, seconds: 70, nanoseconds: 1 }) AS aDuration
RETURN aDuration, aDuration * 2, aDuration / 3
```

Table 19. Result

aDuration	aDuration * 2	aDuration / 3
P14DT13M10.000000001S	P28DT26M20.000000002S	P4DT16H4M23.333333333S
1 row		

6.7.8. List operators

List operators comprise:

- concatenating lists l_1 and l_2 : $[l_1] + [l_2]$
- checking if an element e exists in a list l : $e \text{ IN } [l]$
- accessing an element(s) in a list using the subscript operator: $[]$



The behavior of the `IN` and `[]` operators with respect to `null` is detailed [here](#).

Concatenating two lists using `+`

Query

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

Table 20. Result

myList
JavaListWrapper(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 21. Result

number
2
3
2 rows

Using `IN` for more complex list membership operations

The general rule is that the `IN` operator will evaluate to `true` if the list given as the right-hand operand contains an element which has the same *type and contents (or value)* as the left-hand operand. Lists are only comparable to other lists, and elements of a list l are compared pairwise in ascending order from the first element in l to the last element in l .

The following query checks whether or not the list $[2, 1]$ is an element of the list $[1, [2, 1], 3]$:

Query

```
RETURN [2, 1] IN [1,[2, 1], 3] AS inList
```

The query evaluates to `true` as the right-hand list contains, as an element, the list $[1, 2]$ which is of the same type (a list) and contains the same contents (the numbers 2 and 1 in the given order) as the

left-hand operand. If the left-hand operator had been [1, 2] instead of [2, 1], the query would have returned `false`.

Table 22. Result

inList
<code>true</code>
1 row

At first glance, the contents of the left-hand operand and the right-hand operand *appear* to be the same in the following query:

Query

```
RETURN [1, 2] IN [1, 2] AS inList
```

However, `IN` evaluates to `false` as the right-hand operand does not contain an element that is of the same *type* — i.e. a *list* — as the left-hand-operand.

Table 23. Result

inList
<code>false</code>
1 row

The following query can be used to ascertain whether or not a list l_{lhs} — obtained from, say, the `labels()` function — contains at least one element that is also present in another list l_{rhs} :

```
MATCH (n)
WHERE size([l IN labels(n) WHERE l IN ['Person', 'Employee'] | 1]) > 0
RETURN count(n)
```

As long as `labels(n)` returns either `Person` or `Employee` (or both), the query will return a value greater than zero.

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 24. Result

result
<code>JavaListWrapper(John, Bill)</code>
1 row

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

6.7.9. 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.

6.7.10. Equality and comparison of values

Equality

Cypher supports comparing values (see [Values and types](#)) 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.

6.7.11. 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`.
- **Comparison of spatial values:**
 - Point values can only be compared within the same Coordinate Reference System (CRS) — otherwise, the result will be `null`.
 - For two points `a` and `b` within the same CRS, `a` is considered to be greater than `b` if `a.x > b.x` and `a.y > b.y` (and `a.z > b.z` for 3D points).
 - `a` is considered less than `b` if `a.x < b.x` and `a.y < b.y` (and `a.z < b.z` for 3D points).
 - If none of the above is true, the points are considered incomparable and any comparison operator between them will return `null`.

- **Ordering** of spatial values:
 - `ORDER BY` requires all values to be orderable.
 - Points are ordered after arrays and before temporal types.
 - Points of different CRS are ordered by the CRS code (the value of SRID field). For the currently supported set of [Coordinate Reference Systems](#) this means the order: 4326, 4979, 7302, 9157
 - Points of the same CRS are ordered by each coordinate value in turn, `x` first, then `y` and finally `z`.
 - Note that this order is different to the order returned by the spatial index, which will be the order of the space filling curve.
- **Comparison** of temporal values:
 - [Temporal instant values](#) are comparable within the same type. An instant is considered less than another instant if it occurs before that instant in time, and it is considered greater than if it occurs after.
 - Instant values that occur at the same point in time — but that have a different time zone — are not considered equal, and must therefore be ordered in some predictable way. Cypher prescribes that, after the primary order of point in time, instant values be ordered by effective time zone offset, from west (negative offset from UTC) to east (positive offset from UTC). This has the effect that times that represent the same point in time will be ordered with the time with the earliest local time first. If two instant values represent the same point in time, and have the same time zone offset, but a different named time zone (this is possible for `DateTime` only, since `Time` only has an offset), these values are not considered equal, and ordered by the time zone identifier, alphabetically, as its third ordering component.
 - [Duration](#) values cannot be compared, since the length of a `day`, `month` or `year` is not known without knowing which `day`, `month` or `year` it is. Since `Duration` values are not comparable, the result of applying a comparison operator between two `Duration` values is `null`. If the type, point in time, offset, and time zone name are all equal, then the values are equal, and any difference in order is impossible to observe.
- **Ordering** of temporal values:
 - `ORDER BY` requires all values to be orderable.
 - Temporal instances are ordered after spatial instances and before strings.
 - Comparable values should be ordered in the same order as implied by their comparison order.
 - Temporal instant values are first ordered by type, and then by comparison order within the type.
 - Since no complete comparison order can be defined for `Duration` values, we define an order for `ORDER BY` specifically for `Duration`:
 - `Duration` values are ordered by normalising all components as if all years were 365.2425 days long (`PT8765H49M12S`), all months were 30.436875 (1/12 year) days long (`PT730H29M06S`), and all days were 24 hours long [1: The 365.2425 days per year comes from the frequency of leap years. A leap year occurs on a year with an ordinal number divisible by 4, that is not divisible by 100, unless it is divisible by 400. This means that over 400 years there are $((365 * 4 + 1) * 25 - 1) * 4 + 1 = 146097$ days, which means an average of 365.2425 days per year.].
 - Comparing for ordering when one argument is `null` (e.g. `null < 3` is `null`).

6.7.12. 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](#).

6.8. 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
```

6.9. 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

6.9.1. 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](#)

6.9.2. 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`.

6.9.3. 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)
```

6.9.4. 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)
```

6.9.5. 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.

6.9.6. 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)
```

6.9.7. 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), p = (a)-[]->(c) RETURN *, relationships(p) AS rs` 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 p = (a)-[]->(c) RETURN *, relationships(p)`. 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:

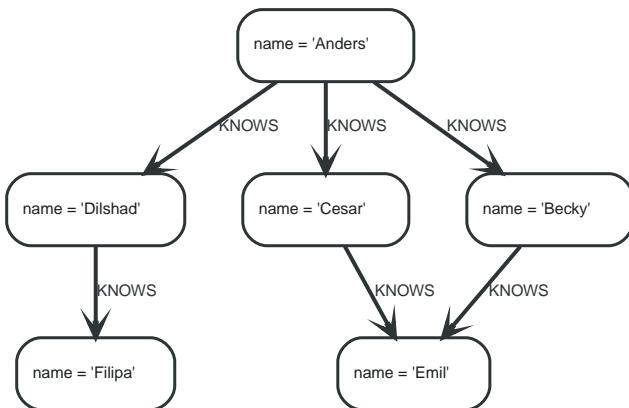


Figure 3. Graph

Query

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

Table 25. Result

remote_friend.name
"Dilshad"
"Anders"
2 rows

This query finds data in the graph which 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**.

6.9.8. 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.

6.10. Temporal (Date/Time) values

Cypher has built-in support for handling temporal values, and the underlying database supports storing these temporal values as properties on nodes and relationships.

- [Introduction](#)
- [Time zones](#)
- [Temporal instants](#)
 - [Specifying temporal instants](#)
 - [Specifying dates](#)
 - [Specifying times](#)
 - [Specifying time zones](#)
 - [Examples](#)
 - [Accessing components of temporal instants](#)
- [Durations](#)
 - [Specifying durations](#)
 - [Examples](#)
 - [Accessing components of durations](#)
- [Examples](#)
- [Temporal indexing](#)

Refer to [Temporal functions](#) for information regarding temporal *functions* allowing for the creation and manipulation of temporal values.



Refer to [Temporal operators](#) for information regarding temporal *operators*.

Refer to [Ordering and comparison of values](#) for information regarding the comparison and ordering of temporal values.

6.10.1. Introduction

The following table depicts the temporal value types and supported components:

Type	Date support	Time support	Time zone support
Date	X		
Time		X	X

Type	Date support	Time support	Time zone support
LocalTime		X	
DateTime	X	X	X
LocalDateTime	X	X	
Duration	-	-	-

Date, *Time*, *LocalTime*, *DateTime* and *LocalDateTime* are *temporal instant* types. A temporal instant value expresses a point in time with varying degrees of precision.

By contrast, *Duration* is not a temporal instant type. A *Duration* represents a temporal amount, capturing the difference in time between two instants, and can be negative. Duration only captures the amount of time between two instants, and thus does not encapsulate a start time and end time.

6.10.2. Time zones

Time zones are represented either as an offset from UTC, or as a logical identifier of a *named time zone* (these are based on the [IANA time zone database](https://www.iana.org/time-zones) (<https://www.iana.org/time-zones>)). In either case the time is stored as UTC internally, and the time zone offset is only applied when the time is presented. This means that temporal instants can be ordered without taking time zone into account. If, however, two times are identical in UTC, then they are ordered by timezone.

When creating a time using a named time zone, the offset from UTC is computed from the rules in the time zone database to create a time instant in UTC, and to ensure the named time zone is a valid one.

It is possible for time zone rules to change in the IANA time zone database. For example, there could be alterations to the rules for daylight savings time in a certain area. If this occurs after the creation of a temporal instant, the presented time could differ from the originally-entered time, insofar as the local timezone is concerned. However, the absolute time in UTC would remain the same.

There are three ways of specifying a time zone in Cypher:

- Specifying the offset from UTC in hours and minutes ([ISO 8601](https://en.wikipedia.org/wiki/ISO_8601) (https://en.wikipedia.org/wiki/ISO_8601))
- Specifying a named time zone
- Specifying both the offset and the time zone name (with the requirement that these match)

The named time zone form uses the rules of the IANA time zone database to manage *daylight savings time* (DST).

The default time zone of the database can be configured using the configuration option `db.temporal.timezone`. This configuration option influences the creation of temporal types for the following functions:

- Getting the current date and time without specifying a time zone.
- Creating a temporal type from its components without specifying a time zone.
- Creating a temporal type by parsing a string without specifying a time zone.
- Creating a temporal type by combining or selecting values that do not have a time zone component, and without specifying a time zone.
- Truncating a temporal value that does not have a time zone component, and without specifying a time zone.

6.10.3. Temporal instants

Specifying temporal instants

A temporal instant consists of three parts; the **date**, the **time**, and the **timezone**. These parts may then be combined to produce the various temporal value types. Literal characters are denoted in **bold**.

Temporal instant type	Composition of parts
<i>Date</i>	< date >
<i>Time</i>	< time >< timezone > or T < time >< timezone >
<i>LocalTime</i>	< time > or T < time >
<i>DateTime</i> *	< date > T < time >< timezone >
<i>LocalDateTime</i> *	< date > T < time >

*When **date** and **time** are combined, **date** must be complete; i.e. fully identify a particular day.

Specifying dates

Component	Format	Description
Year	YYYY	Specified with at least four digits (special rules apply in certain cases)
Month	MM	Specified with a double digit number from 01 to 12
Week	ww	Always prefixed with W and specified with a double digit number from 01 to 53
Quarter	q	Always prefixed with Q and specified with a single digit number from 1 to 4
Day of the month	DD	Specified with a double digit number from 01 to 31
Day of the week	D	Specified with a single digit number from 1 to 7
Day of the quarter	DD	Specified with a double digit number from 01 to 92
Ordinal day of the year	DDD	Specified with a triple digit number from 001 to 366

If the year is before 0000 or after 9999, the following additional rules apply:

- - must prefix any year before 0000
- + must prefix any year after 9999
- The year must be separated from the next component with the following characters:
 - - if the next component is month or day of the year
 - Either - or **W** if the next component is week of the year
 - **Q** if the next component is quarter of the year

If the year component is prefixed with either - or +, and is separated from the next component, **Year** is allowed to contain up to nine digits. Thus, the allowed range of years is between -999,999,999 and +999,999,999. For all other cases, i.e. the year is between 0000 and 9999 (inclusive), **Year** must have exactly four digits (the year component is interpreted as a year of the Common Era (CE)).

The following formats are supported for specifying dates:

Format	Description	Example	Interpretation of example
YYYY-MM-DD	Calendar date: Year-Month-Day	2015-07-21	2015-07-21
YYYYMMDD	Calendar date: Year-Month-Day	20150721	2015-07-21
YYYY-MM	Calendar date: Year-Month	2015-07	2015-07-01
YYYYMM	Calendar date: Year-Month	201507	2015-07-01
YYYY-Www-D	Week date: Year-Week-Day	2015-W30-2	2015-07-21
YYYYWwwD	Week date: Year-Week-Day	2015W302	2015-07-21
YYYY-Www	Week date: Year-Week	2015-W30	2015-07-20
YYYYWww	Week date: Year-Week	2015W30	2015-07-20
YYYY-Qq-DD	Quarter date: Year-Quarter-Day	2015-Q2-60	2015-05-30
YYYYQqDD	Quarter date: Year-Quarter-Day	2015Q260	2015-05-30
YYYY-Qq	Quarter date: Year-Quarter	2015-Q2	2015-04-01
YYYYQq	Quarter date: Year-Quarter	2015Q2	2015-04-01
YYYY-DDD	Ordinal date: Year-Day	2015-202	2015-07-21
YYYYDDD	Ordinal date: Year-Day	2015202	2015-07-21
YYYY	Year	2015	2015-01-01

The least significant components can be omitted. Cypher will assume omitted components to have their lowest possible value. For example, `2013-06` will be interpreted as being the same date as `2013-06-01`.

Specifying times

Component	Format	Description
Hour	HH	Specified with a double digit number from <code>00</code> to <code>23</code>
Minute	MM	Specified with a double digit number from <code>00</code> to <code>59</code>
Second	SS	Specified with a double digit number from <code>00</code> to <code>59</code>
fraction	ssssssss	Specified with a number from <code>0</code> to <code>99999999</code> . It is not required to specify trailing zeros. <code>fraction</code> is an optional, sub-second component of <code>Second</code> . This can be separated from <code>Second</code> using either a full stop (<code>.</code>) or a comma (<code>,</code>). The <code>fraction</code> is in addition to the two digits of <code>Second</code> .

Cypher does not support leap seconds; [UTC-SLS](https://www.cl.cam.ac.uk/~mgk25/time/utc-sls/) (<https://www.cl.cam.ac.uk/~mgk25/time/utc-sls/>) (*UTC with Smoothed Leap Seconds*) is used to manage the difference in time between UTC and TAI (*International Atomic Time*).

The following formats are supported for specifying times:

Format	Description	Example	Interpretation of example
HH:MM:SS.aaaaaaaaaa	Hour:Minute:Second.fraction	21:40:32.142	21:40:32.142
HHMMSS.aaaaaaaaaa	Hour:Minute:Second.fraction	214032.142	21:40:32.142
HH:MM:SS	Hour:Minute:Second	21:40:32	21:40:32.000
HHMMSS	Hour:Minute:Second	214032	21:40:32.000
HH:MM	Hour:Minute	21:40	21:40:00.000
HHMM	Hour:Minute	2140	21:40:00.000
HH	Hour	21	21:00:00.000

The least significant components can be omitted. For example, a time may be specified with **Hour** and **Minute**, leaving out **Second** and **fraction**. On the other hand, specifying a time with **Hour** and **Second**, while leaving out **Minute**, is not possible.

Specifying time zones

The time zone is specified in one of the following ways:

- As an offset from UTC
- Using the **Z** shorthand for the UTC (**+00:00**) time zone

When specifying a time zone as an offset from UTC, the rules below apply:

- The time zone always starts with either a plus (+) or minus (-) sign.
 - Positive offsets, i.e. time zones beginning with +, denote time zones east of UTC.
 - Negative offsets, i.e. time zones beginning with -, denote time zones west of UTC.
- A double-digit hour offset follows the +/- sign.
- An optional double-digit minute offset follows the hour offset, optionally separated by a colon (:).
- The time zone of the International Date Line is denoted either by **+12:00** or **-12:00**, depending on country.

When creating values of the *DateTime* temporal instant type, the time zone may also be specified using a named time zone, using the names from the IANA time zone database. This may be provided either in addition to, or in place of the offset. The named time zone is given last and is enclosed in square brackets ([]). Should both the offset and the named time zone be provided, the offset must match the named time zone.

The following formats are supported for specifying time zones:

Format	Description	Example	Supported for <i>DateTime</i>	Supported for <i>Time</i>
Z	UTC	Z	X	X
+HH:MM	Hour:Minute	+09:30	X	X
+HH:MM[ZoneName]	Hour:Minute[ZoneName]	+08:45[Australia/Eucla]	X	
+HHMM	Hour:Minute	+0100	X	X

Format	Description	Example	Supported for DateTime	Supported for Time
<code>±HHMM[ZoneName]</code>	Hour:Minute[ZoneName]	<code>+0200[Africa/Johannesburg]</code>	X	
<code>±HH</code>	Hour	<code>-08</code>	X	X
<code>±HH[ZoneName]</code>	Hour[ZoneName]	<code>+08[Asia/Singapore]</code>	X	
<code>[ZoneName]</code>	[ZoneName]	<code>[America/Regina]</code>	X	

Examples

We show below examples of parsing temporal instant values using various formats. For more details, refer to [An overview of temporal instant type creation](#).

Parsing a *DateTime* using the *calendar date* format:

Query

```
RETURN datetime('2015-06-24T12:50:35.556+0100') AS theDateTime
```

Table 26. Result

theDateTime
<code>2015-06-24T12:50:35.556+01:00</code>
1 row

Parsing a *LocalDateTime* using the *ordinal date* format:

Query

```
RETURN localdatetime('2015185T19:32:24') AS theLocalDateTime
```

Table 27. Result

theLocalDateTime
<code>2015-07-04T19:32:24</code>
1 row

Parsing a *Date* using the *week date* format:

Query

```
RETURN date('+2015-W13-4') AS theDate
```

Table 28. Result

theDate
<code>2015-03-26</code>
1 row

Parsing a *Time*:

Query

```
RETURN time('125035.556+0100') AS theTime
```

Table 29. Result

theTime
12:50:35.556+01:00
1 row

Parsing a *LocalTime*:

Query

```
RETURN localtime('12:50:35.556') AS theLocalTime
```

Table 30. Result

theLocalTime
12:50:35.556
1 row

Accessing components of temporal instants

Components of temporal instant values can be accessed as properties.

Table 31. Components of temporal instant values and where they are supported

Component	Description	Type	Range/Format	Date	DateTime	LocalDateTime	Time	LocalTime
instant.year	The year component represents the astronomical year number (https://en.wikipedia.org/wiki/Astronomical_year_number) of the instant [2]: This is in accordance with the Gregorian calendar (https://en.wikipedia.org/wiki/Gregorian_calendar); i.e. years AD/CE start at year 1, and the year before that (year 1 BC/BCE) is 0, while year 2 BCE is -1 etc.]	Integer	At least 4 digits. For more information, see the rules for using the Year component	X	X	X		
instant.quarter	The quarter-of-the-year component	Integer	1 to 4	X	X	X		

Component	Description	Type	Range/Format	Date	DateTime	LocalDateTime	Time	LocalTime
instant.month	The month-of-the-year component	Integer	1 to 12	X	X	X		
instant.week	The week-of-the-year component [3: The first week of any year (https://en.wikipedia.org/wiki/ISO_week_date#First_week) is the week that contains the first Thursday of the year, and thus always contains January 4.]	Integer	1 to 53	X	X	X		
instant.weekYear	The year that the week-of-year component belongs to [4: For dates from December 29, this could be the next year, and for dates until January 3 this could be the previous year, depending on how week 1 begins.]	Integer	At least 4 digits. For more information, see the rules for using the Year component	X	X	X		
instant.dayOfQuarter	The day-of-the-quarter component	Integer	1 to 92	X	X	X		
instant.day	The day-of-the-month component	Integer	1 to 31	X	X	X		
instant.ordinalDay	The day-of-the-year component	Integer	1 to 366	X	X	X		
instant.dayOfWeek	The day-of-the-week component (the first day of the week is Monday)	Integer	1 to 7	X	X	X		
instant.hour	The hour component	Integer	0 to 23		X	X	X	X
instant.minute	The minute component	Integer	0 to 59		X	X	X	X

Component	Description	Type	Range/Format	Date	DateTime	LocalDateTime	Time	LocalTime
instant.second	The <i>second</i> component	Integer	0 to 60		X	X	X	X
instant.millisecond	The <i>millisecond</i> component	Integer	0 to 999		X	X	X	X
instant.microsecond	The <i>microsecond</i> component	Integer	0 to 999999		X	X	X	X
instant.nanosecond	The <i>nanosecond</i> component	Integer	0 to 999999999		X	X	X	X
instant.timezone	The <i>timezone</i> component	String	Depending on how the time zone was specified , this is either a time zone name or an offset from UTC in the format <code>±HHMM</code>		X		X	
instant.offset	The <i>timezone</i> offset	String	<code>±HHMM</code>		X		X	
instant.offsetMinutes	The <i>timezone</i> offset in minutes	Integer	-1080 to +1080		X		X	
instant.offsetSeconds	The <i>timezone</i> offset in seconds	Integer	-64800 to +64800		X		X	
instant.epochMillis	The number of milliseconds between <code>1970-01-01T00:00:00+0000</code> and the instant [5: <code>datetime().epochMillis</code> returns the equivalent value of the <code>timestamp()</code> function.]	Integer	Positive for instants after and negative for instants before <code>1970-01-01T00:00:00+0000</code>		X			

Component	Description	Type	Range/Format	Date	DateTime	LocalDateTime	Time	LocalTime
instant.epochSeconds	The number of seconds between 1970-01-01T00:00:00+0000 and the instant [6: For the nanosecond part of the epoch offset, the regular nanosecond component (<code>instant.nanosecond</code>) can be used.]	Integer	Positive for instants after and negative for instants before 1970-01-01T00:00:00+0000		X			

The following query shows how to extract the components of a `Date` value:

Query

```
WITH date({ year:1984, month:10, day:11 }) AS d
RETURN d.year, d.quarter, d.month, d.week, d.weekYear, d.day, d.ordinalDay, d.dayOfWeek, d.dayOfQuarter
```

Table 32. Result

d.year	d.quarter	d.month	d.week	d.weekYear	d.day	d.ordinalDay	d.dayOfWeek	d.dayOfQuarter
1984	4	10	41	1984	11	285	4	11
1 row								

The following query shows how to extract the components of a `DateTime` value:

Query

```
WITH datetime({ year:1984, month:11, day:11, hour:12, minute:31, second:14, nanosecond: 645876123,
timezone:'Europe/Stockholm' }) AS d
RETURN d.year, d.quarter, d.month, d.week, d.weekYear, d.day, d.ordinalDay, d.dayOfWeek, d.dayOfQuarter,
d.hour, d.minute, d.second, d.millisecond, d.microsecond, d.nanosecond, d.timezone, d.offset,
d.offsetMinutes, d.epochSeconds, d.epochMillis
```

Table 33. Result

d.year	d.quarter	d.month	d.week	d.weekYear	d.day	d.ordinalDay	d.dayOfWeek	d.dayOfQuarter	d.hour	d.minute	d.second	d.millisecond	d.microsecond	d.nanosecond	d.timezone	d.offset	d.offsetMinutes	d.epochSeconds	d.epochMillis
1984	4	11	45	1984	11	316	7	42	12	31	14	645	645876	645876123	"Europe/Stockholm"	"+01:00"	60	469020674	469020674645
1 row																			

6.10.4. Durations

Specifying durations

A *Duration* represents a temporal amount, capturing the difference in time between two instants, and can be negative.

The specification of a *Duration* is prefixed with a **P**, and can use either a *unit-based form* or a *date-and-time-based form*:

- Unit-based form: **P[nY][nM][nW][nD][T[nH][nM][nS]]**
 - The square brackets (**[]**) denote an optional component (components with a zero value may be omitted).
 - The **n** denotes a numeric value which can be arbitrarily large.
 - The value of the last — and least significant — component may contain a decimal fraction.
 - Each component must be suffixed by a component identifier denoting the unit.
 - The unit-based form uses **M** as a suffix for both months and minutes. Therefore, time parts must always be preceded with **T**, even when no components of the date part are given.
- Date-and-time-based form: **P<date>T<time>**
 - Unlike the unit-based form, this form requires each component to be within the bounds of a valid *LocalDateTime*.

The following table lists the component identifiers for the unit-based form:

Component identifier	Description	Comments
Y	Years	
M	Months	Must be specified before T
W	Weeks	
D	Days	
H	Hours	
M	Minutes	Must be specified after T
S	Seconds	

Examples

The following examples demonstrate various methods of parsing *Duration* values. For more details, refer to [Creating a Duration from a string](#).

Return a *Duration* of **14 days, 16 hours and 12 minutes**:

Query

```
RETURN duration('P14DT16H12M') AS theDuration
```

Table 34. Result

theDuration
P14DT16H12M
1 row

Return a *Duration* of **5 months, 1 day and 12 hours**:

Query

```
RETURN duration('P5M1.5D') AS theDuration
```

Table 35. Result

theDuration
P5M1DT12H
1 row

Return a *Duration* of 45 seconds:

Query

```
RETURN duration('PT0.75M') AS theDuration
```

Table 36. Result

theDuration
PT45S
1 row

Return a *Duration* of 2 weeks, 3 days and 12 hours:

Query

```
RETURN duration('P2.5W') AS theDuration
```

Table 37. Result

theDuration
P17DT12H
1 row

Accessing components of durations

A *Duration* can have several components. These are categorized into the following groups:

Component group	Constituent components
Months	Years, Quarters and Months
Days	Weeks and Days
Seconds	Hours, Minutes, Seconds, Milliseconds, Microseconds and Nanoseconds

Within each group, the components can be converted without any loss:

- There are always 4 quarters in 1 year.
- There are always 12 months in 1 year.
- There are always 3 months in 1 quarter.
- There are always 7 days in 1 week.
- There are always 60 minutes in 1 hour.

- There are always **60 seconds** in **1 minute** (Cypher uses **UTC-SLS** (<https://www.cl.cam.ac.uk/~mgk25/time/utc-sls/>) when handling leap seconds).
- There are always **1000 milliseconds** in **1 second**.
- There are always **1000 microseconds** in **1 millisecond**.
- There are always **1000 nanoseconds** in **1 microsecond**.

Please note that:

- There are not always **24 hours** in **1 day**; when switching to/from daylight savings time, a **day** can have **23** or **25 hours**.
- There are not always the same number of **days** in a **month**.
- Due to leap years, there are not always the same number of **days** in a **year**.

Table 38. Components of Duration values and how they are truncated within their component group

Component	Component Group	Description	Type	Details
duration.years	Months	The total number of <i>years</i>	Integer	Each set of 4 quarters is counted as 1 year ; each set of 12 months is counted as 1 year .
duration.months	Months	The total number of <i>months</i>	Integer	Each year is counted as 12 months ; each quarter is counted as 3 months .
duration.days	Days	The total number of <i>days</i>	Integer	Each week is counted as 7 days .
duration.hours	Seconds	The total number of <i>hours</i>	Integer	Each set of 60 minutes is counted as 1 hour ; each set of 3600 seconds is counted as 1 hour .
duration.minutes	Seconds	The total number of <i>minutes</i>	Integer	Each hour is counted as 60 minutes ; each set of 60 seconds is counted as 1 minute .
duration.seconds	Seconds	The total number of <i>seconds</i>	Integer	Each hour is counted as 3600 seconds ; each minute is counted as 60 seconds .
duration.milliseconds	Seconds	The total number of <i>milliseconds</i>	Integer	
duration.microseconds	Seconds	The total number of <i>microseconds</i>	Integer	
duration.nanoseconds	Seconds	The total number of <i>nanoseconds</i>	Integer	

It is also possible to access the smaller (less significant) components of a component group bounded by the largest (most significant) component of the group:

Component	Component Group	Description	Type
duration.monthsOfYear	Months	The number of <i>months</i> in the group that do not make a whole year	Integer
duration.minutesOfHour	Seconds	The total number of <i>minutes</i> in the group that do not make a whole hour	Integer

Component	Component Group	Description	Type
duration.secondsOfMinute	Seconds	The total number of <i>seconds</i> in the group that do not make a whole <i>minute</i>	Integer
duration.millisecondsOfSecond	Seconds	The total number of <i>milliseconds</i> in the group that do not make a whole <i>second</i>	Integer
duration.microsecondsOfSecond	Seconds	The total number of <i>microseconds</i> in the group that do not make a whole <i>second</i>	Integer
duration.nanosecondsOfSecond	Seconds	The total number of <i>nanoseconds</i> in the group that do not make a whole <i>second</i>	Integer

The following query shows how to extract the components of a *Duration* value:

Query

```
WITH duration({ years: 1, months:4, days: 111, hours: 1, minutes: 1, seconds: 1, nanoseconds: 111111111 }) AS d
RETURN d.years, d.months, d.monthsOfYear, d.days, d.hours, d.minutes, d.minutesOfHour, d.seconds,
d.secondsOfMinute, d.milliseconds, d.millisecondsOfSecond, d.microseconds, d.microsecondsOfSecond,
d.nanoseconds, d.nanosecondsOfSecond
```

Table 39. Result

d.years	d.months	d.monthsOfYear	d.days	d.hours	d.minutes	d.minutesOfHour	d.seconds	d.secondsOfMinute	d.nanoseconds	d.nanosecondsOfSecond	d.nanosecondsOfSecond	d.nanosecondsOfSecond	d.nanosecondsOfSecond
1	16	4	111	1	61	1	3661	1	3661111	111	366111111	1111111	36611111111
1 row													

6.10.5. Examples

The following examples illustrate the use of some of the temporal functions and operators. Refer to [Temporal functions](#) and [Temporal operators](#) for more details.

Create a *Duration* representing 1.5 days:

Query

```
RETURN duration({ days: 1, hours: 12 }) AS theDuration
```

Table 40. Result

theDuration
P1DT12H
1 row

Compute the *Duration* between two temporal instants:

Query

```
RETURN duration.between(date('1984-10-11'), date('2015-06-24')) AS theDuration
```

Table 41. Result

theDuration
P30Y8M13D
1 row

Compute the number of days between two *Date* values:

Query

```
RETURN duration.inDays(date('2014-10-11'), date('2015-08-06')) AS theDuration
```

Table 42. Result

theDuration
P299D
1 row

Get the *Date* of Thursday in the current week:

Query

```
RETURN date.truncate('week', date(), { dayOfWeek: 4 }) AS thursday
```

Table 43. Result

thursday
2018-06-21
1 row

Get the *Date* of the last day of the next month:

Query

```
RETURN date.truncate('month', date() + duration('P2M')) - duration('P1D') AS lastDay
```

Table 44. Result

lastDay
2018-07-31
1 row

Add a *Duration* to a *Date*:

Query

```
RETURN time('13:42:19') + duration({ days: 1, hours: 12 }) AS theTime
```

Table 45. Result

theTime

| 01:42:19Z |
| 1 row |

Add two *Duration* values:

Query

```
RETURN duration({ days: 2, hours: 7 })+ duration({ months: 1, hours: 18 }) AS theDuration
```

Table 46. Result

theDuration

| P1M2DT25H |
| 1 row |

Multiply a *Duration* by a number:

Query

```
RETURN duration({ hours: 5, minutes: 21 })* 14 AS theDuration
```

Table 47. Result

theDuration

| PT74H54M |
| 1 row |

Divide a *Duration* by a number:

Query

```
RETURN duration({ hours: 3, minutes: 16 })/ 2 AS theDuration
```

Table 48. Result

theDuration

| PT1H38M |
| 1 row |

Examine whether two instants are less than one day apart:

Query

```
WITH datetime('2015-07-21T21:40:32.142+0100') AS date1, datetime('2015-07-21T17:12:56.333+0100') AS date2
RETURN
CASE
WHEN date1 < date2
THEN date1 + duration("P1D")> date2
ELSE date2 + duration("P1D")> date1 END AS lessThanOneDayApart
```

Table 49. Result

lessThanOneDayApart

| true |

lessThanOneDayApart

1 row

Return the abbreviated name of the current month:

Query

```
RETURN ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"] [date().month-1] AS month
```

Table 50. Result

month

"Jun"

1 row

6.10.6. Temporal indexing

All temporal types can be indexed, and thereby support exact lookups for equality predicates. Indexes for temporal instant types additionally support range lookups.

6.11. Lists

Cypher has comprehensive support for lists.

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



Information regarding operators such as list concatenation (`+`), element existence checking (`IN`) and access (`[]`) can be found [here](#). The behavior of the `IN` and `[]` operators with respect to `null` is detailed [here](#).

6.11.1. 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 51. Result

list

JavaListWrapper(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 52. 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 53. 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 54. Result

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

Query

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

Table 55. Result

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

Query

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

Table 56. Result

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

Query

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

Table 57. Result

range(0, 10)[..4]
JavaListWrapper(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 58. Result

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

Query

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

Table 59. Result

range(0, 10)[5..15]
JavaListWrapper(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 60. Result

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

6.11.2. 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 61. Result

result
JavaListWrapper(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 62. Result

result
JavaListWrapper(0, 2, 4, 6, 8, 10)
1 row

Query

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

Table 63. Result

result
JavaListWrapper(0.0, 1.0, 8.0, 27.0, 64.0, 125.0, 216.0, 343.0, 512.0, 729.0, 1000.0)
1 row

6.11.3. 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:

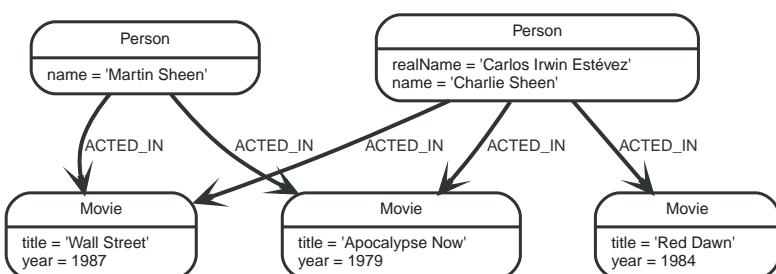


Figure 4. Graph

Query

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

Table 64. Result

years
JavaListWrapper(1979, 1984, 1987)
1 row

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

6.12. Maps

Cypher has solid support for maps.

- Literal maps
- Map projection
- Examples of map projection

The following graph is used for the examples below:

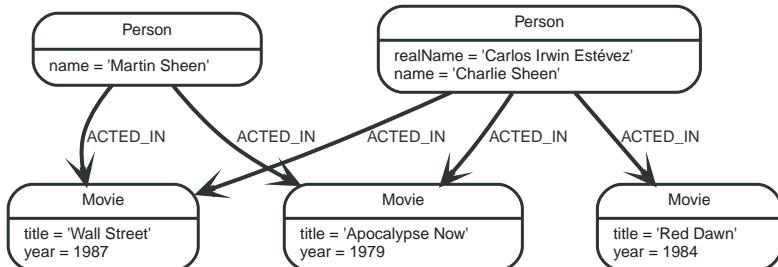


Figure 5. Graph



Information regarding property access operators such as `.` and `[]` can be found [here](#). The behavior of the `[]` operator with respect to `null` is detailed [here](#).

6.12.1. 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 65. Result

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

6.12.2. 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 66. Result

actor
<pre>{movies -> JavaListWrapper(Map(year -> 1979, title -> Apocalypse Now), Map(year -> 1984, title -> Red Dawn), Map(year -> 1987, title -> Wall Street)), realName -> "Carlos Irwin Estévez", name -> "Charlie Sheen"}</pre>
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 67. Result

actor
<pre>{nrOfMovies -> 2, name -> "Martin Sheen"}</pre>
<pre>{nrOfMovies -> 3, name -> "Charlie Sheen"}</pre>
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 68. Result

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

6.13. Spatial values

Cypher has built-in support for handling spatial values (points), and the underlying database supports storing these point values as properties on nodes and relationships.

- [Introduction](#)
- [Coordinate Reference Systems](#)
 - [Geographic coordinate reference systems](#)
 - [Cartesian coordinate reference systems](#)
- [Spatial instants](#)
 - [Creating points](#)
 - [Accessing components of points](#)
- [Spatial index](#)



Refer to [Spatial functions](#) for information regarding spatial *functions* allowing for the creation and manipulation of spatial values.

Refer to [Ordering and comparison of values](#) for information regarding the comparison and ordering of spatial values.

6.13.1. Introduction

Neo4j supports only one type of spatial geometry, the *Point* with the following characteristics:

- Each point can have either 2 or 3 dimensions. This means it contains either 2 or 3 64-bit floating point values, which together are called the *Coordinate*.
- Each point will also be associated with a specific [Coordinate Reference System](#) (CRS) that determines the meaning of the values in the *Coordinate*.
- Instances of *Point* and lists of *Point* can be assigned to node and relationship properties.
- Nodes with *Point* or *List(Point)* properties can be indexed using a spatial index. This is true for all CRS (and for both 2D and 3D). There is no special syntax for creating spatial indexes, as it is supported using the existing [schema indexes](#).
- The [distance function](#) will work on points in all CRS and in both 2D and 3D but only if the two points have the same CRS (and therefore also same dimension).

6.13.2. Coordinate Reference Systems

Four Coordinate Reference Systems (CRS) are supported, each of which falls within one of two types: *geographic coordinates* modeling points on the earth, or *cartesian coordinates* modeling points in euclidean space:

- [Geographic coordinate reference systems](#)
 - WGS-84: longitude, latitude (x, y)
 - WGS-84-3D: longitude, latitude, height (x, y, z)
- [Cartesian coordinate reference systems](#)
 - Cartesian: x, y
 - Cartesian 3D: x, y, z

Data within different coordinate systems are entirely incomparable, and cannot be implicitly converted from one to the other. This is true even if they are both cartesian or both geographic. For example, if you search for 3D points using a 2D range, you will get no results. However, they can be ordered, as discussed in more detail in the section on [Cypher ordering](#).

Geographic coordinate reference systems

Two Geographic Coordinate Reference Systems (CRS) are supported, modeling points on the earth:

- [WGS 84 2D](#) (<http://spatialreference.org/ref/epsg/4326/>)
 - A 2D geographic point in the *WGS 84* CRS is specified in one of two ways:
 - `longitude` and `latitude` (if these are specified, and the `crs` is not, then the `crs` is assumed to be `WGS-84`)
 - `x` and `y` (in this case the `crs` must be specified, or will be assumed to be `Cartesian`)
 - Specifying this CRS can be done using either the name 'wgs-84' or the SRID 4326 as described in [Point\(WGS-84\)](#)
- [WGS 84 3D](#) (<http://spatialreference.org/ref/epsg/4979/>)
 - A 3D geographic point in the *WGS 84* CRS is specified one of in two ways:
 - `longitude`, `latitude` and either `height` or `z` (if these are specified, and the `crs` is not, then the `crs` is assumed to be `WGS-84-3D`)
 - `x`, `y` and `z` (in this case the `crs` must be specified, or will be assumed to be `Cartesian-3D`)
 - Specifying this CRS can be done using either the name 'wgs-84-3d' or the SRID 4979 as described in [Point\(WGS-84-3D\)](#)

The units of the `latitude` and `longitude` fields are in decimal degrees, and need to be specified as floating point numbers using Cypher literals. It is not possible to use any other format, like 'degrees, minutes, seconds'. The units of the `height` field are in meters. When geographic points are passed to the `distance` function, the result will always be in meters. If the coordinates are in any other format or unit than supported, it is necessary to explicitly convert them. For example, if the incoming `$height` is a string field in kilometers, you would need to type `height:toFloat($height) * 1000`. Likewise if the results of the `distance` function are expected to be returned in kilometers, an explicit conversion is required. For example: `RETURN distance(a,b) / 1000 AS km`. An example demonstrating conversion on incoming and outgoing values is:

Query

```
WITH point({ latitude:toFloat('13.43'), longitude:toFloat('56.21')}) AS p1, point({  
    latitude:toFloat('13.10'), longitude:toFloat('56.41')}) AS p2  
RETURN.toInt(distance(p1,p2)/1000) AS km
```

Table 69. Result

km
42
1 row

Cartesian coordinate reference systems

Two Cartesian Coordinate Reference Systems (CRS) are supported, modeling points in euclidean space:

- [Cartesian 2D](http://spatialreference.org/ref/sr-org/7203/) (<http://spatialreference.org/ref/sr-org/7203/>)
 - A 2D point in the *Cartesian* CRS is specified with a map containing `x` and `y` coordinate values
 - Specifying this CRS can be done using either the name 'cartesian' or the SRID 7203 as described in [Point\(Cartesian\)](#)
- [Cartesian 3D](http://spatialreference.org/ref/sr-org/9157/) (<http://spatialreference.org/ref/sr-org/9157/>)
 - A 3D point in the *Cartesian* CRS is specified with a map containing `x`, `y` and `z` coordinate values
 - Specifying this CRS can be done using either the name 'cartesian-3d' or the SRID 9157 as described in [Point\(Cartesian-3D\)](#)

The units of the `x`, `y` and `z` fields are unspecified and can mean anything the user intends them to mean. This also means that when two cartesian points are passed to the `distance` function, the resulting value will be in the same units as the original coordinates. This is true for both 2D and 3D points, as the *pythagoras* equation used is generalized to any number of dimensions. However, just as you cannot compare geographic points to cartesian points, you cannot calculate the distance between a 2D point and a 3D point. If you need to do that, explicitly transform the one type into the other. For example:

Query

```
WITH point({ x:3, y:0 }) AS p2d, point({ x:0, y:4, z:1 }) AS p3d  
RETURN distance(p2d,p3d) AS bad, distance(p2d,point({ x:p3d.x, y:p3d.y })) AS good
```

Table 70. Result

bad	good
<null>	5.0
1 row	

6.13.3. Spatial instants

Creating points

All point types are created from two components:

- The *Coordinate* containing either 2 or 3 floating point values (64-bit)

- The Coordinate Reference System (or CRS) defining the meaning (and possibly units) of the values in the *Coordinate*

For most use cases it is not necessary to specify the CRS explicitly as it will be deduced from the keys used to specify the coordinate. Two rules are applied to deduce the CRS from the coordinate:

- Choice of keys:
 - If the coordinate is specified using the keys `latitude` and `longitude` the CRS will be assumed to be *Geographic* and therefore either `WGS-84` or `WGS-84-3D`.
 - If instead `x` and `y` are used, then the default CRS would be `Cartesian` or `Cartesian-3D`
- Number of dimensions:
 - If there are 2 dimensions in the coordinate, `x` & `y` or `longitude` & `latitude` the CRS will be a 2D CRS
 - If there is a third dimension in the coordinate, `z` or `height` the CRS will be a 3D CRS

All fields are provided to the `point` function in the form of a map of explicitly named arguments. We specifically do not support an ordered list of coordinate fields because of the contradictory conventions between geographic and cartesian coordinates, where geographic coordinates normally list `y` before `x` (`latitude` before `longitude`). See for example the following query which returns points created in each of the four supported CRS. Take particular note of the order and keys of the coordinates in the original `point` function calls, and how those values are displayed in the results:

Query

```
RETURN point({ x:3, y:0 }) AS cartesian_2d, point({ x:0, y:4, z:1 }) AS cartesian_3d, point({ latitude: 12, longitude: 56 }) AS geo_2d, point({ latitude: 12, longitude: 56, height: 1000 }) AS geo_3d
```

Table 71. Result

cartesian_2d	cartesian_3d	geo_2d	geo_3d
<code>point({x: 3.0, y: 0.0, crs: 'cartesian'})</code>	<code>point({x: 0.0, y: 4.0, z: 1.0, crs: 'cartesian-3d'})</code>	<code>point({x: 56.0, y: 12.0, crs: 'wgs-84'})</code>	<code>point({x: 56.0, y: 12.0, z: 1000.0, crs: 'wgs-84-3d'})</code>
1 row			

Accessing components of points

Just as we construct points using a map syntax, we can also access components as properties of the instance.

Table 72. Components of point instances and where they are supported

Component	Description	Type	Range/Form at	WGS-84	WGS-84-3D	Cartesian	Cartesian-3D
<code>instant.x</code>	The first element of the <i>Coordinate</i>	Float	Number literal, range depends on CRS	X	X	X	X
<code>instant.y</code>	The second element of the <i>Coordinate</i>	Float	Number literal, range depends on CRS	X	X	X	X
<code>instant.z</code>	The third element of the <i>Coordinate</i>	Float	Number literal, range depends on CRS		X		X

Component	Description	Type	Range/Form at	WGS-84	WGS-84-3D	Cartesian	Cartesian-3D
instant.latitude	The <i>second</i> element of the <i>Coordinate</i> for geographic CRS, degrees North of the equator	Float	Number literal, -90.0 to 90.0	X	X		
instant.longitude	The <i>first</i> element of the <i>Coordinate</i> for geographic CRS, degrees East of the prime meridian	Float	Number literal, -180.0 to 180.0	X	X		
instant.height	The third element of the <i>Coordinate</i> for geographic CRS, meters above the ellipsoid defined by the datum (WGS-84)	Float	Number literal, range limited only by the underlying 64-bit floating point type		X		
instant.crs	The name of the CRS	String	One of wgs-84, wgs-84-3d, cartesian, cartesian-3d	X	X	X	X
instant.srid	The internal Neo4j ID for the CRS	Integer	One of 4326, 4979, 7203, 9157	X	X	X	X

The following query shows how to extract the components of a *Cartesian 2D* point value:

Query

```
WITH point({ x:3, y:4 }) AS p
RETURN p.x, p.y, p.crs, p.srid
```

Table 73. Result

p.x	p.y	p.crs	p.srid
3.0	4.0	"cartesian"	7203
1 row			

The following query shows how to extract the components of a *WGS-84 3D* point value:

Query

```
WITH point({ latitude:3, longitude:4, height: 4321 }) AS p
RETURN p.latitude, p.longitude, p.height, p.x, p.y, p.z, p.crs, p.srid
```

Table 74. Result

p.latitude	p.longitude	p.height	p.x	p.y	p.z	p.crs	p.srid
3.0	4.0	4321.0	4.0	3.0	4321.0	"wgs-84-3d"	4979
1 row							

6.13.4. Spatial index

If there is a [schema index](#) on a particular `:Label(property)` combination, and a spatial point is assigned to that property on a node with that label, the node will be indexed in a spatial index. For spatial indexing, Neo4j uses space filling curves in 2D or 3D over an underlying generalized B+Tree. Points will be stored in up to four different trees, one for each of the [four coordinate reference systems](#). This allows for both [equality](#) and [range](#) queries using exactly the same syntax and behaviour as for other property types. In addition, queries using the [distance](#) function can, under the right conditions, also use the index, as described in the section '[Use index when executing a spatial distance search](#)'

6.14. Working with `null`

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

6.14.1. 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`.

6.14.2. 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

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

6.14.3. 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.

6.14.4. The [] operator and null

Accessing a list or a map with `null` will result in `null`:

Expression	Result
<code>[1, 2, 3][null]</code>	<code>null</code>
<code>[1, 2, 3, 4][null..2]</code>	<code>null</code>
<code>[1, 2, 3][1..null]</code>	<code>null</code>
<code>{age: 25}[null]</code>	<code>null</code>

Using parameters to pass in the bounds, such as `a[$lower..$upper]`, may result in a `null` for the lower or upper bound (or both). The following workaround will prevent this from happening by setting the absolute minimum and maximum bound values:

```
a[coalesce($lower,0)..coalesce($upper,size(a))]
```

6.14.5. Expressions that return null

- Getting a missing element from a list: `[][][0]`, `head([])`
- Trying to access a property that does not exist on a node or relationship: `n.missingProperty`
- Comparisons when either side is `null`: `1 < null`
- Arithmetic expressions containing `null`: `1 + null`
- Function calls where any arguments are `null`: `sin(null)`

Chapter 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. The returned expressions may all be aliased using `AS`.

Clause	Description
RETURN ... [AS]	Defines what to include in the query result set.
WITH ... [AS]	Allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.
UNWIND ... [AS]	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 <code>MATCH</code> or <code>OPTIONAL MATCH</code> clause or filters the results of a <code>WITH</code> clause.
ORDER BY [ASC ENDING] DESC[ENDING]]	A sub-clause following <code>RETURN</code> or <code>WITH</code> , specifying that the output should be sorted in either ascending (the default) or descending order.

Sub-clause	Description
SKIP	Defines from which row to start including the rows in the output.
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 <code>LOAD CSV</code> .

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 having 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:

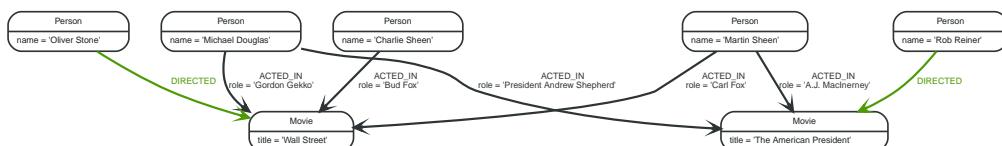


Figure 6. Graph

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 75. 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 76. 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 77. 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 78. 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 79. 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 80. 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 81. 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 82. 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 83. 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:

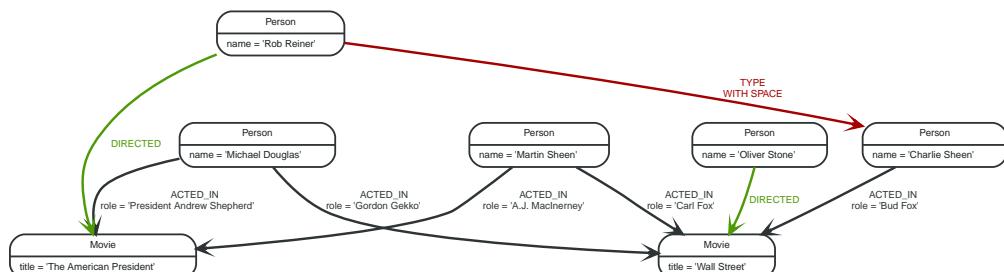


Figure 7. Graph

Query

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

Returns a relationship type with a space in it

Table 84. Result

```
type(r)
```

```
"TYPE WITH SPACE"
```

```
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 85. 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 86. 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, the list of relationships comprising the connection can be returned using the following syntax:

Query

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

Returns a list of relationships.

Table 87. Result

relationships(p)

```
JavaListWrapper((0)-[ACTED_IN,0]->(5), (1)-[ACTED_IN,1]->(5))  
JavaListWrapper((0)-[ACTED_IN,0]->(5), (2)-[ACTED_IN,2]->(5))
```

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:

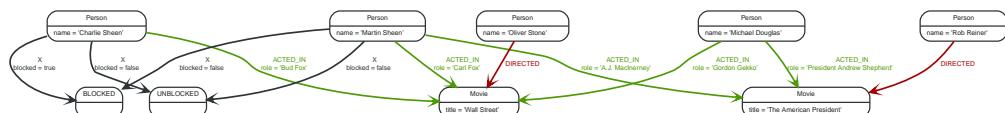


Figure 8. Graph

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 88. Result

p
(0)-[X,20]->(20)<-[X,21]-(1)
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 89. 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 90. Result

p
(2)-[ACTED_IN,5]->(6)
(2)-[ACTED_IN,2]->(5)
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 91. 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 92. Result

p
(1)-[ACTED_IN,1]->(5)<-[DIRECTED,3]-(3)
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 93. Result

p
(0)-[ACTED_IN,0]->(5)<-[ACTED_IN,1]-(1)
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 94. Result

p
(1)-[ACTED_IN,1]->(5)<-[ACTED_IN,2]-(2)
(1)-[ACTED_IN,4]->(6)<-[ACTED_IN,5]-(2)
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 95. 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 96. 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 97. 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:

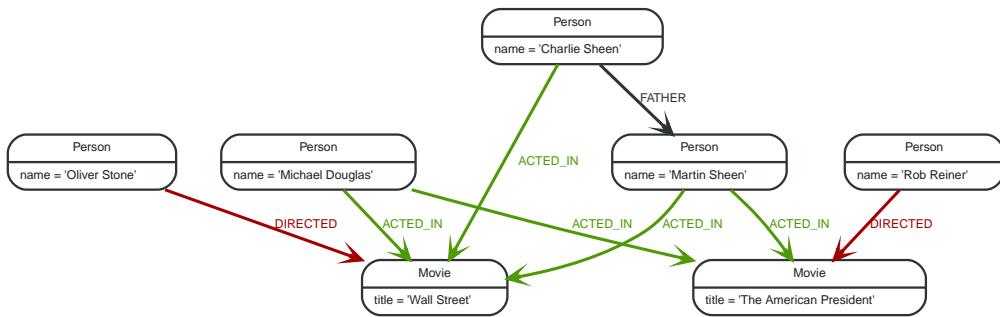


Figure 9. Graph

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 98. 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 99. 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 100. Result

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

7.3. START

Find starting points through explicit indexes.

The `START` clause was removed in Cypher 3.2, and the recommendation is to use `MATCH` instead (see [MATCH](#)). However, if the use of explicit indexes is required, a series of [built-in procedures](#) allows these to be managed and used. These procedures offer the same functionality as the `START` clause. In addition, queries using these procedures may exhibit superior execution performance over queries using `START` owing to the use of the [cost planner](#) and newer Cypher 3.2 compiler.



Using the `START` clause explicitly in a query will cause the query to fall back to using Cypher 3.1.

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.

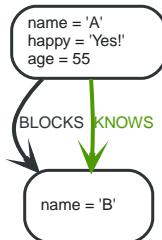


Figure 10. Graph

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 101. 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 102. 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 103. 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 104. Result

a	b	p	r
Node[0]{name: "A", happy: "Yes!", age:55}	Node[1]{name: "B"}	(0)-[BLOCKS,1]->(1)	:BLOCKS[1]{}
Node[0]{name: "A", happy: "Yes!", age:55}	Node[1]{name: "B"}	(0)-[KNOWS,0]->(1)	: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 105. 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 106. 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 107. 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 108. Result

a.age > 30	"I'm a literal"	(a)-->()
true	"I'm a literal"	JavaListWrapper((0)-[BLOCKS,1]->(1), (0)-[KNOWS,0]->(1))
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 109. Result

b
Node[1]{name : "B"}
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.

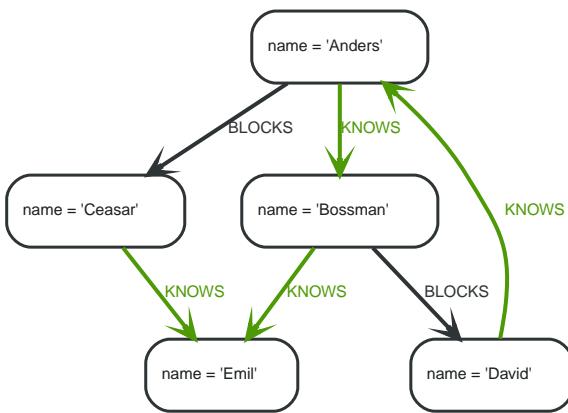


Figure 11. Graph

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 the at least more than one outgoing relationship will be returned by the query.

Table 110. 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 111. Result

collect(n.name)
JavaListWrapper(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 112. Result

o.name
"Bossman"
"Anders"
2 rows

7.6. UNWIND

UNWIND expands a list into a sequence of rows.

- [Introduction](#)
- [Unwinding a list](#)
- [Creating a distinct list](#)
- [Using UNWIND with any expression returning a list](#)
- [Using UNWIND with a list of lists](#)
- [Using UNWIND with an empty list](#)
- [Using UNWIND with an expression that is not a list](#)
- [Creating 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. Unwinding a list

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

Query

```
UNWIND [1, 2, 3, NULL ] AS x
RETURN x, 'val' AS y
```

Each value of the original list — including `null` — is returned as an individual row.

Table 113. Result

x	y
1	"val"
2	"val"
3	"val"
<null>	"val"
4 rows	

7.6.3. Creating 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 setOfVals
```

Each value of the original list is unwound and passed through `DISTINCT` to create a unique set.

Table 114. Result

setOfVals
JavaListWrapper(1, 2)
1 row

7.6.4. Using `UNWIND` with any expression returning a list

Any expression that returns a list may be used with `UNWIND`.

Query

```
WITH [1, 2] AS a,[3, 4] AS b
UNWIND (a + b) AS x
RETURN x
```

The two lists — a and b — are concatenated to form a new list, which is then operated upon by `UNWIND`.

Table 115. Result

x
1
2
3
4
4 rows

7.6.5. Using UNWIND with a list of lists

Multiple `UNWIND` clauses can be chained to unwind nested list elements.

Query

```
WITH [[1, 2], [3, 4], 5] AS nested
UNWIND nested AS x
UNWIND x AS y
RETURN y
```

The first `UNWIND` results in three rows for `x`, each of which contains an element of the original list (two of which are also lists); namely, `[1, 2]`, `[3, 4]` and `5`. The second `UNWIND` then operates on each of these rows in turn, resulting in five rows for `y`.

Table 116. Result

y
1
2
3
4
5
5 rows

7.6.6. Using UNWIND with an empty list

Using an empty list with `UNWIND` will produce no rows, irrespective of whether or not any rows existed beforehand, or whether or not other values are being projected.

Essentially, `UNWIND []` reduces the number of rows to zero, and thus causes the query to cease its execution, returning no results. This has value in cases such as `UNWIND v`, where `v` is a variable from an earlier clause that may or may not be an empty list — when it is an empty list, this will behave just as a `MATCH` that has no results.

Query

```
UNWIND [] AS empty
RETURN empty, 'literal_that_is_not_returned'
```

Table 117. Result

(empty result)
0 rows

To avoid inadvertently using `UNWIND` on an empty list, `CASE` may be used to replace an empty list with a `null`:

```
WITH [] AS list
UNWIND
CASE
  WHEN list = []
    THEN [null]
  ELSE list
END AS emptylist
RETURN emptylist
```

7.6.7. Using UNWIND with an expression that is not a list

Attempting to use `UNWIND` on an expression that does not return a list — such as `UNWIND 5` — will cause an error. The exception to this is when the expression returns `null` — this will reduce the number of rows to zero, causing it to cease its execution and return no results.

Query

```
UNWIND NULL AS x
RETURN x, 'some_literal'
```

Table 118. Result

(empty result)

0 rows

7.6.8. Creating 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.

Table 119. 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:

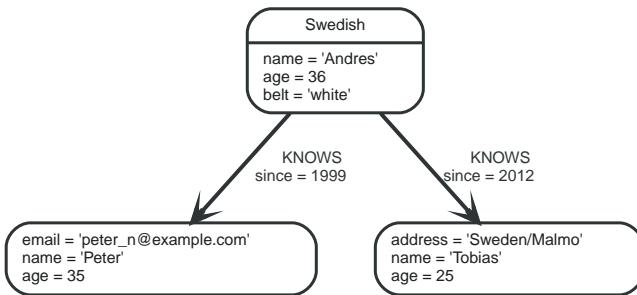


Figure 12. Graph

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 120. Result

n.name	n.age
"Andres"	36
"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 121. Result

n.name	n.age
"Andres"	36

n.name	n.age
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 122. 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 123. 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[toLower(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 124. 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 125. 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 126. 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 127. 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 128. 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 129. 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.*' RETURN n
```

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 130. Result

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

Escaping in regular expressions

Characters like `.` or `*` have special meaning in a regular expression. To use these as ordinary characters, without special meaning, escape them.

Query

```
MATCH (n)
WHERE n.email =~ '.*\\\\.com'
RETURN n.name, n.age, n.email
```

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

Table 131. Result

n.name	n.age	n.email
"Peter"	35	"peter_n@example.com"
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 132. Result

n.name	n.age
"Andres"	36

n.name	n.age
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 133. 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 134. 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 135. 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 136. 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 137. 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 138. 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 139. 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 140. 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 141. 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 142. 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.

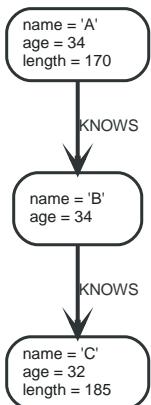


Figure 13. Graph

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 143. 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 144. 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 145. 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 146. 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.

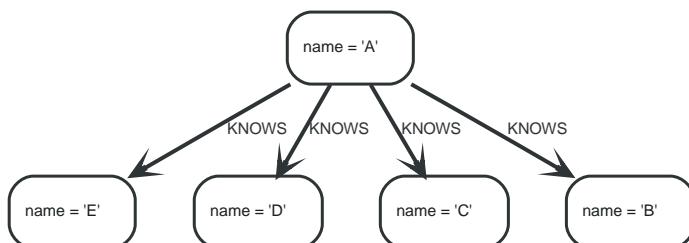


Figure 14. Graph

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 147. 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 148. 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 toInteger(3*rand())+ 1
```

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

Table 149. Result

n.name
"C"
"D"
"E"
3 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.

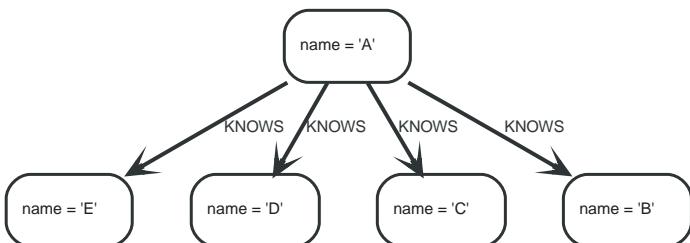


Figure 15. Graph

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 150. 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 toInteger(3 * rand())+ 1
  
```

Returns one to three top items.

Table 151. 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 extensively. 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.

Table 152. Result

(empty result)
0 rows
Nodes created: 1

Create multiple nodes

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

Query

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

Table 153. Result

(empty result)

0 rows
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.

Table 154. Result

(empty result)

0 rows
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.

Table 155. Result

(empty result)

0 rows
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.

Table 156. Result

(empty result)
0 rows
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.name
```

The newly-created node is returned.

Table 157. Result

a.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 = 'A' AND b.name = 'B'
CREATE (a)-[r:RELTYPE]->(b)
RETURN type(r)
```

The created relationship is returned by the query.

Table 158. Result

type(r)
"RELTYPE"
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 = 'A' AND b.name = 'B'
CREATE (a)-[r:RELTYPE { name: a.name + '<->' + b.name }]->(b)
RETURN type(r), r.name
```

The newly-created relationship is returned by the example query.

Table 159. Result

type(r)	r.name
"RELTYPE"	"A<->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.

Table 160. Result

p
(20)-[WORKS_AT,0]->(21)<-[WORKS_AT,1]-(22)

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
```

Table 161. Result

n
Node[20]{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
```

Table 162. Result

(empty result)
0 rows
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:

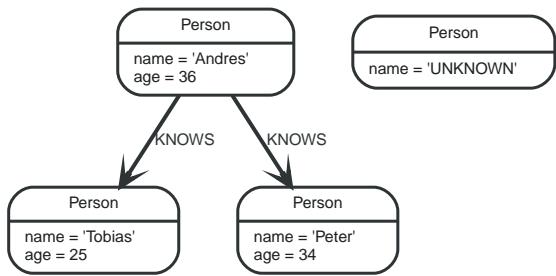


Figure 16. Graph

7.12.2. Delete single node

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

Query

```
MATCH (n:Person { name: 'UNKNOWN' })  
DELETE n
```

Table 163. Result

(empty result)
0 rows
Nodes deleted: 1

7.12.3. Delete all nodes and relationships

This query isn't for deleting large amounts of data, but is useful when experimenting with small example data sets.

Query

```
MATCH (n)  
DETACH DELETE n
```

Table 164. Result

(empty result)
0 rows
Nodes deleted: 4
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
```

Table 165. Result

```
(empty result)
```

0 rows

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**'.

Table 166. Result

```
(empty result)
```

0 rows

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:

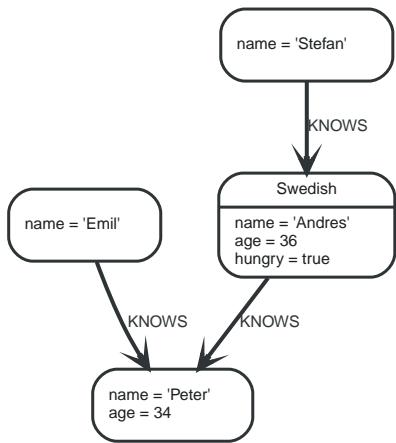


Figure 17. Graph

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.name, n.surname

```

The newly changed node is returned by the query.

Table 167. Result

n.name	n.surname
"Andres"	"Taylor"

1 row
Properties set: 1

7.13.3. Remove a property

Normally you remove a property by using **REMOVE**, but it's sometimes convenient 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.name, n.age

```

The node is returned by the query, and the name property is now missing.

Table 168. Result

n.name	n.age
<null>	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. 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.name, at.age, at.hungry, pn.name, pn.age
```

The 'Andres' node has had all its properties replaced by the properties in the 'Peter' node.

Table 169. Result

at.name	at.age	at.hungry	pn.name	pn.age
"Peter"	34	<null>	"Peter"	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 (p { name: 'Peter' })
SET p += { hungry: TRUE, position: 'Entrepreneur' }
```

Table 170. Result

(empty result)
0 rows 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.name, n.surname
```

The 'Andres' node has got a surname added.

Table 171. Result

n.name	n.surname
"Andres"	"Taylor"

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.name, n.position, n.age, n.hungry
```

The 'Andres' node has had all its properties replaced by the properties in the `props` parameter.

Table 172. Result

n.name	n.position	n.age	n.hungry
"Andres"	"Developer"	<null>	<null>

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'
```

Table 173. Result

(empty result)
0 rows 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.name, labels(n) AS labels
```

The newly labeled node is returned by the query.

Table 174. Result

n.name	labels
"Stefan"	JavaListWrapper(German)
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.name, labels(n) AS labels
```

The newly labeled node is returned by the query.

Table 175. Result

n.name	labels
"Emil"	JavaListWrapper(Swedish, Bossman)
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:

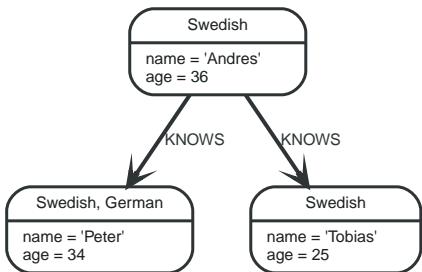


Figure 18. Graph

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, `REMOVE` is used to remove a property value from a node or a relationship.

Query

```
MATCH (a { name: 'Andres' })
REMOVE a.age
RETURN a.name, a.age
```

The node is returned, and no property `age` exists on it.

Table 176. Result

a.name	a.age
"Andres"	<null>
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.name, labels(n)
```

Table 177. Result

n.name	labels(n)
"Peter"	JavaListWrapper(Swedish)
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.name, labels(n)
```

Table 178. Result

n.name	labels(n)
"Peter"	JavaListWrapper()
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. `FOREACH` can be used to update data, such as executing update commands on elements in a path, or on 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.

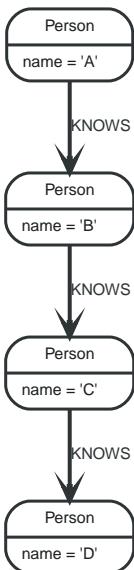


Figure 19. Graph

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.

Table 179. Result

```
(empty result)
```

0 rows

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 following graph is used for the examples below:

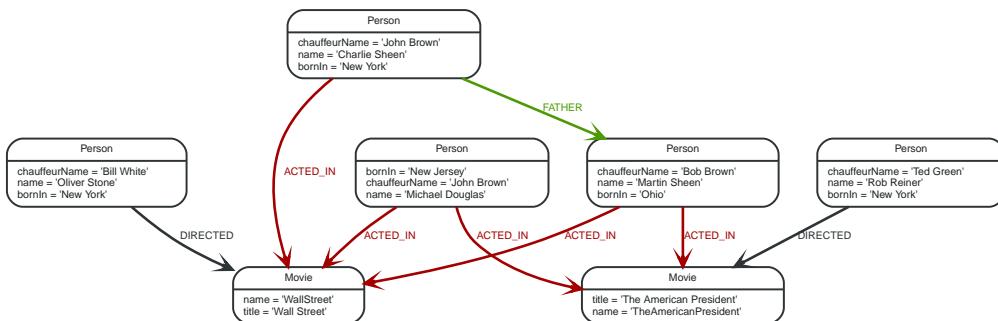


Figure 20. Graph

7.16.2. Merge nodes

Merge single node with a label

Merging a single node with the 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"	1529531045036

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" | 1529531052909 | <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 | 1529531049822 |  
| "Oliver Stone" | true | 1529531049822 |  
| "Charlie Sheen" | true | 1529531049822 |  
| "Michael Douglas" | true | 1529531049822 |  
| "Martin Sheen" | true | 1529531049822 |  
+-----+  
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 unique constraints. In this case, there must be at most one node that matches that pattern.

For example, given two unique 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 unique constraints that have been created using:

```
CREATE CONSTRAINT ON (n:Person) ASSERT n.name IS UNIQUE;
CREATE CONSTRAINT ON (n:Person) ASSERT n.role IS UNIQUE;
```

Merge using unique constraints creates a new node if no node is found

Merge using unique constraints creates a new node if no node is found.

Query

```
MERGE (laurence:Person { name: 'Laurence Fishburne' })
RETURN laurence.name
```

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

existing node.

Result

```
+-----+  
| laurence.name |  
+-----+  
| "Laurence Fishburne" |  
+-----+  
1 row  
Nodes created: 1  
Properties set: 1  
Labels added: 1
```

Merge using unique constraints matches an existing node

Merge using unique constraints matches an existing node.

Query

```
MERGE (oliver:Person { name: 'Oliver Stone' })  
RETURN oliver.name, oliver.bornIn
```

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

Result

```
+-----+  
| oliver.name | oliver.bornIn |  
+-----+  
| "Oliver Stone" | "New York" |  
+-----+  
1 row
```

Merge with unique constraints and partial matches

Merge using unique constraints fails when finding partial matches.

Query

```
MERGE (michael:Person { name: 'Michael Douglas', role: 'Gordon Gekko' })  
RETURN michael
```

While there is a matching unique 'michael' node with the name '**Michael Douglas**', there is no unique node with the role of '**Gordon Gekko**' and **MERGE** fails to match.

Error message

```
Merge did not find a matching node 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**](#)
- [View the signature for a procedure](#)
- [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. Thus, in order to be able to use `YIELD`, the names (and types) of the output parameters need be known in advance. 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).

This section explains how to determine a procedure's input parameters (needed for `CALL`) and output parameters (needed for `YIELD`).

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).



Neo4j comes with a number of built-in procedures. For a list of these, see [Operations Manual □ Built-in procedures](#).

Users can also develop custom procedures and deploy to the database. See [Procedures](#) for details.

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 explicit 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 ) );
    }
}

```

7.17.2. Call a procedure using `CALL`

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. View the signature for a procedure

To `CALL` a procedure, its input parameters need to be known, and to use `YIELD`, its output parameters need to be known. The built-in procedure `dbms.procedures` returns the name, signature and description for all procedures. The following query can be used to return the signature for a particular procedure:

Query

```

CALL dbms.procedures() YIELD name, signature
WHERE name='dbms.listConfig'
RETURN signature

```

We can see that the `dbms.listConfig` has one input parameter, `searchString`, and three output parameters, `name`, `description` and `value`.

Result

```
+-----+
|-----+
| signature
|
+-----+
| "dbms.listConfig(searchString = :: STRING?) :: (name :: STRING?, description :: STRING?, value :: STRING?)" |
+-----+
-----+
1 row
```

7.17.4. 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
```

7.17.5. 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. |
+-----+
```

7.17.6. 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. |  
+-----+
```

7.17.7. 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. |  
+-----+
```

7.17.8. 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. |
+-----+
```

7.17.9. Call a procedure within a complex query using `CALL YIELD`

This calls the built-in procedure `db.labels` to count all labels used in the database.

Query

```
CALL db.labels() YIELD label
RETURN count(label) AS numLabels
```

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

Result

```
+-----+
| numLabels |
+-----+
| 2         |
+-----+
1 row
```

7.17.10. 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
```

7.17.11. 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.



The `CREATE UNIQUE` clause was removed in Cypher 3.2. Using the `CREATE UNIQUE` clause will cause the query to fall back to using Cypher 3.1. Use `MERGE` instead of `CREATE UNIQUE`; refer to the [Introduction](#) for an example of how to achieve the same level of node and relationship uniqueness.

- [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

`CREATE UNIQUE` is in the middle of `MATCH` and `CREATE` — it will match what it can, and create what is missing.

We show in the following example how to express using `MERGE` the same level of uniqueness guaranteed by `CREATE UNIQUE` for nodes and relationships.

Assume the original set of queries is given by:

```
MERGE (p:Person {name: 'Joe'})
RETURN p

MATCH (a:Person {name: 'Joe'})
CREATE UNIQUE (a)-[r:LIKES]->(b:Person {name: 'Jill'})-[r1:EATS]->(f:Food {name: 'Margarita Pizza'})
RETURN a

MATCH (a:Person {name: 'Joe'})
CREATE UNIQUE (a)-[r:LIKES]->(b:Person {name: 'Jill'})-[r1:EATS]->(f:Food {name: 'Banana'})
RETURN a
```

This will create two `:Person` nodes, a `:LIKES` relationship between them, and two `:EATS` relationships

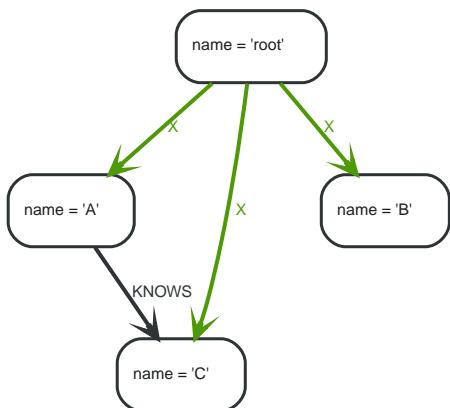
from one of the `:Person` nodes to two `:Food` nodes. No node or relationship is duplicated.

The following set of queries — using `MERGE` — will achieve the same result:

```
MERGE (p:Person {name: 'Joe'})  
RETURN p  
  
MATCH (a:Person {name: 'Joe'})  
MERGE (b:Person {name: 'Jill'})  
MERGE (a)-[r:LIKES]->(b)  
MERGE (b)-[r1:EATS]->(f:Food {name: 'Margarita Pizza'})  
RETURN a  
  
MATCH (a:Person {name: 'Joe'})  
MERGE (b:Person {name: 'Jill'})  
MERGE (a)-[r:LIKES]->(b)  
MERGE (b)-[r1:EATS]->(f:Food {name: 'Banana'})  
RETURN a
```

We note that all these queries can also be combined into a single, larger query.

The `CREATE UNIQUE` examples below 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[20]{ } |  
+-----+  
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[20]{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[20]{} |
+-----+
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[20]{}
| :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[20]{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[20]{ } |
+-----+
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.

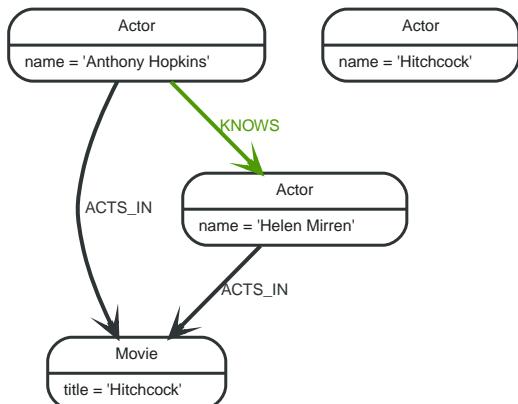


Figure 21. Graph

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 180. 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 181. 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: toInteger(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: toInteger(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`. Hexadecimal representation of the unicode character encoding can be used if prepended by `\u`. The encoding must be written with four digits. For example, `\u002C` is equivalent to `,`.

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: toInteger(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: toInteger(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: toInteger(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: toInteger(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](#)]
- Temporal functions [[Summary](#) | [Detail](#)]
- Spatial functions [[Summary](#) | [Detail](#)]
- [User-defined functions](#)

Related information may be found in [Operators](#).

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



Functions taking a string as input all operate on *Unicode characters* rather than on a standard `char[]`. For example, `size(s)`, where `s` is a character in the Chinese alphabet, will return 1.

Predicate functions

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

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

Scalar functions

These functions return a single value.

Function	Description
<code>coalesce()</code>	Returns the first non- <code>null</code> value in a list of expressions.
<code>endNode()</code>	Returns the end node of a relationship.
<code>head()</code>	Returns the first element in a list.
<code>id()</code>	Returns the id of a relationship or node.

Function	Description
last()	Returns the last element in a list.
length()	Returns the length of a path.
properties()	Returns a map containing all the properties of a node or relationship.
randomUUID()	Returns a string value corresponding to a randomly-generated UUID.
size()	Returns the number of items in a list.
size() applied to pattern expression	Returns the number of sub-graphs matching the pattern expression.
size() applied to string	Returns the number of Unicode characters in a string.
startNode()	Returns the start node of a relationship.
timestamp()	Returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.
toBoolean()	Converts a string value to a boolean value.
toFloat()	Converts an integer or string value to a floating point number.
toInteger()	Converts a floating point or string value to an integer value.
type()	Returns the 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
avg()	Returns the average of a set of numeric values.
collect()	Returns a list containing the values returned by an expression.
count()	Returns the number of values or rows.
max()	Returns the maximum value in a set of values.
min()	Returns the minimum value in a set of values.
percentileCont()	Returns the percentile of a value over a group using linear interpolation.
percentileDisc()	Returns the nearest value to the given percentile over a group using a rounding method.
stDev()	Returns the standard deviation for the given value over a group for a sample of a population.
stDevP()	Returns the standard deviation for the given value over a group for an entire population.
sum()	Returns the sum of a set of numeric values.

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 list <code>l_{result}</code> containing the values resulting from an expression which has been applied to each element in a list <code>list</code> .
<code>filter()</code>	Returns a list <code>l_{result}</code> containing all the elements from a list <code>list</code> that comply with a predicate.
<code>keys()</code>	Returns a list containing the string representations for all the property names of a node, relationship, or map.
<code>labels()</code>	Returns a list containing the string representations for all the labels of a node.
<code>nodes()</code>	Returns a list containing all the nodes in a path.
<code>range()</code>	Returns a list comprising all integer values within a specified range.
<code>reduce()</code>	Runs an expression against individual elements of a list, storing the result of the expression in an accumulator.
<code>relationships()</code>	Returns a list containing all the relationships in a path.
<code>reverse()</code>	Returns a list in which the order of all elements in the original list have been reversed.
<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 floating point number that is greater than or equal to a number and equal to a mathematical integer.
<code>floor()</code>	Returns the largest floating point number that is less than or equal to a number and equal to a mathematical integer.
<code>rand()</code>	Returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e. <code>[0, 1)</code> .
<code>round()</code>	Returns the value of a number rounded to the nearest integer.
<code>sign()</code>	Returns the signum of a number: <code>0</code> if the number is <code>0</code> , <code>-1</code> for any negative number, and <code>1</code> 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, <code>e</code> .
<code>exp()</code>	Returns <code>e^n</code> , where <code>e</code> is the base of the natural logarithm, and <code>n</code> is the value of the argument expression.
<code>log()</code>	Returns the natural logarithm of a number.
<code>log10()</code>	Returns the common logarithm (base 10) of a number.
<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
acos()	Returns the arccosine of a number in radians.
asin()	Returns the arcsine of a number in radians.
atan()	Returns the arctangent of a number in radians.
atan2()	Returns the arctangent2 of a set of coordinates in radians.
cos()	Returns the cosine of a number.
cot()	Returns the cotangent of a number.
degrees()	Converts radians to degrees.
haversin()	Returns half the versine of a number.
pi()	Returns the mathematical constant <i>pi</i> .
radians()	Converts degrees to radians.
sin()	Returns the sine of a number.
tan()	Returns the tangent of a number.

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 specified number of leftmost characters of the original string.
lTrim()	Returns the original string with leading whitespace removed.
replace()	Returns a string in which all occurrences of a specified string in the original string have been replaced by another (specified) string.
reverse()	Returns a string in which the order of all characters in the original string have been reversed.
right()	Returns a string containing the specified number of rightmost characters of the original string.
rTrim()	Returns the original string with trailing whitespace removed.
split()	Returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.
substring()	Returns a substring of the original string, beginning with a 0-based index start and length.
toLower()	Returns the original string in lowercase.
toString()	Converts an integer, float, boolean or temporal type (i.e. Date, Time, LocalTime, DateTime, LocalDateTime or Duration) value to a string.
toUpper()	Returns the original string in uppercase.
trim()	Returns the original string with leading and trailing whitespace removed.

Temporal functions

Values of the [temporal types](#) — *Date*, *Time*, *LocalTime*, *DateTime*, *LocalDateTime* and *Duration* — can be created manipulated using the following functions:

Function	Description
<code>date()</code>	Returns the current <i>Date</i> .
<code>date.transaction()</code>	Returns the current <i>Date</i> using the transaction clock.
<code>date.statement()</code>	Returns the current <i>Date</i> using the statement clock.
<code>date.realtime()</code>	Returns the current <i>Date</i> using the realtime clock.
<code>date({year [, month, day]})</code>	Returns a calendar (Year-Month-Day) <i>Date</i> .
<code>date({year [, week, dayOfWeek]})</code>	Returns a week (Year-Week-Day) <i>Date</i> .
<code>date({year [, quarter, dayOfQuarter]})</code>	Returns a quarter (Year-Quarter-Day) <i>Date</i> .
<code>date({year [, ordinalDay]})</code>	Returns an ordinal (Year-Day) <i>Date</i> .
<code>date(string)</code>	Returns a <i>Date</i> by parsing a string.
<code>date({map})</code>	Returns a <i>Date</i> from a map of another temporal value's components.
<code>date.truncate()</code>	Returns a <i>Date</i> obtained by truncating a value at a specific component boundary. Truncation summary .
<code>datetime()</code>	Returns the current <i>DateTime</i> .
<code>datetime.transaction()</code>	Returns the current <i>DateTime</i> using the transaction clock.
<code>datetime.statement()</code>	Returns the current <i>DateTime</i> using the statement clock.
<code>datetime.realtime()</code>	Returns the current <i>DateTime</i> using the realtime clock.
<code>datetime({year [, month, day, ...]})</code>	Returns a calendar (Year-Month-Day) <i>DateTime</i> .
<code>datetime({year [, week, dayOfWeek, ...]})</code>	Returns a week (Year-Week-Day) <i>DateTime</i> .
<code>datetime({year [, quarter, dayOfQuarter, ...]})</code>	Returns a quarter (Year-Quarter-Day) <i>DateTime</i> .
<code>datetime({year [, ordinalDay, ...]})</code>	Returns an ordinal (Year-Day) <i>DateTime</i> .
<code>datetime(string)</code>	Returns a <i>DateTime</i> by parsing a string.
<code>datetime({map})</code>	Returns a <i>DateTime</i> from a map of another temporal value's components.
<code>datetime({epochSeconds})</code>	Returns a <i>DateTime</i> from a timestamp.
<code>datetime.truncate()</code>	Returns a <i>DateTime</i> obtained by truncating a value at a specific component boundary. Truncation summary .
<code>localdatetime()</code>	Returns the current <i>LocalDateTime</i> .
<code>localdatetime.transaction()</code>	Returns the current <i>LocalDateTime</i> using the transaction clock.
<code>localdatetime.statement()</code>	Returns the current <i>LocalDateTime</i> using the statement clock.
<code>localdatetime.realtime()</code>	Returns the current <i>LocalDateTime</i> using the realtime clock.
<code>localdatetime({year [, month, day, ...]})</code>	Returns a calendar (Year-Month-Day) <i>LocalDateTime</i> .
<code>localdatetime({year [, week, dayOfWeek, ...]})</code>	Returns a week (Year-Week-Day) <i>LocalDateTime</i> .
<code>localdatetime({year [, quarter, dayOfQuarter, ...]})</code>	Returns a quarter (Year-Quarter-Day) <i>DateTime</i> .
<code>localdatetime({year [, ordinalDay, ...]})</code>	Returns an ordinal (Year-Day) <i>LocalDateTime</i> .
<code>localdatetime(string)</code>	Returns a <i>LocalDateTime</i> by parsing a string.

Function	Description
localdatetime({map})	Returns a <i>LocalDateTime</i> from a map of another temporal value's components.
localdatetime.truncate()	Returns a <i>LocalDateTime</i> obtained by truncating a value at a specific component boundary. Truncation summary .
localtime()	Returns the current <i>LocalTime</i> .
localtime.transaction()	Returns the current <i>LocalTime</i> using the <code>transaction</code> clock.
localtime.statement()	Returns the current <i>LocalTime</i> using the <code>statement</code> clock.
localtime.realtime()	Returns the current <i>LocalTime</i> using the <code>realtime</code> clock.
localtime({hour [, minute, second, ...]})	Returns a <i>LocalTime</i> with the specified component values.
localtime(string)	Returns a <i>LocalTime</i> by parsing a string.
localtime({time [, hour, ...]})	Returns a <i>LocalTime</i> from a map of another temporal value's components.
localtime.truncate()	Returns a <i>LocalTime</i> obtained by truncating a value at a specific component boundary. Truncation summary .
time()	Returns the current <i>Time</i> .
time.transaction()	Returns the current <i>Time</i> using the <code>transaction</code> clock.
time.statement()	Returns the current <i>Time</i> using the <code>statement</code> clock.
time.realtime()	Returns the current <i>Time</i> using the <code>realtime</code> clock.
time({hour [, minute, ...]})	Returns a <i>Time</i> with the specified component values.
time(string)	Returns a <i>Time</i> by parsing a string.
time({time [, hour, ..., timezone]})	Returns a <i>Time</i> from a map of another temporal value's components.
time.truncate()	Returns a <i>Time</i> obtained by truncating a value at a specific component boundary. Truncation summary .
duration({map})	Returns a <i>Duration</i> from a map of its components.
duration(string)	Returns a <i>Duration</i> by parsing a string.
duration.between()	Returns a <i>Duration</i> equal to the difference between two given instants.
duration.inMonths()	Returns a <i>Duration</i> equal to the difference in whole months, quarters or years between two given instants.
duration.inDays()	Returns a <i>Duration</i> equal to the difference in whole days or weeks between two given instants.
duration.inSeconds()	Returns a <i>Duration</i> equal to the difference in seconds and fractions of seconds, or minutes or hours, between two given instants.

Spatial functions

These functions are used to specify 2D or 3D points in a geographic or cartesian Coordinate Reference 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 points in the same CRS.
point() - Cartesian 2D	Returns a 2D point object, given two coordinate values in the Cartesian coordinate system.
point() - Cartesian 3D	Returns a 3D point object, given three coordinate values in the Cartesian coordinate system.

Function	Description
<code>point() - WGS 84 2D</code>	Returns a 2D point object, given two coordinate values in the WGS 84 geographic coordinate system.
<code>point() - WGS 84 3D</code>	Returns a 3D point object, given three coordinate values in the WGS 84 geographic 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()`

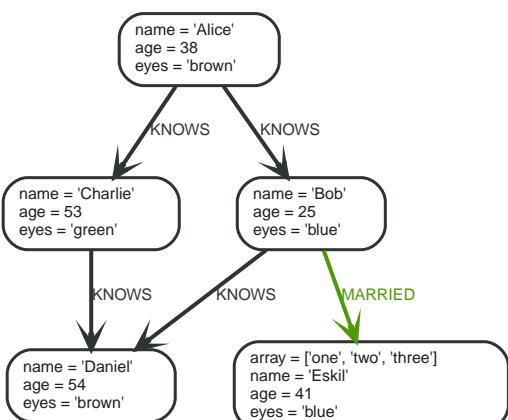


Figure 22. Graph

8.1.1. `all()`

`all()` returns true if the predicate holds for all elements in the given list.

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

Returns:

A Boolean.

Arguments:

Name	Description
<code>list</code>	An expression that returns a list.
<code>variable</code>	This is the variable that can be used from within the predicate.
<code>predicate</code>	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 182. Result

p
(0)-[KNOWS,1]->(2)-[KNOWS,3]->(3)
1 row

8.1.2. any()

`any()` returns true if the predicate holds for at least one element in the given list.

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

Returns:

A Boolean.

Arguments:

Name	Description
<code>list</code>	An expression that returns a list.
<code>variable</code>	This is the variable that can be used from within 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 183. Result

a.name	a.array
"Eskil"	[Ljava.lang.String;@4041b733
1 row	

8.1.3. exists()

`exists()` returns true if a match for the given pattern exists in the graph, or if the specified property exists in the node, relationship or map.

Syntax: `exists(pattern-or-property)`

Returns:

A Boolean.

Arguments:

Name	Description
pattern-or-property	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
```

The names of all nodes with the `name` property are returned, along with a boolean `true / false` indicating if they are married.

Table 184. Result

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

8.1.4. none()

`none()` returns true if the predicate holds for no element in the given list.

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

Returns:

A Boolean.

Arguments:

Name	Description
list	An expression that returns a list.
variable	This is the variable that can be used from within the predicate.
predicate	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 node in the returned paths has an `age` property set to '25'.

Table 185. Result

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

8.1.5. single()

`single()` returns true if the predicate holds for exactly one of the elements in the given list.

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

Returns:

A Boolean.

Arguments:

Name	Description
<code>list</code>	An expression that returns a list.
<code>variable</code>	This is the variable that can be used from within 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 has the `eyes` property set to 'blue'.

Table 186. Result

p
(0)-[KNOWS,0]->(1)
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 `timestamp()` function returns the equivalent value of `datetime().epochMillis`.



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

Functions:

- `coalesce()`
- `endNode()`
- `head()`
- `id()`
- `last()`
- `length()`
- `properties()`
- `randomUUID()`
- `size()`
- **Size of pattern expression**
- **Size of string**
- `startNode()`
- `timestamp()`
- `toBoolean()`
- `toFloat()`
- `toInteger()`
- `type()`

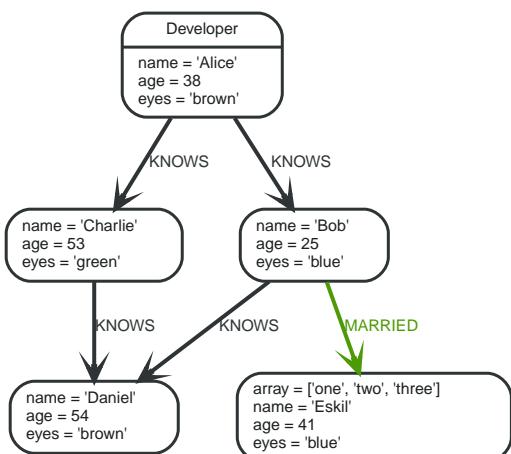


Figure 23. Graph

8.2.1. coalesce()

`coalesce()` returns the first non-`null` value in the given list of expressions.

Syntax: `coalesce(expression [, expression]*)`

Returns:

The type of the value returned will be that of the first non-`null` expression.

Arguments:

Name	Description
expression	An expression which may return <code>null</code> .

Considerations:

`null` will be returned if all the arguments are `null`.

Query

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

Table 187. Result

coalesce(a.hairColor, a.eyes)
"brown"
1 row

8.2.2. endNode()

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

Syntax: `endNode(relationship)`

Returns:

A Node.

Arguments:

Name	Description
relationship	An expression that returns a relationship.

Considerations:

`endNode(null)` returns `null`.

Query

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

Table 188. 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(list)`

Returns:

The type of the value returned will be that of the first element of `list`.

Arguments:

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

Considerations:

`head(null)` returns `null`.

If the first element in `list` is `null`, `head(list)` will return `null`.

Query

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

The first element in the list is returned.

Table 189. Result

a.array	head(a.array)
[Ljava.lang.String;@5f96ea5	"one"
1 row	

8.2.4. id()

`id()` returns the id of a relationship or node.

Syntax: `id(expression)`

Returns:

An Integer.

Arguments:

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

Considerations:

`id(null)` returns `null`.

Query

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

The node id for each of the nodes is returned.

Table 190. Result

id(a)
0
1
2
3
4
5 rows

8.2.5. last()

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

Syntax: `last(expression)`

Returns:

The type of the value returned will be that of the last element of `list`.

Arguments:

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

Considerations:

`last(null)` returns `null`.

If the last element in `list` is `null`, `last(list)` will return `null`.

Query

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

The last element in the list is returned.

Table 191. Result

a.array	last(a.array)
[Ljava.lang.String;@41db1728	"three"
1 row	

8.2.6. length()

`length()` returns the length of a path.

Syntax: `length(path)`

Returns:

An Integer.

Arguments:

Name	Description
path	An expression that returns a path.

Considerations:

`length(null)` returns `null`.

Query

```
MATCH p =(a)-->(b)-->(c)
WHERE a.name = 'Alice'
RETURN length(p)
```

The length of the path `p` is returned.

Table 192. Result

length(p)
2
2
2
3 rows

8.2.7. properties()

`properties()` returns a map containing all the properties of a node or relationship. If the argument is already a map, it is returned unchanged.

Syntax: `properties(expression)`

Returns:

A Map.

Arguments:

Name	Description
expression	An expression that returns a node, a relationship, or a map.

Considerations:

`properties(null)` returns `null`.

Query

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

Table 193. Result

```
properties(p)
```

```
{city -> "Berlin", name -> "Stefan"}
```

```
1 row
```

```
Nodes created: 1
```

```
Properties set: 2
```

```
Labels added: 1
```

8.2.8. randomUUID()

`randomUUID()` returns a randomly-generated Universally Unique Identifier (UUID), also known as a Globally Unique Identifier (GUID). This is a 128-bit value with strong guarantees of uniqueness.

Syntax: `randomUUID()`

Returns:

```
A String.
```

Query

```
RETURN randomUUID() AS uuid
```

Table 194. Result

uuid
"9eb1e9e9-4eb9-4366-a6b8-849f990b03d9"
1 row

A randomly-generated UUID is returned.

8.2.9. size()

`size()` returns the number of elements in a list.

Syntax: `size(list)`

Returns:

```
An Integer.
```

Arguments:

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

Considerations:

```
size(null) returns null.
```

Query

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

Table 195. Result

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

```
2
```

```
1 row
```

The number of elements in the list is returned.

8.2.10. size() applied to pattern expression

This is the same `size()` method as described above, but instead of passing in a list directly, a pattern expression can be provided 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 196. Result

<code>fofof</code>
<code>3</code>
1 row

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

8.2.11. size() applied to string

`size()` returns the number of Unicode characters in a string.

Syntax: `size(string)`

Returns:

```
An Integer.
```

Arguments:

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

Considerations:

```
size(null) returns null.
```

Query

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

Table 197. Result

size(a.name)
7
1 row

The number of characters in the string 'Charlie' is returned.

8.2.12. startNode()

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

Syntax: `startNode(relationship)`

Returns:

A Node.

Arguments:

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

Considerations:

`startNode(null)` returns `null`.

Query

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

Table 198. Result

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

8.2.13. timestamp()

`timestamp()` returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.

Syntax: `timestamp()`

Returns:

An Integer.

Considerations:

`timestamp()` will return the same value during one entire query, even for long-running queries.

Query

```
RETURN timestamp()
```

The time in milliseconds is returned.

Table 199. Result

<code>timestamp()</code>
1529531057649
1 row

8.2.14. toBoolean()

`toBoolean()` converts a string value to a boolean value.

Syntax: `toBoolean(expression)`

Returns:

A Boolean.

Arguments:

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

Considerations:

`toBoolean(null)` returns `null`.

If `expression` is a boolean value, it will be returned unchanged.

If the parsing fails, `null` will be returned.

Query

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

Table 200. Result

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

8.2.15. toFloat()

`toFloat()` converts an integer or string value to a floating point number.

Syntax: `toFloat(expression)`

Returns:

A Float.

Arguments:

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

Considerations:

`toFloat(null)` returns `null`.

If `expression` is a floating point number, it will be returned unchanged.

If the parsing fails, `null` will be returned.

Query

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

Table 201. Result

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

8.2.16. `tolnteger()`

`toInteger()` converts a floating point or string value to an integer value.

Syntax: `toInteger(expression)`

Returns:

An Integer.

Arguments:

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

Considerations:

`toInteger(null)` returns `null`.

If `expression` is an integer value, it will be returned unchanged.

If the parsing fails, `null` will be returned.

Query

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

Table 202. Result

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

8.2.17. type()

`type()` returns the string representation of the relationship type.

Syntax: `type(relationship)`

Returns:

A String.

Arguments:

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

Considerations:

`type(null)` returns `null`.

Query

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

The relationship type of `r` is returned.

Table 203. Result

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

8.3. Aggregating functions

To calculate aggregated data, Cypher offers aggregation, analogous to SQL's `GROUP BY`.

Aggregating functions take a set of values and calculate an aggregated value over them. Examples are `avg()` that calculates the average of multiple numeric values, or `min()` that finds the smallest numeric or string value in a set of values. When we say below that an aggregating function operates on a *set of values*, we mean these to be the result of the application of the inner expression (such as `n.age`) to all the records within the same aggregation group.

Aggregation can be computed over all the matching subgraphs, or it can be further divided by introducing grouping keys. These are non-aggregate expressions, that are used to group the values going into the aggregate functions.

Assume we have the following return statement:

```
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. The matching subgraphs will be divided into different buckets, depending on the grouping key. The aggregate function will then be run on these buckets, calculating an aggregate value per bucket.

To use aggregations to sort the result set, the aggregation must be included in the `RETURN` to be used in the `ORDER BY`.

The `DISTINCT` operator works in conjunction with aggregation. 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:

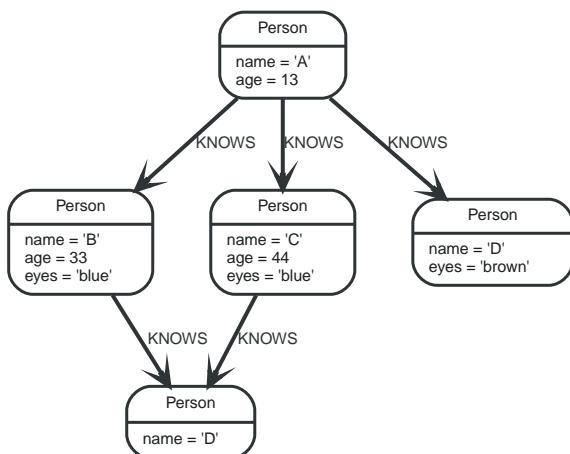


Figure 24. Graph

8.3.1. `avg()`

`avg()` returns the average of a set of numeric values.

Syntax: `avg(expression)`

Returns:

Either an Integer or a Float, depending on the values returned by `expression` and whether or not the calculation overflows.

Arguments:

Name	Description
<code>expression</code>	An expression returning a set of numeric values.

Considerations:

Any `null` values are excluded from the calculation.

`avg(null)` returns `null`.

Query

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

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

Table 204. Result

<code>avg(n.age)</code>
30.0
1 row

8.3.2. collect()

`collect()` returns a list containing the values returned by an expression. Using this function aggregates data by amalgamating multiple records or values into a single list.

Syntax: `collect(expression)`

Returns:

A list containing heterogeneous elements; the types of the elements are determined by the values returned by `expression`.

Arguments:

Name	Description
<code>expression</code>	An expression returning a set of values.

Considerations:

Any `null` values are ignored and will not be added to the list.

`collect(null)` returns an empty list.

Query

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

All the values are collected and returned in a single list.

Table 205. Result

collect(n.age)
JavaListWrapper(13, 33, 44)
1 row

8.3.3. count()

`count()` returns the number of values or rows, and appears in two variants:

- `count(*)` returns the number of matching rows, and
- `count(expr)` returns the number of non-`null` values returned by an expression.

Syntax: `count(expression)`

Returns:

An Integer.

Arguments:

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

Considerations:

`count(*)` includes rows returning `null`.

`count(expr)` ignores `null` values.

`count(null)` returns 0.

Using `count(*)` to return the number of nodes

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

Query

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

The labels and `age` property of the start node `n` and the number of nodes related to `n` are returned.

Table 206. Result

labels(n)	n.age	count(*)
JavaListWrapper(Person)	13	3
1 row		

Using `count(*)` to group and count relationship types

`count(*)` can be used to group relationship types and return the number.

Query

```
MATCH (n { name: 'A' })-[r]->()
RETURN type(r), count(*)
```

The relationship types and their group count are returned.

Table 207. Result

type(r)	count(*)
"KNOWS"	3
1 row	

Using `count(expression)` to return the number of values

Instead of simply returning the number of rows with `count(*)`, it may be more useful to return the actual number of values returned by an expression.

Query

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

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

Table 208. Result

count(x)
3
1 row

Counting non-`null` values

`count(expression)` can be used to return the number of non-`null` values returned by the expression.

Query

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

The number of `:Person` nodes having an `age` property is returned.

Table 209. Result

count(n.age)
3
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 count a `friend_of_friend` once, as `DISTINCT` removes the duplicates.
- The second aggregate function, `count(friend_of_friend)`, will consider the same `friend_of_friend`

multiple times.

Query

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

Both **B** and **C** know **D** and thus **D** will get counted twice when not using **DISTINCT**.

Table 210. Result

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

8.3.4. max()

max() returns the maximum value in a set of values.

Syntax: `max(expression)`

Returns:

A **property type**, or a list, depending on the values returned by **expression**.

Arguments:

Name	Description
<code>expression</code>	An expression returning a set containing any combination of property types and lists thereof.

Considerations:

Any **null** values are excluded from the calculation.

In a mixed set, any numeric value is always considered to be higher than any string value, and any string value is always considered to be higher than any list.

Lists are compared in dictionary order, i.e. list elements are compared pairwise in ascending order from the start of the list to the end.

`max(null)` returns **null**.

Query

```
UNWIND [1, 'a', NULL , 0.2, 'b', '1', '99'] AS val
RETURN max(val)
```

The highest of all the values in the mixed set — in this case, the numeric value **1** — is returned. Note that the (string) value **"99"**, which may *appear* at first glance to be the highest value in the list, is considered to be a lower value than **1** as the latter is a string.

Table 211. Result

max(val)
1
1 row

Query

```
UNWIND [[1, 'a', 89],[1, 2]] AS val  
RETURN max(val)
```

The highest of all the lists in the set — in this case, the list [1, 2] — is returned, as the number 2 is considered to be a higher value than the string "a", even though the list [1, 'a', 89] contains more elements.

Table 212. Result

max(val)
JavaListWrapper(1, 2)
1 row

Query

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

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

Table 213. Result

max(n.age)
44
1 row

8.3.5. `min()`

`min()` returns the minimum value in a set of values.

Syntax: `min(expression)`

Returns:

A [property type](#), or a list, depending on the values returned by `expression`.

Arguments:

Name	Description
<code>expression</code>	An expression returning a set containing any combination of property types and lists thereof.

Considerations:

Any `null` values are excluded from the calculation.

In a mixed set, any string value is always considered to be lower than any numeric value, and any list is always considered to be lower than any string.

Lists are compared in dictionary order, i.e. list elements are compared pairwise in ascending order from the start of the list to the end.

`min(null)` returns `null`.

Query

```
UNWIND [1, 'a', NULL , 0.2, 'b', '1', '99'] AS val  
RETURN min(val)
```

The lowest of all the values in the mixed set — in this case, the string value "`1`" — is returned. Note that the (numeric) value `0.2`, which may *appear* at first glance to be the lowest value in the list, is considered to be a higher value than "`1`" as the latter is a string.

Table 214. Result

min(val)
<code>"1"</code>
1 row

Query

```
UNWIND ['d',[1, 2],['a', 'c', 23]] AS val  
RETURN min(val)
```

The lowest of all the values in the set — in this case, the list `['a', 'c', 23]` — is returned, as (i) the two lists are considered to be lower values than the string "`d`", and (ii) the string "`a`" is considered to be a lower value than the numerical value `1`.

Table 215. Result

min(val)
<code>JavaListWrapper(a, c, 23)</code>
1 row

Query

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

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

Table 216. Result

min(n.age)
<code>13</code>
1 row

8.3.6. percentileCont()

`percentileCont()` returns the percentile of the 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)`

Returns:

A Float.

Arguments:

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

Considerations:

Any `null` values are excluded from the calculation.

`percentileCont(null, percentile)` returns `null`.

Query

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

The 40th percentile of the values in the property `age` is returned, calculated with a weighted average. In this case, 0.4 is the median, or 40th percentile.

Table 217. Result

<code>percentileCont(n.age, 0.4)</code>
29.0
1 row

8.3.7. percentileDisc()

`percentileDisc()` returns the percentile of the given value over a group, with a percentile from 0.0 to 1.0. It uses a rounding method and calculates the nearest value to the percentile. For interpolated values, see `percentileCont`.

Syntax: `percentileDisc(expression, percentile)`

Returns:

Either an Integer or a Float, depending on the values returned by `expression` and whether or not the calculation overflows.

Arguments:

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

Considerations:

Any `null` values are excluded from the calculation.

`percentileDisc(null, percentile)` returns `null`.

Query

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

The 50th percentile of the values in the property `age` is returned.

Table 218. Result

<code>percentileDisc(n.age, 0.5)</code>
33
1 row

8.3.8. `stDev()`

`stDev()` returns the standard deviation for the 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)`

Returns:

A Float.

Arguments:

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

Considerations:

Any `null` values are excluded from the calculation.

`stDev(null)` returns 0.

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 219. Result

<code>stDev(n.age)</code>
15.716233645501712
1 row

8.3.9. `stDevP()`

`stDevP()` returns the standard deviation for the 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)`

Returns:

A Float.

Arguments:

Name	Description
expression	A numeric expression.

Considerations:

Any `null` values are excluded from the calculation.

`stDevP(null)` returns `0`.

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.

Table 220. Result

stDevP(n.age)
12.832251036613439
1 row

8.3.10. sum()

`sum()` returns the sum of a set of numeric values.

Syntax: `sum(expression)`

Returns:

Either an Integer or a Float, depending on the values returned by `expression`.

Arguments:

Name	Description
expression	An expression returning a set of numeric values.

Considerations:

Any `null` values are excluded from the calculation.

`sum(null)` returns `0`.

Query

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

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

Table 221. 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](#).



The function `rels()` has been superseded by `relationships()`, and will be removed in a future release.

Functions:

- [extract\(\)](#)
- [filter\(\)](#)
- [keys\(\)](#)
- [labels\(\)](#)
- [nodes\(\)](#)
- [range\(\)](#)
- [reduce\(\)](#)
- [relationships\(\)](#)
- [reverse\(\)](#)
- [tail\(\)](#)

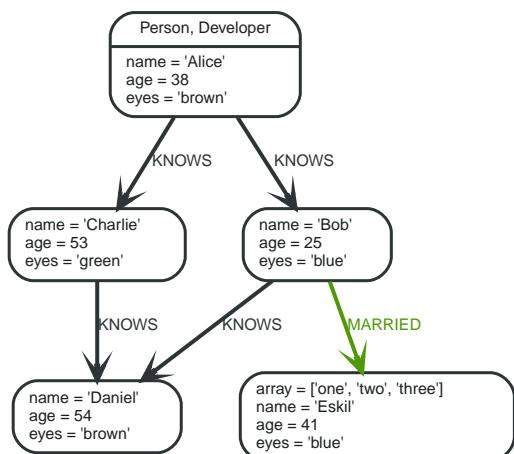


Figure 25. Graph

8.4.1. extract()

`extract()` returns a list `lresult` containing the values resulting from an expression which has been applied to each element in a list `list`. This function is analogous to the `map` method in functional languages such as Lisp and Scala.

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

Returns:

A list containing heterogeneous elements; the types of the elements are determined by `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. We decide here which variable to use.
<code>expression</code>	This expression will run once per value in <code>list</code> , and add it to the list which is returned by <code>extract()</code> .

Considerations:

Any `null` values in `list` are preserved.

Common usages of `extract()` include:

- Returning a property from a list of nodes or relationships; for example, `expression = n.prop` and `list = nodes(<some-path>)`.
- Returning the result of the application of a function on each element in a list; for example, `expression = toUpper(x)` and `variable = x`.

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 path `p` are returned.

Table 222. Result

<code>extracted</code>
<code>JavaListWrapper(38, 25, 54)</code>
1 row

8.4.2. filter()

`filter()` returns a list `lresult` containing all the elements from a list `list` that comply with the given predicate.

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

Returns:

A list containing heterogeneous elements; the types of the elements are determined by the elements in `list`.

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 elements in <code>list</code> .

Query

```
MATCH (a)
WHERE a.name = 'Eskil'
RETURN a.array, filter(x IN a.array WHERE size(x)= 3)
```

The property named `array` and a list of all values having size '3' are returned.

Table 223. Result

a.array	filter(x IN a.array WHERE size(x)= 3)
[Ljava.lang.String;@6079ac84	JavaListWrapper(one, two)
1 row	

8.4.3. keys()

`keys` returns a list containing the string representations for all the property names of a node, relationship, or map.

Syntax: `keys(expression)`

Returns:

A list containing String elements.

Arguments:

Name	Description
expression	An expression that returns a node, a relationship, or a map.

Considerations:

`keys(null)` returns `null`.

Query

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

A list containing the names of all the properties on the node bound to `a` is returned.

Table 224. Result

keys(a)
JavaListWrapper(name, age, eyes)
1 row

8.4.4. labels()

`labels` returns a list containing the string representations for all the labels of a node.

Syntax: `labels(node)`

Returns:

A list containing String elements.

Arguments:

Name	Description
node	An expression that returns a single node.

Considerations:

`labels(null)` returns `null`.

Query

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

A list containing all the labels of the node bound to `a` is returned.

Table 225. Result

<code>labels(a)</code>
<code>JavaListWrapper(Person, Developer)</code>
1 row

8.4.5. nodes()

`nodes()` returns a list containing all the nodes in a path.

Syntax: `nodes(path)`

Returns:

A list containing Node elements.

Arguments:

Name	Description
path	An expression that returns a path.

Considerations:

`nodes(null)` returns `null`.

Query

```
MATCH p =(a)-->(b)-->(c)
WHERE a.name = 'Alice' AND c.name = 'Eskil'
RETURN nodes(p)
```

A list containing all the nodes in the path `p` is returned.

Table 226. Result

nodes(p)

JavaListWrapper(Node[0], Node[1], Node[4])
--

1 row

8.4.6. range()

`range()` returns a list comprising all integer values within a range bounded by a start value `start` and end value `end`, where the difference `step` between any two consecutive values is constant; i.e. an arithmetic progression. The range is inclusive, and the arithmetic progression will therefore always contain `start` and — depending on the values of `start`, `step` and `end` — `end`.

Syntax: `range(start, end [, step])`

Returns:

A list of Integer elements.

Arguments:

Name	Description
<code>start</code>	An expression that returns an integer value.
<code>end</code>	An expression that returns an integer value.
<code>step</code>	A numeric expression defining the difference between any two consecutive values, with a default of <code>1</code> .

Query

RETURN range(0, 10), range(2, 18, 3)

Two lists of numbers in the given ranges are returned.

Table 227. Result

<code>range(0, 10)</code>	<code>range(2, 18, 3)</code>
<code>JavaListWrapper(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)</code>	<code>JavaListWrapper(2, 5, 8, 11, 14, 17)</code>
1 row	

8.4.7. reduce()

`reduce()` returns the value resulting from the application of an expression on each successive element in a list in conjunction with the result of the computation thus far. This function will iterate through each element `e` in the given list, run the expression on `e` — taking into account the current partial result — and store the new partial result in the accumulator. This function is analogous to the `fold` or `reduce` method in functional languages such as Lisp and Scala.

Syntax: `reduce(accumulator = initial, variable IN list | expression)`

Returns:

The type of the value returned depends on the arguments provided, along with the semantics of <code>expression</code> .

Arguments:

Name	Description
accumulator	A variable that will hold the result and the partial results as the list is iterated.
initial	An expression that runs once to give a starting value to the accumulator.
list	An expression that returns a list.
variable	The closure will have a variable introduced in its context. We decide here which variable to use.
expression	This expression will run once per value in the list, and produce 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 228. Result

reduction
117
1 row

8.4.8. relationships()

`relationships()` returns a list containing all the relationships in a path.

Syntax: `relationships(path)`

Returns:

A list containing Relationship elements.

Arguments:

Name	Description
path	An expression that returns a path.

Considerations:

`relationships(null)` returns `null`.

Query

```
MATCH p =(a)-->(b)-->(c)
WHERE a.name = 'Alice' AND c.name = 'Eskil'
RETURN relationships(p)
```

A list containing all the relationships in the path `p` is returned.

Table 229. Result

```
relationships(p)
```

```
JavaListWrapper((0)-[KNOWS,0]->(1), (1)-[MARRIED,4]->(4))
```

```
1 row
```

8.4.9. reverse()

`reverse()` returns a list in which the order of all elements in the original list have been reversed.

Syntax: `reverse(original)`

Returns:

A list containing homogeneous or heterogeneous elements; the types of the elements are determined by the elements within `original`.

Arguments:

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

Considerations:

Any `null` element in `original` is preserved.

Query

```
WITH [4923,'abc',521, NULL , 487] AS ids
RETURN reverse(ids)
```

Table 230. Result

```
reverse(ids)
```

```
JavaListWrapper(487, null, 521, abc, 4923)
```

```
1 row
```

8.4.10. tail()

`tail()` returns a list `lresult` containing all the elements, excluding the first one, from a list `list`.

Syntax: `tail(list)`

Returns:

A list containing heterogeneous elements; the types of the elements are determined by the elements in `list`.

Arguments:

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

Query

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

The property named `array` and a list comprising all but the first element of the `array` property are returned.

Table 231. Result

a.array	tail(a.array)
[Ljava.lang.String;@6c9dc2b0	JavaListWrapper[two, three]
1 row	

8.5. Mathematical functions - numeric

These functions all operate on numeric 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:

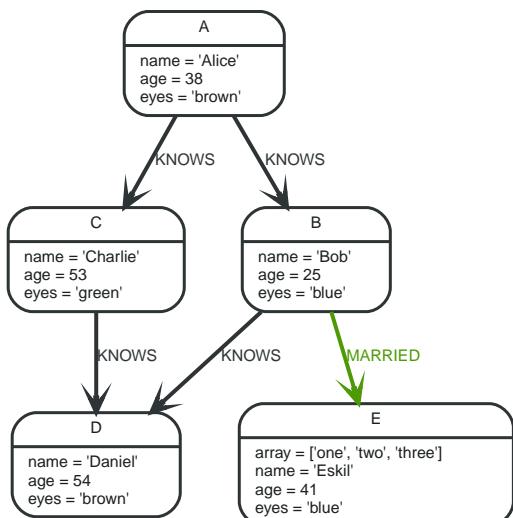


Figure 26. Graph

8.5.1. abs()

`abs()` returns the absolute value of the given number.

Syntax: `abs(expression)`

Returns:

The type of the value returned will be that of `expression`.

Arguments:

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

Considerations:

`abs(null)` returns `null`.

If `expression` is negative, `-(expression)` (i.e. the *negation* of `expression`) is returned.

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 232. Result

a.age	e.age	abs(a.age - e.age)
38	41	3
1 row		

8.5.2. `ceil()`

`ceil()` returns the smallest floating point number that is greater than or equal to the given number and equal to a mathematical integer.

Syntax: `ceil(expression)`

Returns:

A Float.

Arguments:

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

Considerations:

`ceil(null)` returns `null`.

Query

```
RETURN ceil(0.1)
```

The ceil of `0.1` is returned.

Table 233. Result

<code>ceil(0.1)</code>
<code>1.0</code>
1 row

8.5.3. floor()

`floor()` returns the largest floating point number that is less than or equal to the given number and equal to a mathematical integer.

Syntax: `floor(expression)`

Returns:

A Float.

Arguments:

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

Considerations:

`floor(null)` returns `null`.

Query

```
RETURN floor(0.9)
```

The floor of `0.9` is returned.

Table 234. Result

<code>floor(0.9)</code>
<code>0.0</code>
1 row

8.5.4. rand()

`rand()` returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e. $[0, 1]$. The numbers returned follow an approximate uniform distribution.

Syntax: `rand()`

Returns:

A Float.

Query

```
RETURN rand()
```

A random number is returned.

Table 235. Result

rand()
0.3014963988046283
1 row

8.5.5. round()

`round()` returns the value of the given number rounded to the nearest integer.

Syntax: `round(expression)`

Returns:

A Float.

Arguments:

Name	Description
expression	A numeric expression.

Considerations:

`round(null)` returns `null`.

Query

```
RETURN round(3.141592)
```

`3.0` is returned.

Table 236. Result

round(3.141592)
3.0
1 row

8.5.6. sign()

`sign()` returns the signum of the given number: `0` if the number is `0`, `-1` for any negative number, and `1` for any positive number.

Syntax: `sign(expression)`

Returns:

An Integer.

Arguments:

Name	Description
expression	A numeric expression.

Considerations:

```
sign(null) returns null.
```

Query

```
RETURN sign(-17), sign(0.1)
```

The signs of `-17` and `0.1` are returned.

Table 237. Result

sign(-17)	sign(0.1)
-1	1
1 row	

8.6. Mathematical functions - logarithmic

These functions all operate on numeric 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()`

Returns:

```
A Float.
```

Query

```
RETURN e()
```

The base of the natural logarithm, `e`, is returned.

Table 238. 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)`

Returns:

A Float.

Arguments:

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

Considerations:

`exp(null)` returns `null`.

Query

```
RETURN exp(2)
```

e to the power of 2 is returned.

Table 239. Result

<code>exp(2)</code>
<code>7.38905609893065</code>
1 row

8.6.3. log()

`log()` returns the natural logarithm of a number.

Syntax: `log(expression)`

Returns:

A Float.

Arguments:

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

Considerations:

`log(null)` returns `null`.

`log(0)` returns `null`.

Query

```
RETURN log(27)
```

The natural logarithm of 27 is returned.

Table 240. Result

log(27)
3.295836866004329
1 row

8.6.4. log10()

`log10()` returns the common logarithm (base 10) of a number.

Syntax: `log10(expression)`

Returns:

A Float.

Arguments:

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

Considerations:

<code>log10(null)</code> returns <code>null</code> .
--

<code>log10(0)</code> returns <code>null</code> .

Query

```
RETURN log10(27)
```

The common logarithm of 27 is returned.

Table 241. Result

log10(27)
1.4313637641589874
1 row

8.6.5. sqrt()

`sqrt()` returns the square root of a number.

Syntax: `sqrt(expression)`

Returns:

A Float.

Arguments:

Name	Description
expression	A numeric expression.

Considerations:

`sqrt(null)` returns `null`.

`sqrt(<any negative number>)` returns `null`

Query

```
RETURN sqrt(256)
```

The square root of `256` is returned.

Table 242. Result

<code>sqrt(256)</code>
<code>16.0</code>
1 row

8.7. Mathematical functions - trigonometric

These functions all operate on numeric 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\(\)](#)

8.7.1. acos()

`acos()` returns the arccosine of a number in radians.

Syntax: `acos(expression)`

Returns:

A Float.

Arguments:

Name	Description
expression	A numeric expression that represents the angle in radians.

Considerations:

`acos(null)` returns `null`.

If `(expression < -1)` or `(expression > 1)`, then `(acos(expression))` returns `null`.

Query

```
RETURN acos(0.5)
```

The arccosine of `0.5` is returned.

Table 243. Result

<code>acos(0.5)</code>
<code>1.0471975511965979</code>
1 row

8.7.2. asin()

`asin()` returns the arcsine of a number in radians.

Syntax: `asin(expression)`

Returns:

A Float.

Arguments:

Name	Description
expression	A numeric expression that represents the angle in radians.

Considerations:

`asin(null)` returns `null`.

If `(expression < -1)` or `(expression > 1)`, then `(asin(expression))` returns `null`.

Query

```
RETURN asin(0.5)
```

The arcsine of `0.5` is returned.

Table 244. Result

<code>asin(0.5)</code>
0.5235987755982989
1 row

8.7.3. atan()

`atan()` returns the arctangent of a number in radians.

Syntax: `atan(expression)`

Returns:

A Float.

Arguments:

Name	Description
<code>expression</code>	A numeric expression that represents the angle in radians.

Considerations:

`atan(null)` returns `null`.

Query

```
RETURN atan(0.5)
```

The arctangent of `0.5` is returned.

Table 245. Result

<code>atan(0.5)</code>
0.4636476090008061
1 row

8.7.4. atan2()

`atan2()` returns the arctangent2 of a set of coordinates in radians.

Syntax: `atan2(expression1, expression2)`

Returns:

A Float.

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.

Considerations:

`atan2(null, null)`, `atan2(null, expression2)` and `atan(expression1, null)` all return `null`.

Query

```
RETURN atan2(0.5, 0.6)
```

The arctangent2 of `0.5` and `0.6` is returned.

Table 246. Result

<code>atan2(0.5, 0.6)</code>
<code>0.6947382761967033</code>
1 row

8.7.5. cos()

`cos()` returns the cosine of a number.

Syntax: `cos(expression)`

Returns:

A Float.

Arguments:

Name	Description
<code>expression</code>	A numeric expression that represents the angle in radians.

Considerations:

`cos(null)` returns `null`.

Query

```
RETURN cos(0.5)
```

The cosine of `0.5` is returned.

Table 247. Result

<code>cos(0.5)</code>
<code>0.8775825618903728</code>
1 row

8.7.6. cot()

`cot()` returns the cotangent of a number.

Syntax: `cot(expression)`

Returns:

A Float.

Arguments:

Name	Description
expression	A numeric expression that represents the angle in radians.

Considerations:

`cot(null)` returns `null`.

`cot(0)` returns `null`.

Query

```
RETURN cot(0.5)
```

The cotangent of `0.5` is returned.

Table 248. Result

<code>cot(0.5)</code>
<code>1.830487721712452</code>
1 row

8.7.7. degrees()

`degrees()` converts radians to degrees.

Syntax: `degrees(expression)`

Returns:

A Float.

Arguments:

Name	Description
expression	A numeric expression that represents the angle in radians.

Considerations:

`degrees(null)` returns `null`.

Query

```
RETURN degrees(3.14159)
```

The number of degrees in something close to π is returned.

Table 249. Result

<code>degrees(3.14159)</code>
<code>179.99984796050427</code>

```
degrees(3.14159)
```

1 row

8.7.8. haversin()

`haversin()` returns half the versine of a number.

Syntax: `haversin(expression)`

Returns:

A Float.

Arguments:

Name	Description
<code>expression</code>	A numeric expression that represents the angle in radians.

Considerations:

`haversin(null)` returns `null`.

Query

```
RETURN haversin(0.5)
```

The haversine of `0.5` is returned.

Table 250. 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 251. 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()`

Returns:

A Float.

Query

```
RETURN pi()
```

The constant *pi* is returned.

Table 252. Result

pi()

3.141592653589793

1 row

8.7.11. radians()

`radians()` converts degrees to radians.

Syntax: `radians(expression)`

Returns:

A Float.

Arguments:

Name	Description
<code>expression</code>	A numeric expression that represents the angle in degrees.

Considerations:

`radians(null)` returns `null`.

Query

```
RETURN radians(180)
```

The number of radians in `180` degrees is returned (*pi*).

Table 253. Result

```
radians(180)
```

```
3.141592653589793
```

```
1 row
```

8.7.12. sin()

`sin()` returns the sine of a number.

Syntax: `sin(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
<code>expression</code>	A numeric expression that represents the angle in radians.

Considerations:

```
sin(null) returns null.
```

Query

```
RETURN sin(0.5)
```

The sine of `0.5` is returned.

Table 254. Result

```
sin(0.5)
```

```
0.479425538604203
```

```
1 row
```

8.7.13. tan()

`tan()` returns the tangent of a number.

Syntax: `tan(expression)`

Returns:

```
A Float.
```

Arguments:

Name	Description
<code>expression</code>	A numeric expression that represents the angle in radians.

Considerations:

```
tan(null) returns null.
```

Query

```
RETURN tan(0.5)
```

The tangent of `0.5` is returned.

Table 255. Result

<code>tan(0.5)</code>
<code>0.5463024898437905</code>
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, booleans and temporal values (i.e. `Date`, `Time`, `LocalTime`, `DateTime`, `LocalDateTime` or `Duration` values).

Functions taking a string as input all operate on *Unicode characters* rather than on a standard `char[]`. For example, `size(s)`, where `s` is a character in the Chinese alphabet, will return 1.



The functions `lower()` and `upper()` have been superseded by `toLower()` and `toUpper()`, respectively, and will be removed in a future release.



When `toString()` is applied to a temporal value, it returns a string representation suitable for parsing by the corresponding [temporal functions](#). This string will therefore be formatted according to the [ISO 8601](#) (https://en.wikipedia.org/wiki/ISO_8601) format.

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 specified number of leftmost characters of the original string.

Syntax: `left(original, length)`

Returns:

A String.

Arguments:

Name	Description
<code>original</code>	An expression that returns a string.
<code>n</code>	An expression that returns a positive integer.

Considerations:

`left(null, length)` and `left(null, null)` both return `null`

`left(original, null)` will raise an error.

If `length` is not a positive integer, an error is raised.

If `length` exceeds the size of `original`, `original` is returned.

Query

```
RETURN left('hello', 3)
```

Table 256. Result

<code>left('hello', 3)</code>
"hel"
1 row

8.8.2. ltrim()

`lTrim()` returns the original string with leading whitespace removed.

Syntax: `lTrim(original)`

Returns:

A String.

Arguments:

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

Considerations:

`lTrim(null)` returns `null`

Query

```
RETURN lTrim('  hello')
```

Table 257. Result

lTrim(' hello')
"hello"
1 row

8.8.3. replace()

`replace()` returns a string in which all occurrences of a specified string in the original string have been replaced by another (specified) string.

Syntax: `replace(original, search, replace)`

Returns:

A String.

Arguments:

Name	Description
original	An expression that returns a string.
search	An expression that specifies the string to be replaced in <code>original</code> .
replace	An expression that specifies the replacement string.

Considerations:

If any argument is <code>null</code> , <code>null</code> will be returned.
--

If <code>search</code> is not found in <code>original</code> , <code>original</code> will be returned.
--

Query

```
RETURN replace("hello", "l", "w")
```

Table 258. Result

replace("hello", "l", "w")
"hewwo"
1 row

8.8.4. reverse()

`reverse()` returns a string in which the order of all characters in the original string have been reversed.

Syntax: `reverse(original)`

Returns:

A String.

Arguments:

Name	Description
original	An expression that returns a string.

Considerations:

`reverse(null)` returns `null`.

Query

```
RETURN reverse('anagram')
```

Table 259. Result

reverse('anagram')
"margana"
1 row

8.8.5. right()

`right()` returns a string containing the specified number of rightmost characters of the original string.

Syntax: `right(original, length)`

Returns:

A String.

Arguments:

Name	Description
original	An expression that returns a string.
n	An expression that returns a positive integer.

Considerations:

`right(null, length)` and `right(null, null)` both return `null`

`right(original, null)` will raise an error.

If `length` is not a positive integer, an error is raised.

If `length` exceeds the size of `original`, `original` is returned.

Query

```
RETURN right('hello', 3)
```

Table 260. Result

right('hello', 3)
"llo"
1 row

8.8.6. rtrim()

`rTrim()` returns the original string with trailing whitespace removed.

Syntax: `rTrim(original)`

Returns:

A String.

Arguments:

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

Considerations:

`rTrim(null)` returns `null`

Query

```
RETURN rTrim('hello   ')
```

Table 261. Result

<code>rTrim('hello ')</code>
<code>"hello"</code>
1 row

8.8.7. split()

`split()` returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.

Syntax: `split(original, splitDelimiter)`

Returns:

A list of Strings.

Arguments:

Name	Description
<code>original</code>	An expression that returns a string.
<code>splitDelimiter</code>	The string with which to split <code>original</code> .

Considerations:

`split(null, splitDelimiter)` and `split(original, null)` both return `null`

Query

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

Table 262. Result

split('one,two', ',')
JavaListWrapper(one, two)
1 row

8.8.8. substring()

`substring()` returns a substring of the original string, beginning with a 0-based index start and length.

Syntax: `substring(original, start [, length])`

Returns:

A String.

Arguments:

Name	Description
<code>original</code>	An expression that returns a string.
<code>start</code>	An expression that returns a positive integer, denoting the position at which the substring will begin.
<code>length</code>	An expression that returns a positive integer, denoting how many characters of <code>original</code> will be returned.

Considerations:

<code>start</code> uses a zero-based index.
If <code>length</code> is omitted, the function returns the substring starting at the position given by <code>start</code> and extending to the end of <code>original</code> .
If <code>original</code> is <code>null</code> , <code>null</code> is returned.
If either <code>start</code> or <code>length</code> is <code>null</code> or a negative integer, an error is raised.
If <code>start</code> is <code>0</code> , the substring will start at the beginning of <code>original</code> .
If <code>length</code> is <code>0</code> , the empty string will be returned.

Query

```
RETURN substring('hello', 1, 3), substring('hello', 2)
```

Table 263. 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)`

Returns:

A String.

Arguments:

Name	Description
original	An expression that returns a string.

Considerations:

`toLowerCase(null)` returns `null`

Query

```
RETURN toLower('HELLO')
```

Table 264. Result

<code>toLowerCase('HELLO')</code>
"hello"
1 row

8.8.10. `toString()`

`toString()` converts an integer, float or boolean value to a string.

Syntax: `toString(expression)`

Returns:

A String.

Arguments:

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

Considerations:

`toString(null)` returns `null`

If `expression` is a string, it will be returned unchanged.

Query

```
RETURN toString(11.5), toString('already a string'), toString(TRUE ), toString(date({ year:1984, month:10, day:11 })) AS dateString, toString(datetime({ year:1984, month:10, day:11, hour:12, minute:31, second:14, millisecond: 341, timezone: 'Europe/Stockholm' })) AS datetimeString, toString(duration({ minutes: 12, seconds: -60 })) AS durationString
```

Table 265. Result

toString(11.5)	toString('already a string')	toString(TRUE)	dateString	datetimeString	durationString
"11.5"	"already a string"	"true"	"1984-10-11"	"1984-10-11T12:31:14.341+01:00[Europe/Stockholm]"	"PT11M"
1 row					

8.8.11. `toUpperCase()`

`toUpperCase()` returns the original string in uppercase.

Syntax: `toUpperCase(original)`

Returns:

A String.

Arguments:

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

Considerations:

`toUpperCase(null)` returns `null`

Query

```
RETURN toUpper('hello')
```

Table 266. Result

<code>toUpper('hello')</code>
"HELLO"
1 row

8.8.12. `trim()`

`trim()` returns the original string with leading and trailing whitespace removed.

Syntax: `trim(original)`

Returns:

A String.

Arguments:

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

Considerations:

```
trim(null) returns null
```

Query

```
RETURN trim(' hello ')
```

Table 267. Result

trim(' hello ')
"hello"
1 row

8.9. Temporal functions

Cypher provides functions allowing for the creation and manipulation of values for each temporal type — *Date*, *Time*, *LocalTime*, *DateTime*, *LocalDateTime* and *Duration*.



See also [Temporal \(Date/Time\) values](#) and [Temporal operators](#).

- [Temporal instant types \(*Date*, *Time*, *LocalTime*, *DateTime* and *LocalDateTime*\)](#)
 - An overview of temporal instant type creation
 - Controlling which clock to use
 - Truncating temporal values
- [Duration](#)
 - Creating a *Duration* from duration components
 - Creating a *Duration* from a string
 - Computing the *Duration* between two temporal instants
 - `duration.between()`
 - `duration.inMonths()`
 - `duration.inDays()`
 - `duration.inSeconds()`

8.9.1. Temporal instant types (*Date*, *Time*, *LocalTime*, *DateTime* and *LocalDateTime*)

An overview of temporal instant type creation

Each function bears the same name as the type, and construct the type they correspond to in one of four ways:

- Capturing the current time
- Composing the components of the type
- Parsing a string representation of the temporal value
- Selecting and composing components from another temporal value by
 - either combining temporal values (such as combining a *Date* with a *Time* to create a *DateTime*),

or

- selecting parts from a temporal value (such as selecting the *Date* from a *DateTime*); the *extractors* — groups of components which can be selected — are:
 - `date` — contains all components for a *Date* (conceptually *year*, *month* and *day*).
 - `time` — contains all components for a *Time* (*hour*, *minute*, *second*, and sub-seconds; namely *millisecond*, *microsecond* and *nanosecond*). If the type being created and the type from which the time component is being selected both contain `timezone` (and a `timezone` is not explicitly specified) the `timezone` is also selected.
 - `datetime` — selects all components, and is useful for overriding specific components. Analogously to `time`, if the type being created and the type from which the time component is being selected both contain `timezone` (and a `timezone` is not explicitly specified) the `timezone` is also selected.
- In effect, this allows for the *conversion* between different temporal types, and allowing for 'missing' components to be specified.

Table 268. Temporal instant type creation functions

Function	Date	Time	LocalTime	DateTime	LocalDateTime
Getting the current value	X	X	X	X	X
Creating a calendar-based (Year-Month-Day) value	X			X	X
Creating a week-based (Year-Week-Day) value	X			X	X
Creating a quarter-based (Year-Quarter-Day) value	X			X	X
Creating an ordinal (Year-Day) value	X			X	X
Creating a value from time components		X	X		
Creating a value from other temporal values using extractors (i.e. converting between different types)	X	X	X	X	X
Creating a value from a string	X	X	X	X	X
Creating a value from a timestamp				X	



All the temporal instant types — including those that do not contain time zone information support such as *Date*, *LocalTime* and *DateTime* — allow for a time zone to be specified for the functions that retrieve the current instant. This allows for the retrieval of the current instant in the specified time zone.

Controlling which clock to use

The functions which create temporal instant values based on the current instant use the `statement` clock as default. However, there are three different clocks available for more fine-grained control:

- `transaction`: The same instant is produced for each invocation within the same transaction. A different time may be produced for different transactions.
- `statement`: The same instant is produced for each invocation within the same statement. A different time may be produced for different statements within the same transaction.
- `realtime`: The instant produced will be the live clock of the system.

The following table lists the different sub-functions for specifying the clock to be used when creating the current temporal instant value:

Type	default	transaction	statement	realtime
Date	<code>date()</code>	<code>date.transaction()</code>	<code>date.statement()</code>	<code>date.realtime()</code>
Time	<code>time()</code>	<code>time.transaction()</code>	<code>time.statement()</code>	<code>time.realtime()</code>
LocalTime	<code>localtime()</code>	<code>localtime.transaction()</code>	<code>localtime.statement()</code>	<code>localtime.realtime()</code>
DateTime	<code>datetime()</code>	<code>datetime.transaction()</code>	<code>datetime.statement()</code>	<code>datetime.realtime()</code>
LocalDateTime	<code>localdatetime()</code>	<code>localdatetime.transaction()</code>	<code>localdatetime.statement()</code>	<code>localdatetime.realtime()</code>

Truncating temporal values

A temporal instant value can be created by truncating another temporal instant value at the nearest preceding point in time at a specified component boundary (namely, a *truncation unit*). A temporal instant value created in this way will have all components which are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of overriding the default values which would otherwise have been set for these less significant components.

The following truncation units are supported:

- `millennium`: Select the temporal instant corresponding to the *millenium* of the given instant.
- `century`: Select the temporal instant corresponding to the *century* of the given instant.
- `decade`: Select the temporal instant corresponding to the *decade* of the given instant.
- `year`: Select the temporal instant corresponding to the *year* of the given instant.
- `weekYear`: Select the temporal instant corresponding to the first day of the first week of the *week-year* of the given instant.
- `quarter`: Select the temporal instant corresponding to the *quarter of the year* of the given instant.
- `month`: Select the temporal instant corresponding to the *month* of the given instant.
- `week`: Select the temporal instant corresponding to the *week* of the given instant.
- `day`: Select the temporal instant corresponding to the *month* of the given instant.
- `hour`: Select the temporal instant corresponding to the *hour* of the given instant.
- `minute`: Select the temporal instant corresponding to the *minute* of the given instant.
- `second`: Select the temporal instant corresponding to the *second* of the given instant.
- `millisecond`: Select the temporal instant corresponding to the *millisecond* of the given instant.

- **microsecond**: Select the temporal instant corresponding to the *microsecond* of the given instant.

The following table lists the supported truncation units and the corresponding sub-functions:

Truncation unit	Date	Time	LocalTime	DateTime	LocalDateTime
millennium	date.truncate('millennium', input)			datetime.truncate('millennium', input)	localdatetime.truncate('millennium', input)
century	date.truncate('century', input)			datetime.truncate('century', input)	localdatetime.truncate('century', input)
decade	date.truncate('decade', input)			datetime.truncate('decade', input)	localdatetime.truncate('decade', input)
year	date.truncate('year', input)			datetime.truncate('year', input)	localdatetime.truncate('year', input)
weekYear	date.truncate('weekYear', input)			datetime.truncate('weekYear', input)	localdatetime.truncate('weekYear', input)
quarter	date.truncate('quarter', input)			datetime.truncate('quarter', input)	localdatetime.truncate('quarter', input)
month	date.truncate('month', input)			datetime.truncate('month', input)	localdatetime.truncate('month', input)
week	date.truncate('week', input)			datetime.truncate('week', input)	localdatetime.truncate('week', input)
day	date.truncate('day', input)	time.truncate('day', input)	localtime.truncate('day', input)	datetime.truncate('day', input)	localdatetime.truncate('day', input)
hour		time.truncate('hour', input)	localtime.truncate('hour', input)	datetime.truncate('hour', input)	localdatetime.truncate('hour', input)
minute		time.truncate('minute', input)	localtime.truncate('minute', input)	datetime.truncate('minute', input)	localdatetime.truncate('minute', input)
second		time.truncate('second', input)	localtime.truncate('second', input)	datetime.truncate('second', input)	localdatetime.truncate('second', input)
millisecond		time.truncate('millisecond', input)	localtime.truncate('millisecond', input)	datetime.truncate('millisecond', input)	localdatetime.truncate('millisecond', input)
microsecond		time.truncate('microsecond', input)	localtime.truncate('microsecond', input)	datetime.truncate('microsecond', input)	localdatetime.truncate('microsecond', input)

8.9.2. Duration

Information regarding specifying and accessing components of a *Duration* value can be found [here](#).

Creating a *Duration* from duration components

`duration()` can construct a *Duration* from a map of its components in the same way as the temporal instant types.

- **years**
- **quarters**
- **months**

- weeks
- days
- hours
- minutes
- seconds
- milliseconds
- microseconds
- nanoseconds

Syntax: `duration([{years, quarters, months, weeks, days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds}])`

Returns:

A Duration.

Arguments:

Name	Description
A single map consisting of the following:	
<code>years</code>	A numeric expression.
<code>quarters</code>	A numeric expression.
<code>months</code>	A numeric expression.
<code>weeks</code>	A numeric expression.
<code>days</code>	A numeric expression.
<code>hours</code>	A numeric expression.
<code>minutes</code>	A numeric expression.
<code>seconds</code>	A numeric expression.
<code>milliseconds</code>	A numeric expression.
<code>microseconds</code>	A numeric expression.
<code>nanoseconds</code>	A numeric expression.

Considerations:

At least one parameter must be provided (`duration()` and `duration({})` are invalid).

There is no constraint on how many of the parameters are provided.

It is possible to have a *Duration* where the amount of a smaller unit (e.g. `seconds`) exceeds the threshold of a larger unit (e.g. `days`).

The values of the parameters may be expressed as decimal fractions.

The values of the parameters may be arbitrarily large.

The values of the parameters may be negative.

Query

```
UNWIND [duration({ days: 14, hours:16, minutes: 12 }), duration({ months: 5, days: 1.5 }), duration({ months: 0.75 }), duration({ weeks: 2.5 }), duration({ minutes: 1.5, seconds: 1, milliseconds: 123, microseconds: 456, nanoseconds: 789 }), duration({ minutes: 1.5, seconds: 1, nanoseconds: 123456789 })] AS aDuration
RETURN aDuration
```

Table 269. Result

aDuration
P14DT16H12M
P5M1DT12H
P22DT19H51M49.5S
P17DT12H
PT1M31.123456789S
PT1M31.123456789S
6 rows

Creating a Duration from a string

`duration()` returns the *Duration* value obtained by parsing a string representation of a temporal amount.

Syntax: `duration(temporalAmount)`

Returns:

A Duration.

Arguments:

Name	Description
temporalAmount	A string representing a temporal amount.

Considerations:

`temporalAmount` must comply with either the [unit based form](#) or [date-and-time based form defined for Durations](#).

Query

```
UNWIND [duration("P14DT16H12M"), duration("P5M1.5D"), duration("P0.75M"), duration("PT0.75M"),
duration("P2012-02-02T14:37:21.545")] AS aDuration
RETURN aDuration
```

Table 270. Result

aDuration
P14DT16H12M
P5M1DT12H
P22DT19H51M49.5S
PT45S
P2012Y2M2DT14H37M21.545S

aDuration

5 rows

Computing the *Duration* between two temporal instants

`duration()` has sub-functions which compute the *logical difference* (in days, months, etc) between two temporal instant values:

- `duration.between(a, b)`: Computes the difference in multiple components between instant `a` and instant `b`. This captures month, days, seconds and sub-seconds differences separately.
- `duration.inMonths(a, b)`: Computes the difference in whole months (or quarters or years) between instant `a` and instant `b`. This captures the difference as the total number of months. Any difference smaller than a whole month is disregarded.
- `duration.inDays(a, b)`: Computes the difference in whole days (or weeks) between instant `a` and instant `b`. This captures the difference as the total number of days. Any difference smaller than a whole day is disregarded.
- `duration.inSeconds(a, b)`: Computes the difference in seconds (and fractions of seconds, or minutes or hours) between instant `a` and instant `b`. This captures the difference as the total number of seconds.

`duration.between()`

`duration.between()` returns the *Duration* value equal to the difference between the two given instants.

Syntax: `duration.between(instant1, instant2)`

Returns:

A Duration.

Arguments:

Name	Description
<code>instant₁</code>	An expression returning any temporal instant type (<i>Date</i> etc) that represents the starting instant.
<code>instant₂</code>	An expression returning any temporal instant type (<i>Date</i> etc) that represents the ending instant.

Considerations:

If `instant2` occurs earlier than `instant1`, the resulting *Duration* will be negative.

If `instant1` has a time component and `instant2` does not, the time component of `instant2` is assumed to be midnight, and vice versa.

If `instant1` has a time zone component and `instant2` does not, the time zone component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

If `instant1` has a date component and `instant2` does not, the date component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

Query

```
UNWIND [duration.between(date("1984-10-11"), date("1985-11-25")), duration.between(date("1985-11-25"),
date("1984-10-11")), duration.between(date("1984-10-11"), datetime("1984-10-12T21:40:32.142+0100")),
duration.between(date("2015-06-24"), localtime("14:30")), duration.between(localtime("14:30"),
time("16:30+0100")), duration.between(localdatetime("2015-07-21T21:40:32.142"), localdatetime("2016-07-
21T21:45:22.142")), duration.between(datetime({ year: 2017, month: 10, day: 29, hour: 0, timezone:
'Europe/Stockholm' }), datetime({ year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/London' })))] AS aDuration
RETURN aDuration
```

Table 271. Result

aDuration
P1Y1M14D
P-1Y-1M-14D
P1DT21H40M32.142S
PT14H30M
PT2H
P1YT4M50S
PT1H
7 rows

duration.inMonths()

`duration.inMonths()` returns the *Duration* value equal to the difference in whole months, quarters or years between the two given instants.

Syntax: `duration.inMonths(instant1, instant2)`

Returns:

A Duration.

Arguments:

Name	Description
instant ₁	An expression returning any temporal instant type (<i>Date</i> etc) that represents the starting instant.
instant ₂	An expression returning any temporal instant type (<i>Date</i> etc) that represents the ending instant.

Considerations:

If `instant2` occurs earlier than `instant1`, the resulting *Duration* will be negative.

If `instant1` has a time component and `instant2` does not, the time component of `instant2` is assumed to be midnight, and vice versa.

If `instant1` has a time zone component and `instant2` does not, the time zone component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

If `instant1` has a date component and `instant2` does not, the date component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

Any difference smaller than a whole month is disregarded.

Query

```
UNWIND [duration.inMonths(date("1984-10-11"), date("1985-11-25")), duration.inMonths(date("1985-11-25"),  
date("1984-10-11")), duration.inMonths(date("1984-10-11"), datetime("1984-10-12T21:40:32.142+0100")),  
duration.inMonths(date("2015-06-24"), localtime("14:30")), duration.inMonths(localdatetime("2015-07-  
21T21:40:32.142"), localdatetime("2016-07-21T21:45:22.142")), duration.inMonths(datetime({ year: 2017,  
month: 10, day: 29, hour: 0, timezone: 'Europe/Stockholm' }), datetime({ year: 2017, month: 10, day: 29,  
hour: 0, timezone: 'Europe/London' })))] AS aDuration  
RETURN aDuration
```

Table 272. Result

aDuration
P1Y1M
P-1Y-1M
PT0S
PT0S
P1Y
PT0S
6 rows

duration.inDays()

`duration.inDays()` returns the *Duration* value equal to the difference in whole days or weeks between the two given instants.

Syntax: `duration.inDays(instant1, instant2)`

Returns:

A Duration.

Arguments:

Name	Description
instant ₁	An expression returning any temporal instant type (<i>Date</i> etc) that represents the starting instant.
instant ₂	An expression returning any temporal instant type (<i>Date</i> etc) that represents the ending instant.

Considerations:

If `instant2` occurs earlier than `instant1`, the resulting *Duration* will be negative.

If `instant1` has a time component and `instant2` does not, the time component of `instant2` is assumed to be midnight, and vice versa.

If `instant1` has a time zone component and `instant2` does not, the time zone component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

If `instant1` has a date component and `instant2` does not, the date component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

Any difference smaller than a whole day is disregarded.

Query

```
UNWIND [duration.inDays(date("1984-10-11"), date("1985-11-25")), duration.inDays(date("1985-11-25"),
date("1984-10-11")), duration.inDays(date("1984-10-11"), datetime("1984-10-12T21:40:32.142+0100")),
duration.inDays(date("2015-06-24"), localtime("14:30")), duration.inDays(localdatetime("2015-07-
21T21:40:32.142"), localdatetime("2016-07-21T21:45:22.142")), duration.inDays(datetime({ year: 2017,
month: 10, day: 29, hour: 0, timezone: 'Europe/Stockholm' }), datetime({ year: 2017, month: 10, day: 29,
hour: 0, timezone: 'Europe/London' }))) AS aDuration
RETURN aDuration
```

Table 273. Result

aDuration
P410D
P-410D
P1D
PT0S
P366D
PT0S
6 rows

duration.inSeconds()

`duration.inSeconds()` returns the *Duration* value equal to the difference in seconds and fractions of seconds, or minutes or hours, between the two given instants.

Syntax: `duration.inSeconds(instant1, instant2)`

Returns:

A Duration.

Arguments:

Name	Description
instant ₁	An expression returning any temporal instant type (<i>Date</i> etc) that represents the starting instant.
instant ₂	An expression returning any temporal instant type (<i>Date</i> etc) that represents the ending instant.

Considerations:

If `instant2` occurs earlier than `instant1`, the resulting *Duration* will be negative.

If `instant1` has a time component and `instant2` does not, the time component of `instant2` is assumed to be midnight, and vice versa.

If `instant1` has a time zone component and `instant2` does not, the time zone component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

If `instant1` has a date component and `instant2` does not, the date component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

Query

```
UNWIND [duration.inSeconds(date("1984-10-11"), date("1984-10-12")), duration.inSeconds(date("1984-10-12"), date("1984-10-11")), duration.inSeconds(date("1984-10-11"), datetime("1984-10-12T01:00:32.142+0100")), duration.inSeconds(date("2015-06-24"), localtime("14:30")), duration.inSeconds(datetime({ year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/Stockholm' }), datetime({ year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/London' })))] AS aDuration
RETURN aDuration
```

Table 274. Result

aDuration
PT24H
PT-24H
PT25H32.142S
PT14H30M
PT1H
5 rows

8.9.3. date(): getting the current Date

`date()` returns the current *Date* value. If no time zone parameter is specified, the local time zone will be used.

Syntax: `date([{timezone}])`

Returns:

A Date.

Arguments:

Name	Description
A single map consisting of the following:	
timezone	A string expression that represents the time zone

Considerations:

If no parameters are provided, `date()` must be invoked (`date({})` is invalid).

Query

```
RETURN date() AS currentDate
```

The current date is returned.

Table 275. Result

currentDate
2018-06-20
1 row

Query

```
RETURN date({ timezone: 'America/Los Angeles' }) AS currentDateInLA
```

The current date in California is returned.

Table 276. Result

currentDateInLA
2018-06-20
1 row

date.transaction()

`date.transaction()` returns the current *Date* value using the `transaction` clock. This value will be the same for each invocation within the same transaction. However, a different value may be produced for different transactions.

Syntax: `date.transaction([{timezone}])`

Returns:

A Date.

Arguments:

Name	Description
timezone	A string expression that represents the time zone

Query

```
RETURN date.transaction() AS currentDate
```

Table 277. Result

currentDate
2018-06-20
1 row

date.statement()

`date.statement()` returns the current *Date* value using the `statement` clock. This value will be the same for each invocation within the same statement. However, a different value may be produced for different statements within the same transaction.

Syntax: `date.statement([{timezone}])`

Returns:

A Date.

Arguments:

Name	Description
timezone	A string expression that represents the time zone

Query

```
RETURN date.statement() AS currentDate
```

Table 278. Result

currentDate
2018-06-20
1 row

date.realtime()

`date.realtime()` returns the current *Date* value using the `realtime` clock. This value will be the live clock of the system.

Syntax: `date.realtime([{timezone}])`

Returns:

A Date.

Arguments:

Name	Description
timezone	A string expression that represents the time zone

Query

```
RETURN date.realtime() AS currentDate
```

Table 279. Result

currentDate
2018-06-20
1 row

Query

```
RETURN date.realtime('America/Los Angeles') AS currentDateInLA
```

Table 280. Result

currentDateInLA
2018-06-20
1 row

8.9.4. `date()`: creating a calendar (Year-Month-Day) *Date*

`date()` returns a *Date* value with the specified *year*, *month* and *day* component values.

Syntax: `date({year [, month, day]})`

Returns:

A Date.

Arguments:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>month</code>	An integer between 1 and 12 that specifies the month.
<code>day</code>	An integer between 1 and 31 that specifies the day of the month.

Considerations:

The `day of the month` component will default to 1 if `day` is omitted.

The `month` component will default to 1 if `month` is omitted.

If `month` is omitted, `day` must also be omitted.

Query

```
UNWIND [date({ year:1984, month:10, day:11 }), date({ year:1984, month:10 }), date({ year:1984 })] AS  
theDate  
RETURN theDate
```

Table 281. Result

theDate
1984-10-11
1984-10-01
1984-01-01
3 rows

8.9.5. `date()`: creating a week (Year-Week-Day) Date

`date()` returns a *Date* value with the specified *year*, *week* and *dayOfWeek* component values.

Syntax: `date({year [, week, dayOfWeek]})`

Returns:

A Date.

Arguments:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.

Name	Description
week	An integer between 1 and 53 that specifies the week.
dayOfWeek	An integer between 1 and 7 that specifies the day of the week.

Considerations:

The *day of the week* component will default to 1 if `dayOfWeek` is omitted.

The *week* component will default to 1 if `week` is omitted.

If `week` is omitted, `dayOfWeek` must also be omitted.

Query

```
UNWIND [date({ year:1984, week:10, dayOfWeek:3 }), date({ year:1984, week:10 }), date({ year:1984 })] AS theDate
RETURN theDate
```

Table 282. Result

theDate
1984-03-07
1984-03-05
1984-01-01
3 rows

8.9.6. `date()`: creating a quarter (Year-Quarter-Day) Date

`date()` returns a *Date* value with the specified *year*, *quarter* and *dayOfQuarter* component values.

Syntax: `date({year [, quarter, dayOfQuarter]})`

Returns:

A Date.

Arguments:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>quarter</code>	An integer between 1 and 4 that specifies the quarter.
<code>dayOfQuarter</code>	An integer between 1 and 92 that specifies the day of the quarter.

Considerations:

The *day of the quarter* component will default to 1 if `dayOfQuarter` is omitted.

The *quarter* component will default to 1 if `quarter` is omitted.

If `quarter` is omitted, `dayOfQuarter` must also be omitted.

Query

```
UNWIND [date({ year:1984, quarter:3, dayOfQuarter: 45 }), date({ year:1984, quarter:3 }), date({ year:1984 })] AS theDate
RETURN theDate
```

Table 283. Result

theDate
1984-08-14
1984-07-01
1984-01-01
3 rows

8.9.7. date(): creating an ordinal (Year-Day) Date

`date()` returns a *Date* value with the specified *year* and *ordinalDay* component values.

Syntax: `date({year [, ordinalDay]})`

Returns:

A Date.

Arguments:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>ordinalDay</code>	An integer between 1 and 366 that specifies the ordinal day of the year.

Considerations:

The *ordinal day of the year* component will default to 1 if `ordinalDay` is omitted.

Query

```
UNWIND [date({ year:1984, ordinalDay:202 }), date({ year:1984 })] AS theDate
RETURN theDate
```

The date corresponding to 11 February 1984 is returned.

Table 284. Result

theDate
1984-07-20
1984-01-01
2 rows

8.9.8. date(): creating a *Date* from a string

`date()` returns the *Date* value obtained by parsing a string representation of a temporal value.

Syntax: `date(temporalValue)`

Returns:

A Date.

Arguments:

Name	Description
<code>temporalValue</code>	A string representing a temporal value.

Considerations:

`temporalValue` must comply with the format defined for `dates`.

`date(null)` returns the current date.

`temporalValue` must denote a valid date; i.e. a `temporalValue` denoting `30 February 2001` is invalid.

Query

```
UNWIND [date('2015-07-21'), date('2015-07'), date('201507'), date('2015-W30-2'), date('2015202'),  
date('2015')] AS theDate  
RETURN theDate
```

Table 285. Result

theDate
2015-07-21
2015-07-01
2015-07-01
2015-07-21
2015-07-21
2015-01-01
6 rows

8.9.9. date(): creating a *Date* using other temporal values as components

`date()` returns the *Date* value obtained by selecting and composing components from another temporal value. In essence, this allows a *DateTime* or *LocalDateTime* value to be converted to a *Date*, and for "missing" components to be provided.

Syntax: `date({date [, year, month, day, week, dayOfWeek, quarter, dayOfQuarter, ordinalDay]})`

Returns:

A Date.

Arguments:

Name	Description
A single map consisting of the following:	
date	A Date value.
year	An expression consisting of at least four digits that specifies the year.
month	An integer between 1 and 12 that specifies the month.
day	An integer between 1 and 31 that specifies the day of the month.
week	An integer between 1 and 53 that specifies the week.
dayOfWeek	An integer between 1 and 7 that specifies the day of the week.
quarter	An integer between 1 and 4 that specifies the quarter.
dayOfQuarter	An integer between 1 and 92 that specifies the day of the quarter.
ordinalDay	An integer between 1 and 366 that specifies the ordinal day of the year.

Considerations:

If any of the optional parameters are provided, these will override the corresponding components of `date`.

`date(dd)` may be written instead of `date({date: dd})`.

Query

```
UNWIND [date({ year:1984, month:11, day:11 }), localdatetime({ year:1984, month:11, day:11, hour:12, minute:31, second:14 }), datetime({ year:1984, month:11, day:11, hour:12, timezone: '+01:00' })] AS dd
RETURN date({ date: dd }) AS dateOnly, date({ date: dd, day: 28 }) AS dateDay
```

Table 286. Result

dateOnly	dateDay
1984-11-11	1984-11-28
1984-11-11	1984-11-28
1984-11-11	1984-11-28
3 rows	

8.9.10. `date.truncate()`: truncating a Date

`date.truncate()` returns the *Date* value obtained by truncating a specified temporal instant value at the nearest preceding point in time at the specified component boundary (which is denoted by the truncation unit passed as a parameter to the function). In other words, the *Date* returned will have all components that are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of *overriding* the default values which would otherwise have been set for these less significant components. For example, `day` — with some value `x` — may be provided when the truncation unit is `year` in order to ensure the returned value has the `day` set to `x` instead of the default `day` (which is 1).

Syntax: `date.truncate(unit, temporalInstantValue [, mapOfComponents])`

Returns:

A Date.

Arguments:

Name	Description
unit	A string expression evaluating to one of the following: {millennium, century, decade, year, weekYear, quarter, month, week, day}.
temporalInstantValue	An expression of one of the following types: {DateTime, LocalDateTime, Date}.
mapOfComponents	An expression evaluating to a map containing components less significant than unit.

Considerations:

Any component that is provided in mapOfComponents must be less significant than unit; i.e. if unit is 'day', mapOfComponents cannot contain information pertaining to a month.

Any component that is not contained in mapOfComponents and which is less significant than unit will be set to its minimal value.

If mapOfComponents is not provided, all components of the returned value which are less significant than unit will be set to their default values.

Query

```
WITH datetime({ year:2017, month:11, day:11, hour:12, minute:31, second:14, nanosecond: 645876123, timezone: '+01:00' }) AS d
RETURN date.truncate('millennium', d) AS truncMillenium, date.truncate('century', d) AS truncCentury,
date.truncate('decade', d) AS truncDecade, date.truncate('year', d, { day:5 }) AS truncYear,
date.truncate('weekYear', d) AS truncWeekYear, date.truncate('quarter', d) AS truncQuarter,
date.truncate('month', d) AS truncMonth, date.truncate('week', d, { dayOfWeek:2 }) AS truncWeek,
date.truncate('day', d) AS truncDay
```

Table 287. Result

truncMillenium	truncCentury	truncDecade	truncYear	truncWeekYear	truncQuarter	truncMonth	truncWeek	truncDay
2000-01-01	2000-01-01	2010-01-01	2017-01-05	2017-01-02	2017-10-01	2017-11-01	2017-11-07	2017-11-11
1 row								

8.9.11. datetime(): getting the current DateTime

datetime() returns the current DateTime value. If no time zone parameter is specified, the default time zone will be used.

Syntax: `datetime([{timezone}])`

Returns:

A DateTime.

Arguments:

Name	Description
A single map consisting of the following:	
timezone	A string expression that represents the time zone

Considerations:

If no parameters are provided, `datetime()` must be invoked (`datetime({})` is invalid).

Query

```
RETURN datetime() AS currentDateTime
```

The current date and time using the local time zone is returned.

Table 288. Result

currentDateTime
2018-06-20T21:44:46.029Z
1 row

Query

```
RETURN datetime({ timezone: 'America/Los Angeles' }) AS currentDateTimeInLA
```

The current date and time of day in California is returned.

Table 289. Result

currentDateTimeInLA
2018-06-20T14:44:46.054-07:00[America/Los_Angeles]
1 row

datetime.transaction()

`datetime.transaction()` returns the current *DateTime* value using the `transaction` clock. This value will be the same for each invocation within the same transaction. However, a different value may be produced for different transactions.

Syntax: `datetime.transaction([{timezone}])`

Returns:

A DateTime.

Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the time zone

Query

```
RETURN datetime.transaction() AS currentDateTime
```

Table 290. Result

currentDateTime
2018-06-20T21:44:46.069Z
1 row

Query

```
RETURN datetime.transaction('America/Los Angeles') AS currentDateTimeInLA
```

Table 291. Result

currentDateTimeInLA
2018-06-20T14:44:46.083-07:00[America/Los_Angeles]
1 row

datetime.statement()

`datetime.statement()` returns the current *Datetime* value using the `statement` clock. This value will be the same for each invocation within the same statement. However, a different value may be produced for different statements within the same transaction.

Syntax: `datetime.statement([{timezone}])`

Returns:

A DateTime.

Arguments:

Name	Description
timezone	A string expression that represents the time zone

Query

```
RETURN datetime.statement() AS currentDateTime
```

Table 292. Result

currentDateTime
2018-06-20T21:44:46.092Z
1 row

datetime.realtime()

`datetime.realtime()` returns the current *Datetime* value using the `realtime` clock. This value will be the live clock of the system.

Syntax: `datetime.realtime([{timezone}])`

Returns:

A DateTime.

Arguments:

Name	Description
timezone	A string expression that represents the time zone

Query

```
RETURN datetime.realtime() AS currentDateTime
```

Table 293. Result

currentDateTime
2018-06-20T21:44:46.101Z
1 row

8.9.12. `datetime()`: creating a calendar (Year-Month-Day) *DateTime*

`datetime()` returns a *DateTime* value with the specified *year*, *month*, *day*, *hour*, *minute*, *second*, *millisecond*, *microsecond*, *nanosecond* and *timezone* component values.

Syntax: `datetime({year [, month, day, hour, minute, second, millisecond, microsecond, nanosecond, timezone]})`

Returns:

A *DateTime*.

Arguments:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>month</code>	An integer between <code>1</code> and <code>12</code> that specifies the month.
<code>day</code>	An integer between <code>1</code> and <code>31</code> that specifies the day of the month.
<code>hour</code>	An integer between <code>0</code> and <code>23</code> that specifies the hour of the day.
<code>minute</code>	An integer between <code>0</code> and <code>59</code> that specifies the number of minutes.
<code>second</code>	An integer between <code>0</code> and <code>59</code> that specifies the number of seconds.
<code>millisecond</code>	An integer between <code>0</code> and <code>999</code> that specifies the number of milliseconds.
<code>microsecond</code>	An integer between <code>0</code> and <code>999,999</code> that specifies the number of microseconds.
<code>nanosecond</code>	An integer between <code>0</code> and <code>999,999,999</code> that specifies the number of nanoseconds.
<code>timezone</code>	An expression that specifies the time zone.

Considerations:

The *month* component will default to `1` if `month` is omitted.

The *day of the month* component will default to `1` if `day` is omitted.

The *hour* component will default to `0` if `hour` is omitted.

The *minute* component will default to `0` if `minute` is omitted.

The `second` component will default to `0` if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to `0`.

The `timezone` component will default to the configured default time zone if `timezone` is omitted.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range `0` to `999`.

The least significant components in the set `year`, `month`, `day`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year`, `month` and `day`, but specifying `year`, `month`, `day` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

Query

```
UNWIND [datetime({ year:1984, month:10, day:11, hour:12, minute:31, second:14, millisecond: 123, microsecond: 456, nanosecond: 789 }), datetime({ year:1984, month:10, day:11, hour:12, minute:31, second:14, millisecond: 645, timezone: '+01:00' }), datetime({ year:1984, month:10, day:11, hour:12, minute:31, second:14, nanosecond: 645876123, timezone: 'Europe/Stockholm' }), datetime({ year:1984, month:10, day:11, hour:12, minute:31, second:14, timezone: '+01:00' }), datetime({ year:1984, month:10, day:11, hour:12, minute:31, second:14, timezone: 'Europe/Stockholm' }), datetime({ year:1984, month:10, day:11, hour:12, minute:31, second:14, timezone: '+01:00' }), datetime({ year:1984, month:10, day:11, hour:12, minute:31, second:14, timezone: 'Europe/Stockholm' })] AS theDate
RETURN theDate
```

Table 294. Result

theDate

1984-10-11T12:31:14.123456789Z

1984-10-11T12:31:14.645+01:00

1984-10-11T12:31:14.645876123+01:00[Europe/Stockholm]

1984-10-11T12:31:14+01:00

1984-10-11T12:31:14Z

1984-10-11T12:31+01:00[Europe/Stockholm]

1984-10-11T12:00+01:00

1984-10-11T00:00+01:00[Europe/Stockholm]

8 rows

8.9.13. `datetime()`: creating a week (Year-Week-Day) *DateTime*

`datetime()` returns a *DateTime* value with the specified `year`, `week`, `dayOfWeek`, `hour`, `minute`, `second`, `millisecond`, `microsecond`, `nanosecond` and `timezone` component values.

Syntax: `datetime({year [, week, dayOfWeek, hour, minute, second, millisecond, microsecond, nanosecond, timezone]})`

Returns:

A *DateTime*.

Arguments:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.

Name	Description
week	An integer between 1 and 53 that specifies the week.
dayOfWeek	An integer between 1 and 7 that specifies the day of the week.
hour	An integer between 0 and 23 that specifies the hour of the day.
minute	An integer between 0 and 59 that specifies the number of minutes.
second	An integer between 0 and 59 that specifies the number of seconds.
millisecond	An integer between 0 and 999 that specifies the number of milliseconds.
microsecond	An integer between 0 and 999,999 that specifies the number of microseconds.
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
timezone	An expression that specifies the time zone.

Considerations:

The `week` component will default to 1 if `week` is omitted.

The `day of the week` component will default to 1 if `dayOfWeek` is omitted.

The `hour` component will default to 0 if `hour` is omitted.

The `minute` component will default to 0 if `minute` is omitted.

The `second` component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

The `timezone` component will default to the configured default time zone if `timezone` is omitted.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `year`, `week`, `dayOfWeek`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year`, `week` and `dayOfWeek`, but specifying `year`, `week`, `dayOfWeek` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

Query

```
UNWIND [datetime({ year:1984, week:10, dayOfWeek:3, hour:12, minute:31, second:14, millisecond: 645 }),  
datetime({ year:1984, week:10, dayOfWeek:3, hour:12, minute:31, second:14, microsecond: 645876, timezone:  
'+01:00' }), datetime({ year:1984, week:10, dayOfWeek:3, hour:12, minute:31, second:14, nanosecond:  
645876123, timezone: 'Europe/Stockholm' }), datetime({ year:1984, week:10, dayOfWeek:3, hour:12,  
minute:31, second:14, timezone: 'Europe/Stockholm' }), datetime({ year:1984, week:10, dayOfWeek:3,  
hour:12, minute:31, second:14 }), datetime({ year:1984, week:10, dayOfWeek:3, hour:12, timezone: '+01:00'  
}), datetime({ year:1984, week:10, dayOfWeek:3, timezone: 'Europe/Stockholm' })] AS theDate  
RETURN theDate
```

Table 295. Result

theDate
1984-03-07T12:31:14.645Z
1984-03-07T12:31:14.645876+01:00
1984-03-07T12:31:14.645876123+01:00[Europe/Stockholm]
1984-03-07T12:31:14+01:00[Europe/Stockholm]

theDate
1984-03-07T12:31:14Z
1984-03-07T12:00+01:00
1984-03-07T00:00+01:00[Europe/Stockholm]
7 rows

8.9.14. `datetime()`: creating a quarter (Year-Quarter-Day) *DateTime*

`datetime()` returns a *DateTime* value with the specified *year*, *quarter*, *dayOfQuarter*, *hour*, *minute*, *second*, *millisecond*, *microsecond*, *nanosecond* and *timezone* component values.

Syntax: `datetime({year [, quarter, dayOfQuarter, hour, minute, second, millisecond, microsecond, nanosecond, timezone]})`

Returns:

A <i>DateTime</i> .

Arguments:

Name	Description
A single map consisting of the following:	
<i>year</i>	An expression consisting of at least four digits that specifies the year.
<i>quarter</i>	An integer between 1 and 4 that specifies the quarter.
<i>dayOfQuarter</i>	An integer between 1 and 92 that specifies the day of the quarter.
<i>hour</i>	An integer between 0 and 23 that specifies the hour of the day.
<i>minute</i>	An integer between 0 and 59 that specifies the number of minutes.
<i>second</i>	An integer between 0 and 59 that specifies the number of seconds.
<i>millisecond</i>	An integer between 0 and 999 that specifies the number of milliseconds.
<i>microsecond</i>	An integer between 0 and 999,999 that specifies the number of microseconds.
<i>nanosecond</i>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
<i>timezone</i>	An expression that specifies the time zone.

Considerations:

The <i>quarter</i> component will default to 1 if <i>quarter</i> is omitted.
--

The <i>day of the quarter</i> component will default to 1 if <i>dayOfQuarter</i> is omitted.
--

The <i>hour</i> component will default to 0 if <i>hour</i> is omitted.
--

The <i>minute</i> component will default to 0 if <i>minute</i> is omitted.
--

The <i>second</i> component will default to 0 if <i>second</i> is omitted.
--

Any missing <i>millisecond</i> , <i>microsecond</i> or <i>nanosecond</i> values will default to 0 .

The `timezone` component will default to the configured default time zone if `timezone` is omitted.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range `0` to `999`.

The least significant components in the set `year`, `quarter`, `dayOfQuarter`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year`, `quarter` and `dayOfQuarter`, but specifying `year`, `quarter`, `dayOfQuarter` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

Query

```
UNWIND [datetime({ year:1984, quarter:3, dayOfQuarter: 45, hour:12, minute:31, second:14, microsecond: 645876 }), datetime({ year:1984, quarter:3, dayOfQuarter: 45, hour:12, minute:31, second:14, timezone: '+01:00' }), datetime({ year:1984, quarter:3, dayOfQuarter: 45, hour:12, timezone: 'Europe/Stockholm' }), datetime({ year:1984, quarter:3, dayOfQuarter: 45 })] AS theDate
RETURN theDate
```

Table 296. Result

theDate
1984-08-14T12:31:14.645876Z
1984-08-14T12:31:14+01:00
1984-08-14T12:00+02:00[Europe/Stockholm]
1984-08-14T00:00Z
4 rows

8.9.15. `datetime()`: creating an ordinal (Year-Day) *DateTime*

`datetime()` returns a *DateTime* value with the specified `year`, `ordinalDay`, `hour`, `minute`, `second`, `millisecond`, `microsecond`, `nanosecond` and `timezone` component values.

Syntax: `datetime({year [, ordinalDay, hour, minute, second, millisecond, microsecond, nanosecond, timezone]})`

Returns:

A DateTime.

Arguments:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>ordinalDay</code>	An integer between <code>1</code> and <code>366</code> that specifies the ordinal day of the year.
<code>hour</code>	An integer between <code>0</code> and <code>23</code> that specifies the hour of the day.
<code>minute</code>	An integer between <code>0</code> and <code>59</code> that specifies the number of minutes.
<code>second</code>	An integer between <code>0</code> and <code>59</code> that specifies the number of seconds.
<code>millisecond</code>	An integer between <code>0</code> and <code>999</code> that specifies the number of milliseconds.

Name	Description
microsecond	An integer between 0 and 999,999 that specifies the number of microseconds.
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
timezone	An expression that specifies the time zone.

Considerations:

The *ordinal day of the year* component will default to 1 if `ordinalDay` is omitted.

The *hour* component will default to 0 if `hour` is omitted.

The *minute* component will default to 0 if `minute` is omitted.

The *second* component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

The `timezone` component will default to the configured default time zone if `timezone` is omitted.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `year`, `ordinalDay`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year` and `ordinalDay`, but specifying `year`, `ordinalDay` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

Query

```
UNWIND [datetime({ year:1984, ordinalDay:202, hour:12, minute:31, second:14, millisecond: 645 }),  
datetime({ year:1984, ordinalDay:202, hour:12, minute:31, second:14, timezone: '+01:00' }), datetime({  
year:1984, ordinalDay:202, timezone: 'Europe/Stockholm' }), datetime({ year:1984, ordinalDay:202 })] AS  
theDate  
RETURN theDate
```

Table 297. Result

theDate
1984-07-20T12:31:14.645Z
1984-07-20T12:31:14+01:00
1984-07-20T00:00+02:00[Europe/Stockholm]
1984-07-20T00:00Z
4 rows

8.9.16. `datetime()`: creating a *DateTime* from a string

`datetime()` returns the *DateTime* value obtained by parsing a string representation of a temporal value.

Syntax: `datetime(temporalValue)`

Returns:

A *DateTime*.

Arguments:

Name	Description
temporalValue	A string representing a temporal value.

Considerations:

`temporalValue` must comply with the format defined for `dates`, `times` and `time zones`.

`datetime(null)` returns the current date and time.

The `timezone` component will default to the configured default time zone if it is omitted.

`temporalValue` must denote a valid date and time; i.e. a `temporalValue` denoting `30 February 2001` is invalid.

Query

```
UNWIND [datetime('2015-07-21T21:40:32.142+0100'), datetime('2015-W30-2T214032.142Z'),
datetime('2015T214032-0100'), datetime('20150721T21:40-01:30'), datetime('2015-W30T2140-02'),
datetime('201502T21+18:00'), datetime('2015-07-21T21:40:32.142[Europe/London]'), datetime('2015-07-
21T21:40:32.142-04[America/New_York]')] AS theDate
RETURN theDate
```

Table 298. Result

theDate
2015-07-21T21:40:32.142+01:00
2015-07-21T21:40:32.142Z
2015-01-01T21:40:32-01:00
2015-07-21T21:40-01:30
2015-07-20T21:40-02:00
2015-07-21T21:00+18:00
2015-07-21T21:40:32.142+01:00[Europe/London]
2015-07-21T21:40:32.142-04:00[America/New_York]
8 rows

8.9.17. `datetime()`: creating a *DateTime* using other temporal values as components

`datetime()` returns the *DateTime* value obtained by selecting and composing components from another temporal value. In essence, this allows a *Date*, *LocalDateTime*, *Time* or *LocalTime* value to be converted to a *DateTime*, and for "missing" components to be provided.

Syntax: `datetime({datetime [, year, ..., timezone]}) | datetime({date [, year, ..., timezone]}) | datetime({time [, year, ..., timezone]}) | datetime({date, time [, year, ..., timezone]})`

Returns:

A *DateTime*.

Arguments:

Name	Description
A single map consisting of the following:	
<code>datetime</code>	A <i>DateTime</i> value.

Name	Description
date	A <i>Date</i> value.
time	A <i>Time</i> value.
year	An expression consisting of at least four digits that specifies the year.
month	An integer between 1 and 12 that specifies the month.
day	An integer between 1 and 31 that specifies the day of the month.
week	An integer between 1 and 53 that specifies the week.
dayOfWeek	An integer between 1 and 7 that specifies the day of the week.
quarter	An integer between 1 and 4 that specifies the quarter.
dayOfQuarter	An integer between 1 and 92 that specifies the day of the quarter.
ordinalDay	An integer between 1 and 366 that specifies the ordinal day of the year.
hour	An integer between 0 and 23 that specifies the hour of the day.
minute	An integer between 0 and 59 that specifies the number of minutes.
second	An integer between 0 and 59 that specifies the number of seconds.
millisecond	An integer between 0 and 999 that specifies the number of milliseconds.
microsecond	An integer between 0 and 999,999 that specifies the number of microseconds.
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
timezone	An expression that specifies the time zone.

Considerations:

If any of the optional parameters are provided, these will override the corresponding components of `datetime`, `date` and/or `time`.

`datetime(dd)` may be written instead of `datetime({datetime: dd})`.

Selecting a *Time* or *DateTime* value as the `time` component also selects its time zone. If a *LocalTime* or *LocalDateTime* is selected instead, the default time zone is used. In any case, the time zone can be overridden explicitly.

Selecting a *DateTime* as the `datetime` component and overwriting the time zone will adjust the local time to keep the same point in time.

Selecting a *DateTime* or *Time* as the `time` component and overwriting the time zone will adjust the local time to keep the same point in time.

The following query shows the various usages of `datetime({date [, year, ..., timezone]})`

Query

```
WITH date({ year:1984, month:10, day:11 }) AS dd
RETURN datetime({ date:dd, hour: 10, minute: 10, second: 10 }) AS dateHHMMSS, datetime({ date:dd, hour: 10, minute: 10, second: 10, timezone:'+05:00' }) AS dateHHMMSSTimezone, datetime({ date:dd, day: 28, hour: 10, minute: 10, second: 10 }) AS dateDDHHMMSS, datetime({ date:dd, day: 28, hour: 10, minute: 10, second: 10, timezone:'Pacific/Honolulu' }) AS dateDDHHMMSSTimezone
```

Table 299. Result

dateHHMMSS	dateHHMMSSTimezone	dateDDHMMSS	dateDDHMMMSSTimezone
1984-10-11T10:10:10Z	1984-10-11T10:10:10+05:00	1984-10-28T10:10:10Z	1984-10-28T10:10:10-10:00[Pacific/Honolulu]
1 row			

The following query shows the various usages of `datetime({time [, year, ..., timezone]})`

Query

```
WITH time({ hour:12, minute:31, second:14, microsecond: 645876, timezone: '+01:00' }) AS tt
RETURN datetime({ year:1984, month:10, day:11, time:tt }) AS YYYYMMDDTime, datetime({ year:1984, month:10, day:11, time:tt, timezone:'+05:00' }) AS YYYYMMDDTimeTimezone, datetime({ year:1984, month:10, day:11, time:tt, second: 42 }) AS YYYYMMDDTimeSS, datetime({ year:1984, month:10, day:11, time:tt, second: 42, timezone:'Pacific/Honolulu' }) AS YYYYMMDDTimeSSTimezone
```

Table 300. Result

YYYYMMDDTime	YYYYMMDDTimeTimezone	YYYYMMDDTimeSS	YYYYMMDDTimeSSTimezone
1984-10-11T12:31:14.645876+01:00	1984-10-11T16:31:14.645876+05:00	1984-10-11T12:31:42.645876+01:00	1984-10-11T01:31:42.645876-10:00[Pacific/Honolulu]
1 row			

The following query shows the various usages of `datetime({date, time [, year, ..., timezone]})`; i.e. combining a *Date* and a *Time* value to create a single *DateTime* value:

Query

```
WITH date({ year:1984, month:10, day:11 }) AS dd, localtime({ hour:12, minute:31, second:14, millisecond: 645 }) AS tt
RETURN datetime({ date:dd, time:tt }) AS dateTime, datetime({ date:dd, time:tt, timezone:'+05:00' }) AS dateTimeTimezone, datetime({ date:dd, time:tt, day: 28, second: 42 }) AS dateTimeDDSS, datetime({ date:dd, time:tt, day: 28, second: 42, timezone:'Pacific/Honolulu' }) AS dateTimeDDSSTimezone
```

Table 301. Result

dateTime	dateTimeTimezone	dateTimeDDSS	dateTimeDDSSTimezone
1984-10-11T12:31:14.645Z	1984-10-11T12:31:14.645+05:00	1984-10-28T12:31:42.645Z	1984-10-28T12:31:42.645-10:00[Pacific/Honolulu]
1 row			

The following query shows the various usages of `datetime({datetime [, year, ..., timezone]})`

Query

```
WITH datetime({ year:1984, month:10, day:11, hour:12, timezone: 'Europe/Stockholm' }) AS dd
RETURN datetime({ datetime:dd }) AS dateTime, datetime({ datetime:dd, timezone:'+05:00' }) AS dateTimeTimezone, datetime({ datetime:dd, day: 28, second: 42 }) AS dateTimeDDSS, datetime({ datetime:dd, day: 28, second: 42, timezone:'Pacific/Honolulu' }) AS dateTimeDDSSTimezone
```

Table 302. Result

dateTime	dateTimeTimezone	dateTimeDDSS	dateTimeDDSSTimezone
1984-10-11T12:00+01:00[Europe/Stockholm]	1984-10-11T16:00+05:00	1984-10-28T12:00:42+01:00[Europe/Stockholm]	1984-10-28T01:00:42-10:00[Pacific/Honolulu]
1 row			

8.9.18. `datetime()`: creating a *DateTime* from a timestamp

`datetime()` returns the *DateTime* value at the specified number of *seconds* or *milliseconds* from the UNIX epoch in the UTC time zone.

Conversions to other temporal instant types from UNIX epoch representations can be achieved by transforming a *DateTime* value to one of these types.

Syntax: `datetime({ epochSeconds | epochMillis })`

Returns:

A *DateTime*.

Arguments:

Name	Description
A single map consisting of the following:	
<code>epochSeconds</code>	A numeric value representing the number of seconds from the UNIX epoch in the UTC time zone.
<code>epochMillis</code>	A numeric value representing the number of milliseconds from the UNIX epoch in the UTC time zone.

Considerations:

`epochSeconds/epochMillis` may be used in conjunction with `nanosecond`

Query

```
RETURN datetime({ epochSeconds:timestamp()/ 1000, nanosecond: 23 }) AS theDate
```

Table 303. Result

theDate
2018-06-20T21:44:46.000000023Z
1 row

Query

```
RETURN datetime({ epochMillis: 424797300000 }) AS theDate
```

Table 304. Result

theDate
1983-06-18T15:15Z
1 row

8.9.19. `datetime.truncate()`: truncating a *DateTime*

`datetime.truncate()` returns the *DateTime* value obtained by truncating a specified temporal instant value at the nearest preceding point in time at the specified component boundary (which is denoted by the truncation unit passed as a parameter to the function). In other words, the *DateTime* returned will have all components that are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of *overriding* the default values which would otherwise have been set for these less significant components. For example, `day` — with some value `x` — may be provided when the truncation unit is `year` in order to ensure the returned value has the `day` set to `x` instead of the default `day` (which is `1`).

Syntax: `datetime.truncate(unit, temporalInstantValue [, mapOfComponents])`

Returns:

A DateTime.

Arguments:

Name	Description
<code>unit</code>	A string expression evaluating to one of the following: <code>{millennium, century, decade, year, weekYear, quarter, month, week, day, hour, minute, second, millisecond, microsecond}</code> .
<code>temporalInstantValue</code>	An expression of one of the following types: <code>{DateTime, LocalDateTime, Date}</code> .
<code>mapOfComponents</code>	An expression evaluating to a map containing components less significant than <code>unit</code> . During truncation, a time zone can be attached or overridden using the key <code>timezone</code> .

Considerations:

`temporalInstantValue` cannot be a `Date` value if `unit` is one of `{hour, minute, second, millisecond, microsecond}`.

The time zone of `temporalInstantValue` may be overridden; for example, `datetime.truncate('minute', input, {timezone: '+0200'})`.

If `temporalInstantValue` is one of `{Time, DateTime}` — a value with a time zone — and the time zone is overridden, no time conversion occurs.

If `temporalInstantValue` is one of `{LocalDateTime, Date}` — a value without a time zone — and the time zone is not overridden, the configured default time zone will be used.

Any component that is provided in `mapOfComponents` must be less significant than `unit`; i.e. if `unit` is `'day'`, `mapOfComponents` cannot contain information pertaining to a `month`.

Any component that is not contained in `mapOfComponents` and which is less significant than `unit` will be set to its `minimal value`.

If `mapOfComponents` is not provided, all components of the returned value which are less significant than `unit` will be set to their default values.

Query

```
WITH datetime({ year:2017, month:11, day:11, hour:12, minute:31, second:14, nanosecond: 645876123, timezone: '+03:00' }) AS d
RETURN datetime.truncate('millennium', d, { timezone:'Europe/Stockholm' }) AS truncMillenium,
datetime.truncate('year', d, { day:5 }) AS truncYear, datetime.truncate('month', d) AS truncMonth,
datetime.truncate('day', d, { millisecond:2 }) AS truncDay, datetime.truncate('hour', d) AS truncHour,
datetime.truncate('second', d) AS truncSecond
```

Table 305. Result

truncMillenium	truncYear	truncMonth	truncDay	truncHour	truncSecond
2000-01-01T00:00+01:00[Europe/Stockholm]	2017-01-05T00:00+03:00	2017-11-01T00:00+03:00	2017-11-11T00:00:00.002+03:00	2017-11-11T12:00+03:00	2017-11-11T12:31:14+03:00
1 row					

8.9.20. localdatetime(): getting the current *LocalDateTime*

`localdatetime()` returns the current *LocalDateTime* value. If no time zone parameter is specified, the local time zone will be used.

Syntax: `localdatetime([{timezone}])`

Returns:

A `LocalDateTime`.

Arguments:

Name	Description
A single map consisting of the following:	
<code>timezone</code>	A string expression that represents the time zone

Considerations:

If no parameters are provided, `localdatetime()` must be invoked (`localdatetime({})` is invalid).

Query

```
RETURN localdatetime() AS now
```

The current local date and time (i.e. in the local time zone) is returned.

Table 306. Result

<code>now</code>
2018-06-20T21:44:47.048
1 row

Query

```
RETURN localdatetime({ timezone: 'America/Los Angeles' }) AS now
```

The current local date and time in California is returned.

Table 307. Result

<code>now</code>
2018-06-20T14:44:47.069
1 row

`localdatetime.transaction()`

`localdatetime.transaction()` returns the current *LocalDateTime* value using the `transaction` clock. This value will be the same for each invocation within the same transaction. However, a different value may be produced for different transactions.

Syntax: `localdatetime.transaction([{timezone}])`

Returns:

A LocalDateTime.

Arguments:

Name	Description
timezone	A string expression that represents the time zone

Query

```
RETURN localdatetime.transaction() AS now
```

Table 308. Result

now
2018-06-20T21:44:47.083
1 row

localdatetime.statement()

`localdatetime.statement()` returns the current *LocalDateTime* value using the `statement` clock. This value will be the same for each invocation within the same statement. However, a different value may be produced for different statements within the same transaction.

Syntax: `localdatetime.statement([{timezone}])`

Returns:

A LocalDateTime.

Arguments:

Name	Description
timezone	A string expression that represents the time zone

Query

```
RETURN localdatetime.statement() AS now
```

Table 309. Result

now
2018-06-20T21:44:47.099
1 row

localdatetime.realtime()

`localdatetime.realtime()` returns the current *LocalDateTime* value using the `realtime` clock. This value will be the live clock of the system.

Syntax: `localdatetime.realtime([{timezone}])`

Returns:

A LocalDateTime.

Arguments:

Name	Description
timezone	A string expression that represents the time zone

Query

```
RETURN localdatetime.realtime() AS now
```

Table 310. Result

now
2018-06-20T21:44:47.140
1 row

Query

```
RETURN localdatetime.realtime('America/Los Angeles') AS nowInLA
```

Table 311. Result

nowInLA
2018-06-20T14:44:47.168
1 row

8.9.21. localdatetime(): creating a calendar (Year-Month-Day)

LocalDateTime

`localdatetime()` returns a `LocalDateTime` value with the specified `year`, `month`, `day`, `hour`, `minute`, `second`, `millisecond`, `microsecond` and `nanosecond` component values.

Syntax: `localdatetime({year [, month, day, hour, minute, second, millisecond, microsecond, nanosecond]})`

Returns:

A LocalDateTime.

Arguments:

Name	Description
A single map consisting of the following:	
year	An expression consisting of at least four digits that specifies the year.
month	An integer between <code>1</code> and <code>12</code> that specifies the month.
day	An integer between <code>1</code> and <code>31</code> that specifies the day of the month.
hour	An integer between <code>0</code> and <code>23</code> that specifies the hour of the day.

Name	Description
minute	An integer between <code>0</code> and <code>59</code> that specifies the number of minutes.
second	An integer between <code>0</code> and <code>59</code> that specifies the number of seconds.
millisecond	An integer between <code>0</code> and <code>999</code> that specifies the number of milliseconds.
microsecond	An integer between <code>0</code> and <code>999,999</code> that specifies the number of microseconds.
nanosecond	An integer between <code>0</code> and <code>999,999,999</code> that specifies the number of nanoseconds.

Considerations:

The <code>month</code> component will default to <code>1</code> if <code>month</code> is omitted.
The <code>day of the month</code> component will default to <code>1</code> if <code>day</code> is omitted.
The <code>hour</code> component will default to <code>0</code> if <code>hour</code> is omitted.
The <code>minute</code> component will default to <code>0</code> if <code>minute</code> is omitted.
The <code>second</code> component will default to <code>0</code> if <code>second</code> is omitted.
Any missing <code>millisecond</code> , <code>microsecond</code> or <code>nanosecond</code> values will default to <code>0</code> .
If <code>millisecond</code> , <code>microsecond</code> and <code>nanosecond</code> are given in combination (as part of the same set of parameters), the individual values must be in the range <code>0</code> to <code>999</code> .
The least significant components in the set <code>year</code> , <code>month</code> , <code>day</code> , <code>hour</code> , <code>minute</code> , and <code>second</code> may be omitted; i.e. it is possible to specify only <code>year</code> , <code>month</code> and <code>day</code> , but specifying <code>year</code> , <code>month</code> , <code>day</code> and <code>minute</code> is not permitted.
One or more of <code>millisecond</code> , <code>microsecond</code> and <code>nanosecond</code> can only be specified as long as <code>second</code> is also specified.

Query

```
RETURN localdatetime({ year:1984, month:10, day:11, hour:12, minute:31, second:14, millisecond: 123,
microsecond: 456, nanosecond: 789 }) AS theDate
```

Table 312. Result

theDate
1984-10-11T12:31:14.123456789
1 row

8.9.22. localdatetime(): creating a week (Year-Week-Day) *LocalDateTime*

`localdatetime()` returns a `LocalDateTime` value with the specified `year`, `week`, `dayOfWeek`, `hour`, `minute`, `second`, `millisecond`, `microsecond` and `nanosecond` component values.

Syntax: `localdatetime({year [, week, dayOfWeek, hour, minute, second, millisecond, microsecond, nanosecond]})`

Returns:

A `LocalDateTime`.

Arguments:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>week</code>	An integer between 1 and 53 that specifies the week.
<code>dayOfWeek</code>	An integer between 1 and 7 that specifies the day of the week.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

Considerations:

The `week` component will default to 1 if `week` is omitted.

The `day of the week` component will default to 1 if `dayOfWeek` is omitted.

The `hour` component will default to 0 if `hour` is omitted.

The `minute` component will default to 0 if `minute` is omitted.

The `second` component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `year`, `week`, `dayOfWeek`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year`, `week` and `dayOfWeek`, but specifying `year`, `week`, `dayOfWeek` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

Query

```
RETURN localdatetime({ year:1984, week:10, dayOfWeek:3, hour:12, minute:31, second:14, millisecond: 645 })
AS theDate
```

Table 313. Result

theDate
1984-03-07T12:31:14.645
1 row

8.9.23. localdatetime(): creating a quarter (Year-Quarter-Day) `DateTime`

`localdatetime()` returns a `LocalDateTime` value with the specified `year`, `quarter`, `dayOfQuarter`, `hour`, `minute`, `second`, `millisecond`, `microsecond` and `nanosecond` component values.

Syntax: `localdatetime({year [, quarter, dayOfQuarter, hour, minute, second, millisecond, microsecond, nanosecond]})`

Returns:

A LocalDateTime.

Arguments:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>quarter</code>	An integer between 1 and 4 that specifies the quarter.
<code>dayOfQuarter</code>	An integer between 1 and 92 that specifies the day of the quarter.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

Considerations:

The `quarter` component will default to 1 if `quarter` is omitted.

The `day of the quarter` component will default to 1 if `dayOfQuarter` is omitted.

The `hour` component will default to 0 if `hour` is omitted.

The `minute` component will default to 0 if `minute` is omitted.

The `second` component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `year`, `quarter`, `dayOfQuarter`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year`, `quarter` and `dayOfQuarter`, but specifying `year`, `quarter`, `dayOfQuarter` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

Query

```
RETURN localdatetime({ year:1984, quarter:3, dayOfQuarter: 45, hour:12, minute:31, second:14, nanosecond: 645876123 }) AS theDate
```

Table 314. Result

theDate
1984-08-14T12:31:14.645876123
1 row

8.9.24. localdatetime(): creating an ordinal (Year-Day) *LocalDateTime*

`localdatetime()` returns a *LocalDateTime* value with the specified `year`, `ordinalDay`, `hour`, `minute`, `second`, `millisecond`, `microsecond` and `nanosecond` component values.

Syntax: `localdatetime({year [, ordinalDay, hour, minute, second, millisecond, microsecond, nanosecond]})`

Returns:

A *LocalDateTime*.

Arguments:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>ordinalDay</code>	An integer between <code>1</code> and <code>366</code> that specifies the ordinal day of the year.
<code>hour</code>	An integer between <code>0</code> and <code>23</code> that specifies the hour of the day.
<code>minute</code>	An integer between <code>0</code> and <code>59</code> that specifies the number of minutes.
<code>second</code>	An integer between <code>0</code> and <code>59</code> that specifies the number of seconds.
<code>millisecond</code>	An integer between <code>0</code> and <code>999</code> that specifies the number of milliseconds.
<code>microsecond</code>	An integer between <code>0</code> and <code>999,999</code> that specifies the number of microseconds.
<code>nanosecond</code>	An integer between <code>0</code> and <code>999,999,999</code> that specifies the number of nanoseconds.

Considerations:

The `ordinal day of the year` component will default to `1` if `ordinalDay` is omitted.

The `hour` component will default to `0` if `hour` is omitted.

The `minute` component will default to `0` if `minute` is omitted.

The `second` component will default to `0` if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to `0`.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range `0` to `999`.

The least significant components in the set `year`, `ordinalDay`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year` and `ordinalDay`, but specifying `year`, `ordinalDay` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

Query

```
RETURN localdatetime({ year:1984, ordinalDay:202, hour:12, minute:31, second:14, microsecond: 645876 }) AS theDate
```

Table 315. Result

theDate
1984-07-20T12:31:14.645876
1 row

8.9.25. localdatetime(): creating a *LocalDateTime* from a string

`localdatetime()` returns the *LocalDateTime* value obtained by parsing a string representation of a temporal value.

Syntax: `localdatetime(temporalValue)`

Returns:

A <i>LocalDateTime</i> .

Arguments:

Name	Description
<code>temporalValue</code>	A string representing a temporal value.

Considerations:

<code>temporalValue</code> must comply with the format defined for dates and times .
--

<code>localdatetime(null)</code> returns the current date and time.

<code>temporalValue</code> must denote a valid date and time; i.e. a <code>temporalValue</code> denoting <code>30 February 2001</code> is invalid.
--

Query

```
UNWIND [localdatetime('2015-07-21T21:40:32.142'), localdatetime('2015-W30-2T214032.142'),  
localdatetime('2015-202T21:40:32'), localdatetime('2015202T21')] AS theDate  
RETURN theDate
```

Table 316. Result

theDate
2015-07-21T21:40:32.142
2015-07-21T21:40:32.142
2015-07-21T21:40:32
2015-07-21T21:00
4 rows

8.9.26. localdatetime(): creating a *LocalDateTime* using other temporal values as components

`localdatetime()` returns the *LocalDateTime* value obtained by selecting and composing components

from another temporal value. In essence, this allows a *Date*, *DateTime*, *Time* or *LocalTime* value to be converted to a *LocalDateTime*, and for "missing" components to be provided.

Syntax: `localdatetime({datetime [, year, ..., nanosecond]}) | localdatetime({date [, year, ..., nanosecond]}) | localdatetime({time [, year, ..., nanosecond]}) | localdatetime({date, time [, year, ..., nanosecond]})`

Returns:

A *LocalDateTime*.

Arguments:

Name	Description
A single map consisting of the following:	
<code>datetime</code>	A <i>DateTime</i> value.
<code>date</code>	A <i>Date</i> value.
<code>time</code>	A <i>Time</i> value.
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>month</code>	An integer between 1 and 12 that specifies the month.
<code>day</code>	An integer between 1 and 31 that specifies the day of the month.
<code>week</code>	An integer between 1 and 53 that specifies the week.
<code>dayOfWeek</code>	An integer between 1 and 7 that specifies the day of the week.
<code>quarter</code>	An integer between 1 and 4 that specifies the quarter.
<code>dayOfQuarter</code>	An integer between 1 and 92 that specifies the day of the quarter.
<code>ordinalDay</code>	An integer between 1 and 366 that specifies the ordinal day of the year.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

Considerations:

If any of the optional parameters are provided, these will override the corresponding components of `datetime`, `date` and/or `time`.

`localdatetime(dd)` may be written instead of `localdatetime({datetime: dd})`.

The following query shows the various usages of `localdatetime({date [, year, ..., nanosecond]})`

Query

```
WITH date({ year:1984, month:10, day:11 }) AS dd
RETURN localdatetime({ date:dd, hour: 10, minute: 10, second: 10 }) AS dateHHMMSS, localdatetime({ date:dd, day: 28, hour: 10, minute: 10, second: 10 }) AS dateDDHHMMSS
```

Table 317. Result

dateHHMMSS	dateDDHHMMSS
1984-10-11T10:10:10	1984-10-28T10:10:10
1 row	

The following query shows the various usages of `localdatetime({time [, year, ..., nanosecond]})`

Query

```
WITH time({ hour:12, minute:31, second:14, microsecond: 645876, timezone: '+01:00' }) AS tt
RETURN localdatetime({ year:1984, month:10, day:11, time:tt }) AS YYYYMMDDTime, localdatetime({ year:1984, month:10, day:11, time:tt, second: 42 }) AS YYYYMMDDTimeSS
```

Table 318. Result

YYYYMMDDTime	YYYYMMDDTimeSS
1984-10-11T12:31:14.645876	1984-10-11T12:31:42.645876
1 row	

The following query shows the various usages of `localdatetime({date, time [, year, ..., nanosecond]})`; i.e. combining a *Date* and a *Time* value to create a single *LocalDateTime* value:

Query

```
WITH date({ year:1984, month:10, day:11 }) AS dd, time({ hour:12, minute:31, second:14, microsecond: 645876, timezone: '+01:00' }) AS tt
RETURN localdatetime({ date:dd, time:tt }) AS dateTime, localdatetime({ date:dd, time:tt, day: 28, second: 42 }) AS dateTimeDDSS
```

Table 319. Result

dateTime	dateTimeDDSS
1984-10-11T12:31:14.645876	1984-10-28T12:31:42.645876
1 row	

The following query shows the various usages of `localdatetime({datetime [, year, ..., nanosecond]})`

Query

```
WITH datetime({ year:1984, month:10, day:11, hour:12, timezone: '+01:00' }) AS dd
RETURN localdatetime({ datetime:dd }) AS dateTime, localdatetime({ datetime:dd, day: 28, second: 42 }) AS dateTimeDDSS
```

Table 320. Result

dateTime	dateTimeDDSS
1984-10-11T12:00	1984-10-28T12:00:42
1 row	

8.9.27. localdatetime.truncate(): truncating a *LocalDateTime*

`localdatetime.truncate()` returns the *LocalDateTime* value obtained by truncating a specified temporal instant value at the nearest preceding point in time at the specified component boundary (which is denoted by the truncation unit passed as a parameter to the function). In other words, the *LocalDateTime* returned will have all components that are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of *overriding* the default values which would otherwise have been set for these less significant components. For example, `day` — with some value `x` — may be provided when the truncation unit is `year` in order to ensure the returned value has the `day` set to `x` instead of the default `day` (which is 1).

Syntax: `localdatetime.truncate(unit, temporalInstantValue [, mapOfComponents])`

Returns:

A *LocalDateTime*.

Arguments:

Name	Description
<code>unit</code>	A string expression evaluating to one of the following: {millennium, century, decade, year, weekYear, quarter, month, week, day, hour, minute, second, millisecond, microsecond}.
<code>temporalInstantValue</code>	An expression of one of the following types: { <i>DateTime</i> , <i>LocalDateTime</i> , <i>Date</i> }.
<code>mapOfComponents</code>	An expression evaluating to a map containing components less significant than <code>unit</code> .

Considerations:

`temporalInstantValue` cannot be a *Date* value if `unit` is one of {hour, minute, second, millisecond, microsecond}.

Any component that is provided in `mapOfComponents` must be less significant than `unit`; i.e. if `unit` is 'day', `mapOfComponents` cannot contain information pertaining to a *month*.

Any component that is not contained in `mapOfComponents` and which is less significant than `unit` will be set to its *minimal value*.

If `mapOfComponents` is not provided, all components of the returned value which are less significant than `unit` will be set to their default values.

Query

```
WITH localdatetime({ year:2017, month:11, day:11, hour:12, minute:31, second:14, nanosecond: 645876123 }) AS d
RETURN localdatetime.truncate('millennium', d) AS truncMillenium, localdatetime.truncate('year', d, { day:2 }) AS truncYear, localdatetime.truncate('month', d) AS truncMonth, localdatetime.truncate('day', d) AS truncDay, localdatetime.truncate('hour', d, { nanosecond:2 }) AS truncHour, localdatetime.truncate('second', d) AS truncSecond
```

Table 321. Result

truncMillenium	truncYear	truncMonth	truncDay	truncHour	truncSecond
2000-01-01T00:00	2017-01-02T00:00	2017-11-01T00:00	2017-11-11T00:00	2017-11-11T12:00:00.000000002	2017-11-11T12:31:14
1 row					

8.9.28. localtime(): getting the current *LocalTime*

`localtime()` returns the current *LocalTime* value. If no time zone parameter is specified, the local time zone will be used.

Syntax: `localtime([{timezone}])`

Returns:

A LocalTime.

Arguments:

Name	Description
A single map consisting of the following:	
<code>timezone</code>	A string expression that represents the time zone

Considerations:

If no parameters are provided, `localtime()` must be invoked (`localtime({})` is invalid).

Query

```
RETURN localtime() AS now
```

The current local time (i.e. in the local time zone) is returned.

Table 322. Result

now
21:44:47.603
1 row

Query

```
RETURN localtime({ timezone: 'America/Los Angeles' }) AS nowInLA
```

The current local time in California is returned.

Table 323. Result

nowInLA
14:44:47.616
1 row

`localtime.transaction()`

`localtime.transaction()` returns the current *LocalTime* value using the `transaction` clock. This value will be the same for each invocation within the same transaction. However, a different value may be produced for different transactions.

Syntax: `localtime.transaction([{timezone}])`

Returns:

A LocalTime.

Arguments:

Name	Description
timezone	A string expression that represents the time zone

Query

```
RETURN localtime.transaction() AS now
```

Table 324. Result

now
21:44:47.628
1 row

localtime.statement()

`localtime.statement()` returns the current *LocalTime* value using the `statement` clock. This value will be the same for each invocation within the same statement. However, a different value may be produced for different statements within the same transaction.

Syntax: `localtime.statement([{timezone}])`

Returns:

A LocalTime.

Arguments:

Name	Description
timezone	A string expression that represents the time zone

Query

```
RETURN localtime.statement() AS now
```

Table 325. Result

now
21:44:47.639
1 row

Query

```
RETURN localtime.statement('America/Los Angeles') AS nowInLA
```

Table 326. Result

nowInLA
14:44:47.652
1 row

localtime.realtime()

`localtime.realtime()` returns the current *LocalTime* value using the `realtime` clock. This value will be the live clock of the system.

Syntax: `localtime.realtime([{timezone}])`

Returns:

A LocalTime.

Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the time zone

Query

```
RETURN localtime.realtime() AS now
```

Table 327. Result

<code>now</code>
<code>21:44:47.677</code>
1 row

8.9.29. localtime(): creating a *LocalTime*

`localtime()` returns a *LocalTime* value with the specified *hour*, *minute*, *second*, *millisecond*, *microsecond* and *nanosecond* component values.

Syntax: `localtime({hour [, minute, second, millisecond, microsecond, nanosecond]})`

Returns:

A LocalTime.

Arguments:

Name	Description
A single map consisting of the following:	
<code>hour</code>	An integer between <code>0</code> and <code>23</code> that specifies the hour of the day.
<code>minute</code>	An integer between <code>0</code> and <code>59</code> that specifies the number of minutes.
<code>second</code>	An integer between <code>0</code> and <code>59</code> that specifies the number of seconds.
<code>millisecond</code>	An integer between <code>0</code> and <code>999</code> that specifies the number of milliseconds.
<code>microsecond</code>	An integer between <code>0</code> and <code>999,999</code> that specifies the number of microseconds.
<code>nanosecond</code>	An integer between <code>0</code> and <code>999,999,999</code> that specifies the number of nanoseconds.

Considerations:

The <code>hour</code> component will default to <code>0</code> if <code>hour</code> is omitted.
The <code>minute</code> component will default to <code>0</code> if <code>minute</code> is omitted.
The <code>second</code> component will default to <code>0</code> if <code>second</code> is omitted.
Any missing <code>millisecond</code> , <code>microsecond</code> or <code>nanosecond</code> values will default to <code>0</code> .
If <code>millisecond</code> , <code>microsecond</code> and <code>nanosecond</code> are given in combination (as part of the same set of parameters), the individual values must be in the range <code>0</code> to <code>999</code> .
The least significant components in the set <code>hour</code> , <code>minute</code> , and <code>second</code> may be omitted; i.e. it is possible to specify only <code>hour</code> and <code>minute</code> , but specifying <code>hour</code> and <code>second</code> is not permitted.
One or more of <code>millisecond</code> , <code>microsecond</code> and <code>nanosecond</code> can only be specified as long as <code>second</code> is also specified.

Query

```
UNWIND [localtime({ hour:12, minute:31, second:14, nanosecond: 789, millisecond: 123, microsecond: 456 }),  
localtime({ hour:12, minute:31, second:14 }), localtime({ hour:12 })] AS theTime  
RETURN theTime
```

Table 328. Result

theTime
12:31:14.123456789
12:31:14
12:00
3 rows

8.9.30. `localtime()`: creating a *LocalTime* from a string

`localtime()` returns the *LocalTime* value obtained by parsing a string representation of a temporal value.

Syntax: `localtime(temporalValue)`

Returns:

A *LocalTime*.

Arguments:

Name	Description
<code>temporalValue</code>	A string representing a temporal value.

Considerations:

<code>temporalValue</code> must comply with the format defined for <code>times</code> .
<code>localtime(null)</code> returns the current time.
<code>temporalValue</code> must denote a valid time; i.e. a <code>temporalValue</code> denoting <code>13:46:64</code> is invalid.

Query

```
UNWIND [localtime('21:40:32.142'), localtime('214032.142'), localtime('21:40'), localtime('21')] AS theTime
RETURN theTime
```

Table 329. Result

theTime
21:40:32.142
21:40:32.142
21:40
21:00
4 rows

8.9.31. localtime(): creating a *LocalTime* using other temporal values as components

`localtime()` returns the *LocalTime* value obtained by selecting and composing components from another temporal value. In essence, this allows a *DateTime*, *LocalDateTime* or *Time* value to be converted to a *LocalTime*, and for "missing" components to be provided.

Syntax: `localtime({time [, hour, ..., nanosecond]})`

Returns:

A LocalTime.

Arguments:

Name	Description
A single map consisting of the following:	
<code>time</code>	A <i>Time</i> value.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

Considerations:

If any of the optional parameters are provided, these will override the corresponding components of `time`.

`localtime(tt)` may be written instead of `localtime({time: tt})`.

Query

```
WITH time({ hour:12, minute:31, second:14, microsecond: 645876, timezone: '+01:00' }) AS tt
RETURN localtime({ time:tt }) AS timeOnly, localtime({ time:tt, second: 42 }) AS timeSS
```

Table 330. Result

timeOnly	timeSS
12:31:14.645876	12:31:42.645876
1 row	

8.9.32. localtime.truncate(): truncating a LocalTime

`localtime.truncate()` returns the *LocalTime* value obtained by truncating a specified temporal instant value at the nearest preceding point in time at the specified component boundary (which is denoted by the truncation unit passed as a parameter to the function). In other words, the *LocalTime* returned will have all components that are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of *overriding* the default values which would otherwise have been set for these less significant components. For example, `minute` — with some value `x` — may be provided when the truncation unit is `hour` in order to ensure the returned value has the `minute` set to `x` instead of the default `minute` (which is `1`).

Syntax: `localtime.truncate(unit, temporalInstantValue [, mapOfComponents])`

Returns:

A LocalTime.

Arguments:

Name	Description
<code>unit</code>	A string expression evaluating to one of the following: { <code>day</code> , <code>hour</code> , <code>minute</code> , <code>second</code> , <code>millisecond</code> , <code>microsecond</code> }.
<code>temporalInstantValue</code>	An expression of one of the following types: { <code>DateTime</code> , <code>LocalDateTime</code> , <code>Time</code> , <code>LocalTime</code> }.
<code>mapOfComponents</code>	An expression evaluating to a map containing components less significant than <code>unit</code> .

Considerations:

Truncating time to day — i.e. `unit` is 'day' — is supported, and yields midnight at the start of the day (`00:00`), regardless of the value of `temporalInstantValue`. However, the time zone of `temporalInstantValue` is retained.

Any component that is provided in `mapOfComponents` must be less significant than `unit`; i.e. if `unit` is 'second', `mapOfComponents` cannot contain information pertaining to a `minute`.

Any component that is not contained in `mapOfComponents` and which is less significant than `unit` will be set to its `minimal value`.

If `mapOfComponents` is not provided, all components of the returned value which are less significant than `unit` will be set to their default values.

Query

```
WITH time({ hour:12, minute:31, second:14, nanosecond: 645876123, timezone: '-01:00' }) AS t
RETURN localtime.truncate('day', t) AS truncDay, localtime.truncate('hour', t) AS truncHour,
localtime.truncate('minute', t, { millisecond:2 }) AS truncMinute, localtime.truncate('second', t) AS
truncSecond, localtime.truncate('millisecond', t) AS truncMillisecond, localtime.truncate('microsecond',
t) AS truncMicrosecond
```

Table 331. Result

truncDay	truncHour	truncMinute	truncSecond	truncMillisecond	truncMicrosecond
00:00	12:00	12:31:00.002	12:31:14	12:31:14.645	12:31:14.645876
1 row					

8.9.33. time(): getting the current Time

`time()` returns the current *Time* value. If no time zone parameter is specified, the local time zone will be used.

Syntax: `time([{timezone}])`

Returns:

A Time.

Arguments:

Name	Description
A single map consisting of the following:	
<code>timezone</code>	A string expression that represents the time zone

Considerations:

If no parameters are provided, `time()` must be invoked (`time({})` is invalid).

Query

```
RETURN time() AS currentTime
```

The current time of day using the local time zone is returned.

Table 332. Result

currentTime
21:44:47.822Z
1 row

Query

```
RETURN time({ timezone: 'America/Los Angeles' }) AS currentTimeInLA
```

The current time of day in California is returned.

Table 333. Result

```
currentTimeInLA
```

```
14:44:47.833-07:00
```

```
1 row
```

time.transaction()

`time.transaction()` returns the current *Time* value using the `transaction` clock. This value will be the same for each invocation within the same transaction. However, a different value may be produced for different transactions.

Syntax: `time.transaction([{timezone}])`

Returns:

```
A Time.
```

Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the time zone

Query

```
RETURN time.transaction() AS currentTime
```

Table 334. Result

```
currentTime
```

```
21:44:47.843Z
```

```
1 row
```

time.statement()

`time.statement()` returns the current *Time* value using the `statement` clock. This value will be the same for each invocation within the same statement. However, a different value may be produced for different statements within the same transaction.

Syntax: `time.statement([{timezone}])`

Returns:

```
A Time.
```

Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the time zone

Query

```
RETURN time.statement() AS currentTime
```

Table 335. Result

```
currentTime
```

```
21:44:47.859Z
```

```
1 row
```

Query

```
RETURN time.statement('America/Los Angeles') AS currentTimeInLA
```

Table 336. Result

```
currentTimeInLA
```

```
14:44:47.869-07:00
```

```
1 row
```

time.realtime()

`time.realtime()` returns the current *Time* value using the `realtime` clock. This value will be the live clock of the system.

Syntax: `time.realtime([{timezone}])`

Returns:

A Time.

Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the time zone

Query

```
RETURN time.realtime() AS currentTime
```

Table 337. Result

```
currentTime
```

```
21:44:47.881Z
```

```
1 row
```

8.9.34. `time()`: creating a *Time*

`time()` returns a *Time* value with the specified *hour*, *minute*, *second*, *millisecond*, *microsecond*, *nanosecond* and *timezone* component values.

Syntax: `time({hour [, minute, second, millisecond, microsecond, nanosecond, timezone]})`

Returns:

A Time.

Arguments:

Name	Description
A single map consisting of the following:	
hour	An integer between 0 and 23 that specifies the hour of the day.
minute	An integer between 0 and 59 that specifies the number of minutes.
second	An integer between 0 and 59 that specifies the number of seconds.
millisecond	An integer between 0 and 999 that specifies the number of milliseconds.
microsecond	An integer between 0 and 999,999 that specifies the number of microseconds.
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
timezone	An expression that specifies the time zone.

Considerations:

The `hour` component will default to 0 if `hour` is omitted.

The `minute` component will default to 0 if `minute` is omitted.

The `second` component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

The `timezone` component will default to the configured default time zone if `timezone` is omitted.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `hour` and `minute`, but specifying `hour` and `second` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

Query

```
UNWIND [time({ hour:12, minute:31, second:14, millisecond: 123, microsecond: 456, nanosecond: 789 }),  
time({ hour:12, minute:31, second:14, nanosecond: 645876123 }), time({ hour:12, minute:31, second:14,  
microsecond: 645876, timezone: '+01:00' }), time({ hour:12, minute:31, timezone: '+01:00' }), time({  
hour:12, timezone: '+01:00' })] AS theTime  
RETURN theTime
```

Table 338. Result

theTime
12:31:14.123456789Z
12:31:14.645876123Z
12:31:14.645876+01:00
12:31+01:00
12:00+01:00
5 rows

8.9.35. `time()`: creating a *Time* from a string

`time()` returns the *Time* value obtained by parsing a string representation of a temporal value.

Syntax: `time(temporalValue)`

Returns:

A Time.

Arguments:

Name	Description
<code>temporalValue</code>	A string representing a temporal value.

Considerations:

`temporalValue` must comply with the format defined for [times](#) and [time zones](#).

The `timezone` component will default to the configured default time zone if it is omitted.

`time(null)` returns the current time.

`temporalValue` must denote a valid time; i.e. a `temporalValue` denoting `15:67` is invalid.

Query

```
UNWIND [time('21:40:32.142+0100'), time('214032.142Z'), time('21:40:32+01:00'), time('214032-0100'),
time('21:40-01:30'), time('2140-00:00'), time('2140-02'), time('22+18:00')] AS theTime
RETURN theTime
```

Table 339. Result

theTime
21:40:32.142+01:00
21:40:32.142Z
21:40:32+01:00
21:40:32-01:00
21:40-01:30
21:40Z
21:40-02:00
22:00+18:00
8 rows

8.9.36. `time()`: creating a *Time* using other temporal values as components

`time()` returns the *Time* value obtained by selecting and composing components from another temporal value. In essence, this allows a *DateTime*, *LocalDateTime* or *LocalTime* value to be converted to a *Time*, and for "missing" components to be provided.

Syntax: `time({time [, hour, ..., timezone]})`

Returns:

A Time.

Arguments:

Name	Description
A single map consisting of the following:	
time	A <i>Time</i> value.
hour	An integer between 0 and 23 that specifies the hour of the day.
minute	An integer between 0 and 59 that specifies the number of minutes.
second	An integer between 0 and 59 that specifies the number of seconds.
millisecond	An integer between 0 and 999 that specifies the number of milliseconds.
microsecond	An integer between 0 and 999,999 that specifies the number of microseconds.
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
timezone	An expression that specifies the time zone.

Considerations:

If any of the optional parameters are provided, these will override the corresponding components of `time`.

`time(tt)` may be written instead of `time({time: tt})`.

Selecting a *Time* or *DateTime* value as the `time` component also selects its time zone. If a *LocalTime* or *LocalDateTime* is selected instead, the default time zone is used. In any case, the time zone can be overridden explicitly.

Selecting a *DateTime* or *Time* as the `time` component and overwriting the time zone will adjust the local time to keep the same point in time.

Query

```
WITH localtime({ hour:12, minute:31, second:14, microsecond: 645876 }) AS tt
RETURN time({ time:tt }) AS timeOnly, time({ time:tt, timezone:'+05:00' }) AS timeTimezone, time({ time:tt, second: 42 }) AS timeSS, time({ time:tt, second: 42, timezone:'+05:00' }) AS timeSSTimezone
```

Table 340. Result

timeOnly	timeTimezone	timeSS	timeSSTimezone
12:31:14.645876Z	12:31:14.645876+05:00	12:31:42.645876Z	12:31:42.645876+05:00
1 row			

8.9.37. `time.truncate()`: truncating a *Time*

`time.truncate()` returns the *Time* value obtained by truncating a specified temporal instant value at the nearest preceding point in time at the specified component boundary (which is denoted by the truncation unit passed as a parameter to the function). In other words, the *Time* returned will have all components that are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of overriding the default values which would otherwise have been set for these less significant components. For example, `minute` — with some value `x` — may be provided when the truncation unit is `hour` in order to ensure the returned value has the `minute` set to `x` instead of the default `minute` (which is 1).

Syntax: `time.truncate(unit, temporalInstantValue [, mapOfComponents])`

Returns:

A Time.

Arguments:

Name	Description
unit	A string expression evaluating to one of the following: { <code>day</code> , <code>hour</code> , <code>minute</code> , <code>second</code> , <code>millisecond</code> , <code>microsecond</code> }.
temporalInstantValue	An expression of one of the following types: { <code>DateTime</code> , <code>LocalDateTime</code> , <code>Time</code> , <code>LocalTime</code> }.
mapOfComponents	An expression evaluating to a map containing components less significant than <code>unit</code> . During truncation, a time zone can be attached or overridden using the key <code>timezone</code> .

Considerations:

Truncating time to day — i.e. `unit` is 'day' — is supported, and yields midnight at the start of the day (`00:00`), regardless of the value of `temporalInstantValue`. However, the time zone of `temporalInstantValue` is retained.

The time zone of `temporalInstantValue` may be overridden; for example, `time.truncate('minute', input, {timezone: '+0200'})`.

If `temporalInstantValue` is one of {`Time`, `DateTime`} — a value with a time zone — and the time zone is overridden, no time conversion occurs.

If `temporalInstantValue` is one of {`LocalTime`, `LocalDateTime`, `Date`} — a value without a time zone — and the time zone is not overridden, the configured default time zone will be used.

Any component that is provided in `mapOfComponents` must be less significant than `unit`; i.e. if `unit` is 'second', `mapOfComponents` cannot contain information pertaining to a `minute`.

Any component that is not contained in `mapOfComponents` and which is less significant than `unit` will be set to its `minimal value`.

If `mapOfComponents` is not provided, all components of the returned value which are less significant than `unit` will be set to their default values.

Query

```
WITH time({ hour:12, minute:31, second:14, nanosecond: 645876123, timezone: '-01:00' }) AS t
RETURN time.truncate('day', t) AS truncDay, time.truncate('hour', t) AS truncHour, time.truncate('minute', t) AS truncMinute, time.truncate('second', t) AS truncSecond, time.truncate('millisecond', t, { nanosecond:2 }) AS truncMillisecond, time.truncate('microsecond', t) AS truncMicrosecond
```

Table 341. Result

truncDay	truncHour	truncMinute	truncSecond	truncMillisecond	truncMicrosecond
<code>00:00-01:00</code>	<code>12:00-01:00</code>	<code>12:31-01:00</code>	<code>12:31:14-01:00</code>	<code>12:31:14.64500000</code>	<code>12:31:14.645876-01:00</code>
1 row					

8.10. Spatial functions

These functions are used to specify 2D or 3D points in a Coordinate Reference System (CRS) and to calculate the geodesic distance between two points.

Functions:

- `distance()`

- [point\(\) - WGS 84 2D](#)
- [point\(\) - WGS 84 3D](#)
- [point\(\) - Cartesian 2D](#)
- [point\(\) - Cartesian 3D](#)

The following graph is used for some of the examples below.

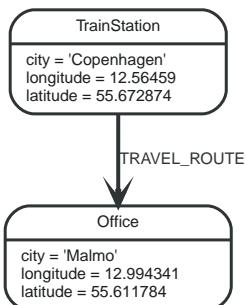


Figure 27. Graph

8.10.1. `distance()`

`distance()` returns a floating point number representing the geodesic distance between two points in the same Coordinate Reference System (CRS).

- If the points are in the *Cartesian* CRS (2D or 3D), 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 (2D), then the units of the returned distance will be meters, based on the haversine formula over a spherical earth approximation.
- If the points are in the *WGS-84* CRS (3D), then the units of the returned distance will be meters.
 - The distance is calculated in two steps.
 - First, a haversine formula over a spherical earth is used, at the average height of the two points.
 - To account for the difference in height, Pythagoras' theorem is used, combining the previously calculated spherical distance with the height difference.
 - This formula works well for points close to the earth's surface; for instance, it is well-suited for calculating the distance of an airplane flight. It is less suitable for greater heights, however, such as when calculating the distance between two satellites.

Syntax: `distance(point1, point2)`

Returns:

A Float.

Arguments:

Name	Description
<code>point1</code>	A point in either a geographic or cartesian coordinate system.
<code>point2</code>	A point in the same CRS as 'point1'.

Considerations:

```
distance(null, null), distance(null, point2) and distance(point1, null) all return null.
```

Attempting to use points with different Coordinate Reference Systems (such as WGS 84 2D and WGS 84 3D) will return `null`.

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 2D points in the *Cartesian CRS* is returned.

Table 342. Result

dist
1.5
1 row

```
WITH point({ longitude: 12.78, latitude: 56.7, height: 100 }) AS p1, point({ latitude: 56.71, longitude: 12.79, height: 100 }) AS p2
RETURN distance(p1,p2) AS dist
```

The distance between two 3D points in the *WGS 84 CRS* is returned.

Table 343. Result

dist
1269.9148706779565
1 row

```
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 Neo4j office in Malmo is returned.

Table 344. Result

travelDistance
27842.0
1 row

```
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 345. Result

d
<null>

d
1 row

8.10.2. point() - WGS 84 2D

`point(longitude|x, latitude|y, crs|srid)` returns a 2D point in the *WGS 84* CRS corresponding to the given coordinate values.

Syntax: `point({longitude | x, latitude | y [, crs][, srid]})`

Returns:

A 2D point in *WGS 84*.

Arguments:

Name	Description
A single map consisting of the following:	
<code>longitude/x</code>	A numeric expression that represents the longitude/x value in decimal degrees
<code>latitude/y</code>	A numeric expression that represents the latitude/y value in decimal degrees
<code>crs</code>	The optional string 'WGS-84'
<code>srid</code>	The optional number 4326

Considerations:

If any argument provided to `point()` is `null`, `null` will be returned.

If the coordinates are specified using `latitude` and `longitude`, the `crs` or `srid` fields are optional and inferred to be '*WGS-84*' (`srid=4326`).

If the coordinates are specified using `x` and `y`, then either the `crs` or `srid` field is required if a geographic CRS is desired.

Query

```
RETURN point({ longitude: 56.7, latitude: 12.78 }) AS point
```

A 2D point with a `longitude` of *56.7* and a `latitude` of *12.78* in the *WGS 84* CRS is returned.

Table 346. Result

point
<code>point({x: 56.7, y: 12.78, crs: 'wgs-84'})</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*', or `srid` is set to *4326*.

Table 347. Result

point

```
point({x: 2.3, y: 4.5, crs: 'wgs-84'})
```

1 row

Query

```
MATCH (p:Office)
RETURN point({ longitude: p.longitude, latitude: p.latitude }) AS officePoint
```

A 2D point representing the coordinates of the city of Malmo in the *WGS 84* CRS is returned.

Table 348. Result

officePoint

```
point({x: 12.994341, y: 55.611784, crs: 'wgs-84'})
```

1 row

Query

```
RETURN point(NULL) AS p
```

If `null` is provided as the argument, `null` is returned.

Table 349. Result

p

```
<null>
```

1 row

8.10.3. point() - WGS 84 3D

`point(longitude|x, latitude|y, height|z, crs|srid)` returns a 3D point in the *WGS 84* CRS corresponding to the given coordinate values.

Syntax: `point({longitude | x, latitude | y, height | z, [, crs][, srid]})`

Returns:

A 3D point in *WGS 84*.

Arguments:

Name	Description
A single map consisting of the following:	
<code>longitude/x</code>	A numeric expression that represents the longitude/x value in decimal degrees
<code>latitude/y</code>	A numeric expression that represents the latitude/y value in decimal degrees
<code>height/z</code>	A numeric expression that represents the height/z value in meters
<code>crs</code>	The optional string 'WGS-84-3D'
<code>srid</code>	The optional number 4979

Considerations:

If any argument provided to <code>point()</code> is <code>null</code> , <code>null</code> will be returned.
If the <code>height/z</code> key and value is not provided, a 2D point in the <i>WGS 84</i> CRS will be returned.
If the coordinates are specified using <code>latitude</code> and <code>longitude</code> , the <code>crs</code> or <code>srid</code> fields are optional and inferred to be ' <i>WGS-84-3D</i> ' (<code>srid=4979</code>).
If the coordinates are specified using <code>x</code> and <code>y</code> , then either the <code>crs</code> or <code>srid</code> field is required if a geographic CRS is desired.

Query

```
RETURN point({ longitude: 56.7, latitude: 12.78, height: 8 }) AS point
```

A 3D point with a `longitude` of **56.7**, a `latitude` of **12.78** and a height of **8** meters in the *WGS 84* CRS is returned.

Table 350. Result

<code>point</code>
<code>point({x: 56.7, y: 12.78, z: 8.0, crs: 'wgs-84-3d'})</code>
1 row

8.10.4. `point()` - Cartesian 2D

`point(x, y)` returns a 2D point in the *Cartesian* CRS corresponding to the given coordinate values.

Syntax: `point({x, y [, crs][, srid]})`

Returns:

A 2D point in *Cartesian*.

Arguments:

Name	Description
A single map consisting of the following:	
<code>x</code>	A numeric expression
<code>y</code>	A numeric expression
<code>crs</code>	The optional string 'cartesian'
<code>srid</code>	The optional number 7203

Considerations:

If any argument provided to <code>point()</code> is <code>null</code> , <code>null</code> will be returned.
The <code>crs</code> or <code>srid</code> fields are optional and default to the <i>Cartesian</i> CRS (which means <code>srid:7203</code>).

Query

```
RETURN point({ x: 2.3, y: 4.5 }) AS point
```

A 2D point with an `x` coordinate of **2.3** and a `y` coordinate of **4.5** in the *Cartesian* CRS is returned.

Table 351. Result

```
point
```

```
point({x: 2.3, y: 4.5, crs: 'cartesian'})
```

```
1 row
```

8.10.5. point() - Cartesian 3D

`point(x, y, z)` returns a 3D point in the *Cartesian* CRS corresponding to the given coordinate values.

Syntax: `point({x, y, z, [, crs][, srid]})`

Returns:

```
A 3D point in Cartesian.
```

Arguments:

Name	Description
<code>A single map consisting of the following:</code>	
<code>x</code>	A numeric expression
<code>y</code>	A numeric expression
<code>z</code>	A numeric expression
<code>crs</code>	The optional string 'cartesian-3D'
<code>srid</code>	The optional number 9157

Considerations:

```
If any argument provided to point() is null, null will be returned.
```

```
If the z key and value is not provided, a 2D point in the Cartesian CRS will be returned.
```

```
The crs or srid fields are optional and default to the 3D Cartesian CRS (which means srid:9157).
```

Query

```
RETURN point({ x: 2.3, y: 4.5, z: 2 }) AS point
```

A 3D point with an `x` coordinate of `2.3`, a `y` coordinate of `4.5` and a `z` coordinate of `2` in the *Cartesian* CRS is returned.

Table 352. Result

```
point
```

```
point({x: 2.3, y: 4.5, z: 2.0, crs: 'cartesian-3d'})
```

```
1 row
```

8.11. 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.

8.11.1. 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
```

Table 353. Result

members
"John,Paul,George,Ringo"
1 row

For developing and deploying user-defined functions in Neo4j, see [Extending Neo4j □ User-defined functions](#).

8.11.2. 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.

8.11.3. 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
 - Get a list of all indexes in the database
 - 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
 - Use index when executing a spatial distance search
 - Use built-in procedures to manage and use explicit indexes
- [Constraints](#)
 - Unique node property constraints
 - Get a list of all constraints in the database
 - Node property existence constraints
 - Relationship property existence constraints
 - Node Keys

9.1. Indexes

This section explains how to work with indexes in Neo4j and Cypher.

- [Introduction](#)
- [Create a single-property index](#)
- [Get a list of all indexes in the database](#)
- [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
- Use index when executing a spatial distance search
- Use built-in procedures to manage and use explicit indexes

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. Get a list of all indexes in the database

Calling the built-in procedure `db.indexes` will list all the indexes in the database.

Query

```
CALL db.indexes
```

Result

description	label	properties	state	type	provider
"INDEX ON :Person(firstname)" "Person" ["firstname"] "ONLINE" "node_label_property" {version -> "2.0", key -> "lucene+native"}					
"INDEX ON :Person(location)" "Person" ["location"] "ONLINE" "node_label_property" {version -> "2.0", key -> "lucene+native"}					

2 rows

9.1.4. 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.

9.1.5. 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

9.1.6. 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)
```

9.1.7. 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

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio
Variables	Other					
+ProduceResults	1	1	0	0	0	1
0.000 person						
+NodeIndexSeek	1	1	3	0	0	1
0.000 person :Person(firstname)						

Total database accesses: 3

9.1.8. 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

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
		Other					
+ProduceResults		1	1	0	0	0	1
0.0000 person							
+NodeIndexSeek		1	1	3	0	0	1
0.0000 person	:Person(firstname)						

Total database accesses: 3

9.1.9. 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](#).

9.1.10. 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

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	10	1	0	0	1	0.0000	person	
+NodeIndexSeekByRange	10	1	3	0	1	0.0000	person	:Person(firstname) > { AUTOSTRING0 }

Total database accesses: 3

9.1.11. 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

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio
Variables	Other					
+ProduceResults	24	2	0	2	0	1.0000
person						
+NodeIndexSeek	24	2	5	2	0	1.0000
person	:Person(firstname)					

Total database accesses: 5

9.1.12. 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

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	26	1	0	1	0	1.0000	person	
+NodeIndexSeekByRange	26	1	3	1	0	1.0000	person	:Person(firstname STARTS WITH \$`AUTOSTRING0`)
Total database accesses:	3							

9.1.13. 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

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	2	2	0	0	0	0.0000	p	
+NodeIndexScan	2	2	4	0	4	0.0000	p	:Person(firstname)
Total database accesses:	4							

9.1.14. Use index when executing a spatial distance search

If a property with point values is indexed, the index is used for spatial distance searches as well as for range queries.

Query

```
MATCH (p:Person)
WHERE distance(p.location, point({ x: 1, y: 2 })) < 2
RETURN p.location
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	15	9	0	31	0	1.0000	p, p.location	
+Projection	15	9	9	31	0	1.0000	p.location -- p {p.location : p.location}	
+Filter	15	9	23	31	0	1.0000	p distance(p.location, point({x: '\$` AUTOINT0`', y: '\$` AUTOINT1`'})) < '\$` AUTOINT2`'	
+NodeIndexSeekByRange	15	23	25	31	0	1.0000	p :Person(location) WHERE distance(_,point(x,y)) < Parameter(AUTOINT2, Integer)	

Total database accesses: 57

9.1.15. Use built-in procedures to manage and use explicit indexes

Explicit indexes are alternative data structures, in which a user can explicitly maintain search and seek data for nodes and relationships. These data structures are special-purpose and the procedures are primarily provided for users who have legacy deployments depending on such structures.

Signature	Description
db.index.explicit.addNode	Add a node to an explicit index based on a specified key and value
db.index.explicit.addRelationship	Add a relationship to an explicit index based on a specified key and value
db.index.explicit.auto.searchNodes	Search nodes from explicit automatic index. Replaces <code>START n=node:node_auto_index('key:foo*')</code>
db.index.explicit.auto.searchRelationships	Search relationship from explicit automatic index. Replaces <code>START r=relationship:relationship_auto_index('key:foo*')</code>

Signature	Description
db.index.explicit.auto.seekNodes	Get node from explicit automatic index. Replaces START n=node:node_auto_index(key = 'A')
db.index.explicit.auto.seekRelationships	Get relationship from explicit automatic index. Replaces START r=relationship:relationship_auto_index(key = 'A')
db.index.explicit.drop	Remove an explicit index - YIELD type, name, config
db.index.explicit.existsForNodes	Check if a node explicit index exists
db.index.explicit.existsForRelationships	Check if a relationship explicit index exists
db.index.explicit.forNodes	Get or create a node explicit index - YIELD type, name, config
db.index.explicit.forRelationships	Get or create a relationship explicit index - YIELD type, name, config
db.index.explicit.list	List all explicit indexes - YIELD type, name, config
db.index.explicit.removeNode(indexName)	Remove a node from an explicit index with an optional key
db.index.explicit.removeRelationship	Remove a relationship from an explicit index with an optional key
db.index.explicit.searchNodes	Search nodes from explicit index. Replaces START n=node:nodes('key:foo*')
db.index.explicit.searchRelationships	Search relationship from explicit index. Replaces START r=relationship:relIndex('key:foo*')
db.index.explicit.searchRelationshipsBetween	Search relationship in explicit index, starting at the node 'in' and ending at 'out'
db.index.explicit.searchRelationshipsIn	Search relationship in explicit index, starting at the node 'in'
db.index.explicit.searchRelationshipsOut	Search relationship in explicit index, ending at the node 'out'
db.index.explicit.seekNodes	Get node from explicit index. Replaces START n=node:nodes(key = 'A')
db.index.explicit.seekRelationships	Get relationship from explicit index. Replaces START r=relationship:relIndex(key = 'A')

Table 354. db.index.explicit.addNode

Signature	Description
db.index.explicit.addNode(indexName :: STRING?, node :: NODE?, key :: STRING?, value :: ANY?) :: (success :: BOOLEAN?)	Add a node to an explicit index based on a specified key and value

Table 355. db.index.explicit.addRelationship

Signature	Description
db.index.explicit.addRelationship(indexName :: STRING?, relationship :: RELATIONSHIP?, key :: STRING?, value :: ANY?) :: (success :: BOOLEAN?)	Add a relationship to an explicit index based on a specified key and value

Table 356. db.index.explicit.auto.searchNodes

Signature	Description
db.index.explicit.auto.searchNodes(query :: ANY?) :: (node :: NODE?, weight :: FLOAT?)	Search nodes from explicit automatic index. Replaces START n=node:node_auto_index('key:foo*')

Table 357. db.index.explicit.auto.searchRelationships

Signature	Description
<code>db.index.explicit.auto.searchRelationships(query :: ANY?) :: (relationship :: RELATIONSHIP?, weight :: FLOAT?)</code>	Search relationship from explicit automatic index. Replaces <code>START r=relationship:relationship_auto_index('key:foo*')</code>

Table 358. `db.index.explicit.auto.seekNodes`

Signature	Description
<code>db.index.explicit.auto.seekNodes(key :: STRING?, value :: ANY?) :: (node :: NODE?)</code>	Get node from explicit automatic index. Replaces <code>START n=node:node_auto_index(key = 'A')</code>

Table 359. `db.index.explicit.auto.seekRelationships`

Signature	Description
<code>db.index.explicit.auto.seekRelationships(key :: STRING?, value :: ANY?) :: (relationship :: RELATIONSHIP?)</code>	Get relationship from explicit automatic index. Replaces <code>START r=relationship:relationship_auto_index(key = 'A')</code>

Table 360. `db.index.explicit.drop`

Signature	Description
<code>db.index.explicit.drop(indexName :: STRING?) :: (type :: STRING?, name :: STRING?, config :: MAP?)</code>	Remove an explicit index - YIELD type, name, config

Table 361. `db.index.explicit.existsForNodes`

Signature	Description
<code>db.index.explicit.existsForNodes(indexName :: STRING?) :: (success :: BOOLEAN?)</code>	Check if a node explicit index exists

Table 362. `db.index.explicit.existsForRelationships`

Signature	Description
<code>db.index.explicit.existsForRelationships(indexName :: STRING?) :: (success :: BOOLEAN?)</code>	Check if a relationship explicit index exists

Table 363. `db.index.explicit.forNodes`

Signature	Description
<code>db.index.explicit.forNodes(indexName :: STRING?) :: (type :: STRING?, name :: STRING?, config :: MAP?)</code>	Get or create a node explicit index - YIELD type, name, config

Table 364. `db.index.explicit.forRelationships`

Signature	Description
<code>db.index.explicit.forRelationships(indexName :: STRING?) :: (type :: STRING?, name :: STRING?, config :: MAP?)</code>	Get or create a relationship explicit index - YIELD type, name, config

Table 365. `db.index.explicit.list`

Signature	Description
<code>db.index.explicit.list() :: (type :: STRING?, name :: STRING?, config :: MAP?)</code>	List all explicit indexes - YIELD type, name, config

Table 366. `db.index.explicit.removeNode`

Signature	Description
<code>db.index.explicit.removeNode(indexName :: STRING?, node :: NODE?, key :: STRING?) :: (success :: BOOLEAN?)</code>	Remove a node from an explicit index with an optional key

Table 367. `db.index.explicit.removeRelationship`

Signature	Description
<code>db.index.explicit.removeRelationship(indexName :: STRING?, relationship :: RELATIONSHIP?, key :: STRING?) :: (success :: BOOLEAN?)</code>	Remove a relationship from an explicit index with an optional key

Table 368. `db.index.explicit.searchNodes`

Signature	Description
<code>db.index.explicit.searchNodes(indexName :: STRING?, query :: ANY?) :: (node :: NODE?, weight :: FLOAT?)</code>	Search nodes from explicit index. Replaces <code>START n=node:nodes('key:foo*')</code>

Table 369. `db.index.explicit.searchRelationships`

Signature	Description
<code>db.index.explicit.searchRelationships(indexName :: STRING?, query :: ANY?) :: (relationship :: RELATIONSHIP?, weight :: FLOAT?)</code>	Search relationship from explicit index. Replaces <code>START r=relationship:relIndex('key:foo*')</code>

Table 370. `db.index.explicit.searchRelationshipsBetween`

Signature	Description
<code>db.index.explicit.searchRelationshipsBetween(indexName :: STRING?, in :: NODE?, out :: NODE?, query :: ANY?) :: (relationship :: RELATIONSHIP?, weight :: FLOAT?)</code>	Search relationship in explicit index, starting at the node 'in' and ending at 'out'

Table 371. `db.index.explicit.searchRelationshipsIn`

Signature	Description
<code>db.index.explicit.searchRelationshipsIn(indexName :: STRING?, in :: NODE?, query :: ANY?) :: (relationship :: RELATIONSHIP?, weight :: FLOAT?)</code>	Search relationship in explicit index, starting at the node 'in'

Table 372. `db.index.explicit.searchRelationshipsOut`

Signature	Description
<code>db.index.explicit.searchRelationshipsOut(indexName :: STRING?, out :: NODE?, query :: ANY?) :: (relationship :: RELATIONSHIP?, weight :: FLOAT?)</code>	Search relationship in explicit index, ending at the node 'out'

Table 373. `db.index.explicit.seekNodes`

Signature	Description
<code>db.index.explicit.seekNodes(indexName :: STRING?, key :: STRING?, value :: ANY?) :: (node :: NODE?)</code>	Get node from explicit index. Replaces <code>START n=node:nodes(key = 'A')</code>

Table 374. `db.index.explicit.seekRelationships`

Signature	Description
<code>db.index.explicit.seekRelationships(indexName :: STRING?, key :: STRING?, value :: ANY?) :: (relationship :: RELATIONSHIP?)</code>	Get relationship from explicit index. Replaces <code>START r=relationship:relIndex(key = 'A')</code>

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 unique constraint
 - Drop unique 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
- [Get a list of all constraints in the database](#)
- [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 unique 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 unique 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(20) have the label 'Book' and property 'isbn' =  
'1449356265'
```

9.2.3. Get a list of all constraints in the database

Calling the built-in procedure `db.constraints` will list all the constraints in the database.

Query

```
CALL db.constraints
```

Result

description
"CONSTRAINT ON (book:Book) ASSERT book.isbn IS UNIQUE"

1 row

9.2.4. Node property existence constraints

Create node property existence constraint

To create a constraint that ensures 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(0) 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'
```

9.2.5. 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(0) 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 **LIKED** type when there already exists a relationship without a property named **day**.

Query

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

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

Error message

```
Unable to create CONSTRAINT ON ()-[ liked:LIKED ]-() ASSERT exists(liked.day):  
Relationship(0) with type 'LIKED' must have the property 'day'
```

9.2.6. 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. |  
+-----+  
Node key constraints added: 1
```

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. |  
+-----+  
Node key constraints removed: 1
```

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(0) 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`
```

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](#).

- [Cypher query options](#)
- [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. Cypher query options

This section describes the query options available in Cypher.

Query execution can be fine-tuned through the use of query options. In order to use one or more of these options, the query must be prepended with `CYPHER`, followed by the query option(s), as exemplified thus: `CYPHER query-option [further-query-options] query`.

10.1.1. Cypher version

Occasionally, there is a requirement to use a previous version of the Cypher compiler when running a query. Here we detail the available versions:

Query option	Description	Default?
<code>2.3</code>	This will force the query to use Neo4j Cypher 2.3.	

Query option	Description	Default?
3.3	This will force the query to use Neo4j Cypher 3.3.	
3.4	This will force the query to use Neo4j Cypher 3.4. As this is the default version, it is not necessary to use this option explicitly.	X

10.1.2. Cypher planner

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.

Neo4j uses a *cost-based execution planning strategy* (known as the 'cost' planner): the statistics service in Neo4j is used to assign a cost to alternative plans and picks the cheapest one.

All versions of Neo4j prior to Neo4j 3.2 also included a rule-based planner, which used rules to produce execution plans. This planner considered available indexes, but did not use statistical information to guide the query compilation. The rule planner was removed in Neo4j 3.2 owing to inferior query execution performance when compared with the cost planner.

Option	Description	Default?
planner=rule	This will force the query to use the rule planner, and will therefore cause the query to fall back to using Cypher 3.1.	
planner=cost	Neo4j 3.4 uses the cost planner for <i>all</i> queries, and therefore it is not necessary to use this option explicitly.	X

It is also possible to change the default planner by using the `cypher.planner` configuration setting (see [Operations Manual Configuration Settings](#)).

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.

10.1.3. Cypher runtime

Using the execution plan, the query is executed — and records returned — by the query engine, or *runtime*. Depending on whether Neo4j Enterprise Edition or Neo4j Community Edition is used, there are three different runtimes available:

Interpreted

In this runtime, the operators in the execution plan are chained together in a tree, where each non-leaf operator feeds from one or two child operators. The tree thus comprises nested iterators, and the records are streamed in a pipelined manner from the top iterator, which pulls from the next iterator and so on.

Slotted

This is very similar to the interpreted runtime, except that there are additional optimizations regarding the way in which the records are streamed through the iterators. This results in improvements to both the performance and memory usage of the query. In effect, this can be thought of as a the 'faster interpreted' runtime.



The slotted runtime is only available in Neo4j Enterprise Edition.

Compiled

Algorithms are employed to intelligently group the operators in the execution plan in order to generate new combinations and orders of execution which are optimised for performance and memory usage. While this should lead to superior performance in most cases (over both the interpreted and slotted runtimes), it is still under development and does not support all possible operators or queries (the slotted runtime covers all operators and queries).



The compiled runtime is only available in Neo4j Enterprise Edition.



In Enterprise Edition, the query execution engine's fallback mechanism is to try the compiled runtime first, and if it does not support the query, to then fall back to the slotted runtime, which supports all queries.

Option	Description	Default?
<code>runtime=interpreted</code>	This will force the query execution engine to use the interpreted runtime.	This is not used in Enterprise Edition unless explicitly asked for. It is the only option for all queries in Community Edition—it is not necessary to specify this option in Community Edition.
<code>runtime=slotted</code>	This will cause the query execution engine to use the slotted runtime.	This is the default option for all queries which are not supported by <code>runtime=compiled</code> in Enterprise Edition.
<code>runtime=compiled</code>	This will cause the query execution engine to use the compiled runtime if it supports the query. If the compiled runtime does not support the query, the engine will fall back to the slotted runtime.	This is the default option for some queries in Enterprise Edition.

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 = toInteger(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 = toInteger(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 = toInteger(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 3.4
```

```
Planner COST
```

```
Runtime INTERPRETED
```

```
Runtime version 3.4
```

Operator Ratio	Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
		Other					
+ProduceResults	0.0000 p	16	1	0	0	0	0
+Filter	0.0000 p	16	1	163	0	0	0
+AllNodesScan	0.0000 p	163	163	164	0	0	0

```
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 3.4
```

```
Planner COST
```

```
Runtime INTERPRETED
```

```
Runtime version 3.4
```

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	13	1	0	0	0	0.0000	p	
+Filter	13	1	125	0	0	0.0000	p	p.name = \$`AUTOSTRING0`
+NodeByLabelScan	125	125	126	0	0	0.0000	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 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
+ProduceResults		1	1	0	0	0	0
0.0000 p							
+NodeIndexSeek		1	1	2	0	0	0
0.0000 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:

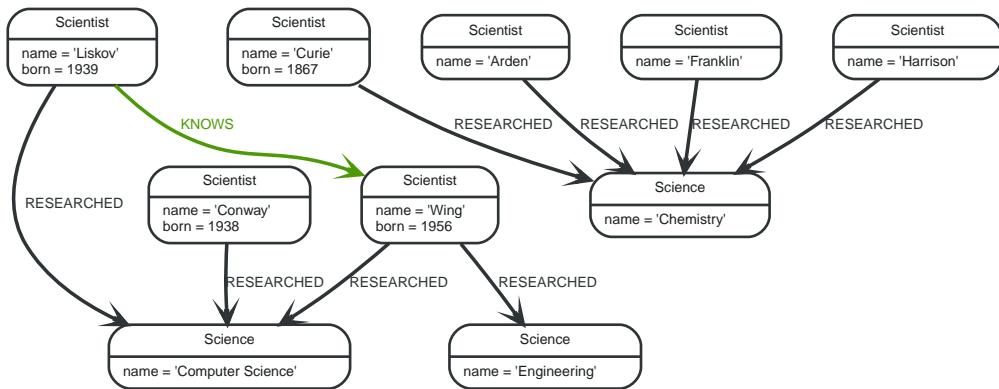


Figure 28. Graph

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

Compiler CYPHER 3.4

Planner COST

Runtime COMPILED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Time (ms)	Variables	Other
+ProduceResults	0	1	0	6	0	0	1.0000	anon[126], anon[43], anon[70], column, conway, cs, liskov, wing	
+Projection	0	1	0	6	0	0	1.0000	column -- anon[126], anon[43], anon[70], conway, cs, liskov, wing	{column : \$`AUToint3`}
+Filter	0	5	6	18	0	0	1.0000	anon[126], anon[43], anon[70], conway, cs, liskov, wing conway.name = \$`AUTOSTRING2`; not `anon[126]` = `anon[70]`	conway:Scientist;
+Expand(All)	0	3	4	6	0	0	1.0000	anon[126], conway -- anon[43], anon[70], cs, liskov, wing	(cs)<-[:RESEARCHED]-(conway)
+Filter	0	3	4	11	0	0	1.0000	anon[43], anon[70], cs, liskov, wing cs.name = \$`AUTOSTRING1`	cs:Science;
+Expand(All)	0	2	3	6	0	0	1.0000	anon[70], cs -- anon[43], liskov, wing	(wing)-[:RESEARCHED]->(cs)
+Filter	0	1	1	6	0	0	1.0000	anon[43], liskov, wing	wing:Scientist
+Expand(All)	0	1	2	6	0	0	1.0000	anon[43], wing -- liskov	(liskov)-[:KNOWS]->(wing)
+NodeIndexSeek	1	1	2	6	0	0	1.0000	liskov	:Scientist(name)

Total database accesses: 22

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

Compiler CYPHER 3.4

Planner COST

Runtime COMPILED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Time (ms)	Variables	Other
+ProduceResults	0	1	0	6	0	0	1.0000	anon[126], anon[43], anon[70], column, conway, cs, liskov, wing	
+Projection	0	1	1	6	0	0	1.0000	column -- anon[126], anon[43], anon[70], conway, cs, liskov, wing	{column : liskov.born}
+Filter	0	5	6	18	0	0	1.0000	anon[126], anon[43], anon[70], conway, cs, liskov, wing	conway.name = \$` AUTOSTRING2`; not `anon[126]` = `anon[70]`
+Expand(All)	0	3	4	6	0	0	1.0000	anon[126], conway -- anon[43], anon[70], cs, liskov, wing	(cs)<-[:RESEARCHED]-(conway)
+Filter	0	3	4	11	0	0	1.0000	anon[43], anon[70], cs, liskov, wing	cs.name = \$` AUTOSTRING1`
+Expand(All)	0	2	3	6	0	0	1.0000	anon[70], cs -- anon[43], liskov, wing	(wing)-[:RESEARCHED]->(cs)
+Filter	0	1	1	6	0	0	1.0000	anon[43], liskov, wing	wing:Scientist
+Expand(All)	0	1	2	6	0	0	1.0000	anon[43], wing -- liskov	(liskov)-[:KNOWS]->(wing)
+NodeIndexSeek	1	1	2	6	0	0	1.0000	liskov	:Scientist(name)

Total database accesses: 23

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

Compiler CYPHER 3.4

Planner COST

Runtime COMPILED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Other	Variables
Hit Ratio							
Time (ms)							
Variables							
+ProduceResults	0	1	0	7	0	0	anon[126], anon[43], anon[70], column, conway, cs, liskov, wing
1.0000 0.177							
+Projection	0	1	1	7	0	0	{column : liskov.born}
1.0000 0.210 column -- anon[126], anon[43], anon[70], conway, cs, liskov, wing {column : liskov.born}							
+Filter	0	1	0	7	0	0	not `anon[126]` = `anon[70]`
1.0000 0.218 anon[126], anon[43], anon[70], conway, cs, liskov, wing							
+NodeHashJoin	0	1	0	12	0	0	anon[126], conway, cs
1.0000 0.854 anon[43], anon[70], liskov, wing -- anon[126], conway, cs cs							
+Expand(Into)	0	1	2	7	0	0	(cs)<-[:RESEARCHED]-(conway)
1.0000 1.423 anon[126] -- conway, cs							
+CartesianProduct	1	1	0	7	0	0	cs -- conway
1.0000 1.431 cs -- conway							
+NodeIndexSeek	1	1	2	7	0	0	:Scientist(name)
1.0000 1.589 conway							

```

| | |
| | +-----+-----+-----+
| | | +NodeIndexSeek | 1 | 1 | 2 | 7 | 0 |
1.0000 | 1.734 | cs | :Science(name)
|
| | +-----+-----+-----+
| | +-----+-----+-----+
| | +-----+-----+-----+
| +Filter | 0 | 3 | 4 | 10 | 0 |
1.0000 | 0.782 | anon[43], anon[70], cs, liskov, wing | cs:Science;
cs.name = $`AUTOSTRING1` |
|
| | +-----+-----+-----+
| | +-----+-----+-----+
| +Expand(All) | 0 | 2 | 3 | 5 | 0 |
1.0000 | 0.854 | anon[70], cs -- anon[43], liskov, wing | (wing)-[:RESEARCHED]->(cs)
|
| | +-----+-----+-----+
| +Filter | 0 | 1 | 1 | 5 | 0 |
1.0000 | 0.946 | anon[43], liskov, wing | wing:Scientist
|
| | +-----+-----+-----+
| +-----+-----+-----+
| +Expand(All) | 0 | 1 | 2 | 5 | 0 |
1.0000 | 1.170 | anon[43], wing -- liskov | (liskov)-[:KNOWS]->(wing)
|
| | +-----+-----+-----+
| +NodeIndexSeek | 1 | 1 | 2 | 5 | 0 |
1.0000 | 1.588 | liskov | :Scientist(name)
|

```

Total database accesses: 19

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

Compiler CYPHER 3.4

Planner COST

Runtime SLOTTED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	0	2	0	2	0	1.0000	column, s	
+Projection	0	2	2	2	0	1.0000	column -- s {column : s.born}	
+Filter	0	2	7	2	0	1.0000	s s.born < \$`AUTOINT0`	
+NodeByLabelScan	7	7	8	3	0	1.0000	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 `(:Science)-[: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

Compiler CYPHER 3.4

Planner COST

Runtime COMPILED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Time (ms)	Variables	Other
+ProduceResults	0	1	0	7	0	0	1.0000	anon[126], anon[43], anon[70], column, conway, cs, liskov, wing	
+Projection	0	1	1	7	0	0	1.0000	column -- anon[126], anon[43], anon[70], conway, cs, liskov, wing {column : wing.born}	{column : wing.born}
+NodeHashJoin	0	1	0	16	0	0	1.0000	anon[43], liskov -- anon[126], anon[70], conway, cs, wing	wing
+Filter	1	2	0	19	0	0	1.0000	anon[126], anon[70], conway, cs, wing not `anon[126]` = `anon[70]`	not `anon[126]` = `anon[70]`
+Expand(All)	1	3	4	7	0	0	1.0000	0.534 anon[70], wing -- anon[126], conway, cs	(cs)<-[:RESEARCHED]-(wing)
+Expand(Into)	0	1	2	7	0	0	1.0000	0.646 anon[126] -- conway, cs	(cs)<-[:RESEARCHED]-(conway)
+CartesianProduct	1	1	0	7	0	0	1.0000	0.654 cs -- conway	
+NodeIndexSeek	1	1	2	7	0	0	1.0000	0.808 conway	:Scientist(name)
+NodeIndexSeek	1	1	2	7	0	0	1.0000	0.942 cs	:Science(name)
+Filter	0	1	1	3	0	0	1.0000	anon[43], liskov, wing	wing:Scientist
+Expand(All)	0	1	2	3	0	0	1.0000	0.854 anon[43], wing -- liskov	(liskov)-

```

[:KNOWS]->(wing)      |
| | +-----+-----+-----+
+-----+-----+-----+
| +NodeIndexSeek |           1 |   1 |     2 |           3 |           0 |
1.0000 |   1.271 | liskov
|
+-----+-----+-----+
+-----+-----+

```

Total database accesses: 16

Hinting a join on multiple nodes

The query planner can be made to produce a join between several specific points. This requires the query to expand from the same node from several directions.

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
RETURN wing.born AS column

```

Returns the birth date of 'Jeanette Wing'.

Query plan

```

Compiler CYPHER 3.4
Planner COST
Runtime COMPILED
Runtime version 3.4

+-----+-----+-----+-----+-----+
+-----+-----+
+-----+
| Operator          | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page Cache
Hit Ratio | Time (ms) | Variables
| Other
| |
+-----+-----+-----+-----+-----+
+-----+
| +ProduceResults |           0 |   1 |     0 |           7 |           0 |
1.0000 |   0.175 | anon[142], anon[43], anon[86], column, cs, liskov, wing |
|
| |
+-----+-----+-----+-----+
+-----+
| +Projection |           0 |   1 |     1 |           7 |           0 |
1.0000 |   0.210 | column -- anon[142], anon[43], anon[86], cs, liskov, wing | {column : wing.born}
|
| |
+-----+-----+-----+-----+
+-----+
| +Filter |           0 |   1 |     0 |           7 |           0 |
1.0000 |   0.219 | anon[142], anon[43], anon[86], cs, liskov, wing | not `anon[142]' =
`anon[86]`
|
| |
+-----+-----+-----+-----+
+-----+
| +NodeHashJoin |           0 |   1 |     0 |           12 |           0 |
1.0000 |   1.051 | anon[43], anon[86], wing -- anon[142], cs, liskov | liskov, cs
|
| \\
+-----+-----+-----+
+-----+
| | +Expand(Into) |           0 |   1 |     2 |           7 |           0 |
1.0000 |   0.441 | anon[142] -- cs, liskov
| | (cs)<-[RESEARCHED]-

```

```

(liskov) | |
| | +-----+-----+-----+
| | +CartesianProduct | 1 | 1 | 0 | 7 | 0 |
1.0000 | 0.455 | cs -- liskov | |
| | | \
| | +-----+-----+-----+
| | +NodeIndexSeek | 1 | 1 | 2 | 7 | 0 |
1.0000 | 0.620 | liskov | :Scientist(name)
| | | \
| | +-----+-----+-----+
| | +NodeIndexSeek | 1 | 1 | 2 | 7 | 0 |
1.0000 | 0.762 | cs | :Science(name)
| | | \
| | +-----+-----+-----+
| +Filter | 0 | 3 | 4 | 10 | 0 |
1.0000 | 0.798 | anon[43], anon[86], cs, liskov, wing | cs:Science; cs.name = $`AUTOSTRING2` |
| | | \
| | +-----+-----+-----+
| +Expand(All) | 0 | 2 | 3 | 5 | 0 |
1.0000 | 0.877 | anon[86], cs -- anon[43], liskov, wing | (wing)-[:RESEARCHED]->(cs)
| | | \
| | +-----+-----+-----+
| +Filter | 0 | 2 | 2 | 5 | 0 |
1.0000 | 1.113 | anon[43], liskov, wing | wing.name = $`AUTOSTRING1` ; wing:Scientist |
| | | \
| | +-----+-----+-----+
| +Expand(All) | 0 | 1 | 2 | 5 | 0 |
1.0000 | 1.427 | anon[43], wing -- liskov | (liskov)-[:KNOWS]->(wing)
| | | \
| | +-----+-----+-----+
| +NodeIndexSeek | 1 | 1 | 2 | 5 | 0 |
1.0000 | 1.934 | liskov | :Scientist(name)
| | | \
| | +-----+-----+-----+

```

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 the operators used as part of a Cypher query's execution plan.

The task of executing a query is decomposed into *operators*, each of which implements a single unit of work. The operators are combined into a structure called an *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. The actions triggering a database hit are listed [here](#).

To produce an efficient plan for a query, Neo4j requires information about the database, such as which indexes and constraints are present (see [Schema](#)). Statistical information maintained by the database is also used to generate an efficient execution plan; i.e. the query engine uses this information to determine which access patterns will produce the best execution plan.

The statistical information maintained by Neo4j is:

1. The number of nodes having a certain label.
2. Selectivity per index.
3. The number of relationships by type.
4. The number of relationships by type, ending with or starting from a node with a specific label.

More information about how the statistics are kept up to date, as well as configuration options about managing query replanning can be found in the [Operations Manual](#) \square [Statistics and execution plans](#)

See [Profiling a query](#) for how to view the execution plan for a query.

For a deeper understanding of how each operator works, refer the relevant section. 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](#)
- [Shortest path planning](#)

11.1. Execution plan operators at a glance

This table comprises all the execution plan operators ordered lexicographically.

- *Leaf* operators, in most cases, locate the starting nodes and relationships required in order to execute the query.
- *Updating* operators are used in queries that update the graph.

Name	Description	Leaf?	Updating?
AllNodesScan	Reads all nodes from the node store.	Y	
AntiConditionalApply	Performs a nested loop. If a variable is <code>null</code> , the right-hand side will be executed.		
AntiSemiApply	Performs a nested loop. Tests for the absence of a pattern predicate.		
Apply	Performs a nested loop. Yields rows from both the left-hand and right-hand side operators.		
Argument	Indicates the variable to be used as an argument to the right-hand side of an <code>Apply</code> operator.	Y	
AssertSameNode	Used to ensure that no unique constraints are violated.		
CartesianProduct	Produces a cartesian product of the inputs from the left-hand and right-hand operators.		
ConditionalApply	Performs a nested loop. If a variable is not <code>null</code> , the right-hand side will be executed.		
CreateIndex	Creates an index on a property for all nodes having a certain label.	Y	Y
CreateNode	Creates a node.	Y	Y
CreateNodeKeyConstraint	Creates a Node Key on a set of properties for all nodes having a certain label.	Y	Y
CreateNodePropertyExistenceConstraint	Creates an existence constraint on a property for all nodes having a certain label.	Y	Y
CreateRelationship	Creates a relationship.		Y
CreateRelationshipPropertyExistenceConstraint	Creates an existence constraint on a property for all relationships of a certain type.	Y	Y
CreateUniqueConstraint	Creates a unique constraint on a property for all nodes having a certain label.	Y	Y
Delete	Deletes a node or relationship.		Y
DetachDelete	Deletes a node and its relationships.		Y
DirectedRelationshipByIdSeek	Reads one or more relationships by id from the relationship store.	Y	
Distinct	Drops duplicate rows from the incoming stream of rows.		

Name	Description	Leaf?	Updating?
DropIndex	Drops an index from a property for all nodes having a certain label.	Y	Y
DropNodeKeyConstraint	Drops a Node Key from a set of properties for all nodes having a certain label.	Y	Y
DropNodePropertyExistenceConstraint	Drops an existence constraint from a property for all nodes having a certain label.	Y	Y
DropRelationshipPropertyExistenceConstraint	Drops an existence constraint from a property for all relationships of a certain type.	Y	Y
DropResult	Produces zero rows when an expression is guaranteed to produce an empty result.		
DropUniqueConstraint	Drops a unique constraint from a property for all nodes having a certain label.	Y	Y
Eager	For isolation purposes, Eager ensures that operations affecting subsequent operations are executed fully for the whole dataset before continuing execution.		
EagerAggregation	Evaluates a grouping expression.		
EmptyResult	Eagerly loads all incoming data and discards it.		
EmptyRow	Returns a single row with no columns.	Y	
Expand(All)	Traverses incoming or outgoing relationships from a given node.		
Expand(Into)	Finds all relationships between two nodes.		
Filter	Filters each row coming from the child operator, only passing through rows that evaluate the predicates to true.		
Foreach	Performs a nested loop. Yields rows from the left-hand operator and discards rows from the right-hand operator.		
LetAntiSemiApply	Performs a nested loop. Tests for the absence of a pattern predicate in queries containing multiple pattern predicates.		
LetSelectOrSemiApply	Performs a nested loop. Tests for the presence of a pattern predicate that is combined with other predicates.		

Name	Description	Leaf?	Updating?
LetSelectOrAntiSemiApply	Performs a nested loop. Tests for the absence of a pattern predicate that is combined with other predicates.		
LetSemiApply	Performs a nested loop. Tests for the presence of a pattern predicate in queries containing multiple pattern predicates.		
Limit	Returns the first 'n' rows from the incoming input.		
LoadCSV	Loads data from a CSV source into the query.	Y	
LockNodes	Locks the start and end node when creating a relationship.		
MergeCreateNode	Creates the node when failing to find the node.	Y	Y
MergeCreateRelationship	Creates the relationship when failing to find the relationship.		Y
NodeByIdSeek	Reads one or more nodes by id from the node store.	Y	
NodeByLabelScan	Fetches all nodes with a specific label from the node label index.	Y	
NodeCountFromCountStore	Uses the count store to answer questions about node counts.	Y	
NodeHashJoin	Executes a hash join on node ids.		
NodeIndexContainsScan	Examines all values stored in an index, searching for entries containing a specific string.	Y	
NodeIndexEndsWithScan	Examines all values stored in an index, searching for entries ending in a specific string.	Y	
NodeIndexScan	Examines all values stored in an index, returning all nodes with a particular label having a specified property.	Y	
NodeIndexSeek	Finds nodes using an index seek.	Y	
NodeIndexSeekByRange	Finds nodes using an index seek where the value of the property matches the given prefix string.	Y	
NodeLeftOuterHashJoin	Executes a left outer hash join.		
NodeRightOuterHashJoin	Executes a right outer hash join.		
NodeUniqueIndexSeek	Finds nodes using an index seek within a unique index.	Y	

Name	Description	Leaf?	Updating?
NodeUniqueIndexSeekByRange	Finds nodes using an index seek within a unique index where the value of the property matches the given prefix string.	Y	
Optional	Yields a single row with all columns set to <code>null</code> if no data is returned by its source.		
OptionalExpand(All)	Traverses relationships from a given node, producing a single row with the relationship and end node set to <code>null</code> if the predicates are not fulfilled.		
OptionalExpand(Into)	Traverses all relationships between two nodes, producing a single row with the relationship and end node set to <code>null</code> if no matching relationships are found (the start node will be the node with the smallest degree).		
ProcedureCall	Calls a procedure.		
ProduceResults	Prepares the result so that it is consumable by the user.		
ProjectEndpoints	Projects the start and end node of a relationship.		
Projection	Evaluates a set of expressions, producing a row with the results thereof.	Y	
RelationshipCountFromCountStore	Uses the count store to answer questions about relationship counts.	Y	
RemoveLabels	Deletes labels from a node.		Y
RollUpApply	Performs a nested loop. Executes a pattern expression or pattern comprehension.		
SelectOrAntiSemiApply	Performs a nested loop. Tests for the absence of a pattern predicate if an expression predicate evaluates to <code>false</code> .		
SelectOrSemiApply	Performs a nested loop. Tests for the presence of a pattern predicate if an expression predicate evaluates to <code>false</code> .		
SemiApply	Performs a nested loop. Tests for the presence of a pattern predicate.		
SetLabels	Sets labels on a node.		Y
SetNodePropertyFromMap	Sets properties from a map on a node.		Y
SetProperty	Sets a property on a node or relationship.		Y
SetRelationshipPropertyFromMap	Sets properties from a map on a relationship.		Y

Name	Description	Leaf?	Updating?
Skip	Skips 'n' rows from the incoming rows.		
Sort	Sorts rows by a provided key.		
Top	Returns the first 'n' rows sorted by a provided key.		
TriadicSelection	Solves triangular queries, such as the very common 'find my friend-of-friends that are not already my friend'.		
UndirectedRelationshipById Seek	Reads one or more relationships by id from the relationship store.	Y	
Union	Concatenates the results from the right-hand operator with the results from the left-hand operator.		
Unwind	Returns one row per item in a list.		
ValueHashJoin	Executes a hash join on arbitrary values.		
VarLengthExpand(All)	Traverses variable-length relationships from a given node.		
VarLengthExpand(Into)	Finds all variable-length relationships between two nodes.		
VarLengthExpand(Pruning)	Traverses variable-length relationships from a given node and only returns unique end nodes.		

11.2. DbHits (Database hits)

Each operator will send a request to the storage engine to do work such as retrieving or updating data. A *database hit* is an abstract unit of this storage engine work.

We list below all the actions that trigger a database hit:

- Create actions
 - Create a node
 - Create a relationship
 - Create a new node label
 - Create a new relationship type
 - Create a new ID for property keys with the same name
- Delete actions
 - Delete a node
 - Delete a relationship
- Update actions
 - Set one or more labels on a node

- Remove one or more labels from a node
- Node-specific actions
 - Get a node by its ID
 - Get the degree of a node
 - Determine whether a node is dense
 - Determine whether a label is set on a node
 - Get the labels of a node
 - Get a property of a node
 - Get an existing node label
 - Get the name of a label by its ID, or its ID by its name
- Relationship-specific actions
 - Get a relationship by its ID
 - Get a property of a relationship
 - Get an existing relationship type
 - Get a relationship type name by its ID, or its ID by its name
- General actions
 - Get the name of a property key by its ID, or its ID by the key name
 - Find a node or relationship through an index seek or index scan
 - Find a path in a variable-length expand
 - Find a shortest path
 - Ask the count store for a value
- Schema actions
 - Add an index
 - Drop an index
 - Get the reference of an index
 - Create a constraint
 - Drop a constraint
- Call a procedure
- Call a user-defined function

11.3. Operators

All operators are listed here, grouped by the similarity of their characteristics.

11.3.1. All Nodes Scan

The `AllNodesScan` operator reads all nodes from the node store. The variable that will contain the nodes is seen in the arguments. Any query using this operator is likely to encounter performance problems on a non-trivial database.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Estimated Rows Variables	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
0.0000	+ProduceResults n	35	35	0	0	0
1.0000	+AllNodesScan n	35	35	36	1	0

Total database accesses: 36

11.3.2. Directed Relationship By Id Seek

The [DirectedRelationshipByIdSeek](#) operator reads one or more relationships by id from the relationship store, and produces both the relationship and the nodes on either side.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	1	1	0	0	0	0.0000	anon[17], n1, r	
+DirectedRelationshipByIdSeek	1	1	1	1	1	1.0000	anon[17], n1, r EntityByIdRhs(ManySeekableArgs(ListLiteral(List(Parameter(AUTOINT0, Integer))))))	
Total database accesses:	1							

11.3.3. Node By Id Seek

The **NodeByIdSeek** operator reads one or more nodes by id from the node store.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	1	1	0	0	0	0.0000	n	
+NodeByIdSeek	1	1	1	1	1	1.0000	n	
Total database accesses:	1							

11.3.4. Node By Label Scan

The [NodeByLabelScan](#) operator fetches all nodes with a specific label from the node label index.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	14	14	0	0	0	0.0000	person	
+NodeByLabelScan	14	14	15	1	0	1.0000	person	:Person

Total database accesses: 15

11.3.5. Node Index Seek

The [NodeIndexSeek](#) operator 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 instead called [NodeUniqueIndexSeek](#).

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Estimated Rows Variables	Rows Other	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
+ProduceResults 0.0000 location 1 1 0 0 1						
+NodeIndexSeek 0.0000 location 1 1 3 0 1						
Total database accesses: 3						

11.3.6. Node Unique Index Seek

The `NodeUniqueIndexSeek` operator finds nodes using an index seek within a unique index. The node variable and the index used is shown in the arguments of the operator. If the index is not unique, the operator is instead called `NodeIndexSeek`. If the index seek is used to solve a `MERGE` clause, it will also be marked with `(Locking)`. This makes it clear that any nodes returned from the index will be locked in order to prevent concurrent conflicting updates.

Query

```
MATCH (t:Team { name: 'Malmo' })
RETURN t
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Cache Hit Ratio	Estimated Rows Variables	Rows Other	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
+ProduceResults 0.0000 t 1 0 0 0 1						
+NodeUniqueIndexSeek 0.0000 t 1 0 2 0 1						

Total database accesses: 2

11.3.7. Node Index Seek By Range

The `NodeIndexSeekByRange` operator finds nodes using an index seek where the value of the property matches a given prefix string. `NodeIndexSeekByRange` can be used for `STARTS WITH` and comparison operators such as `<`, `>`, `<=` and `>=`. If the index is a unique index, the operator is instead called `NodeUniqueIndexSeekByRange`.

Query

```
MATCH (l:Location)
WHERE l.name STARTS WITH 'Lon'
RETURN l
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page
Cache Hit Ratio	Variables	Other				
+ProduceResults	1	25	1	0	1	0
1.0000 1		+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+NodeIndexSeekByRange	1	25	1	3	1	0
1.0000 1		:Location(name STARTS WITH \$`AUTOSTRING0`)				

Total database accesses: 3

11.3.8. Node Unique Index Seek By Range

The `NodeUniqueIndexSeekByRange` operator finds nodes using an index seek within a unique index, where the value of the property matches a given prefix string. `NodeUniqueIndexSeekByRange` is used by `STARTS WITH` and comparison operators such as `<`, `>`, `<=` and `>=`. If the index is not unique, the operator is instead called `NodeIndexSeekByRange`.

Query

```
MATCH (t:Team)
WHERE t.name STARTS WITH 'Ma'
RETURN t
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	25	0	0	1	0	1.0000	t	
+NodeUniqueIndexSeekByRange	25	0	2	1	0	1.0000	t	:Team(name STARTS WITH \$`AUTOSTRING0`)
Total database accesses: 2								

11.3.9. Node Index Contains Scan

The [NodeIndexContainsScan](#) operator examines all values stored in an index, searching for entries containing a specific string; for example, in queries including [CONTAINS](#). Although this is slower than an index seek (since all entries need to be examined), it is still faster than the indirection resulting from a label scan using [NodeByLabelScan](#), and a property store filter.

Query

```
MATCH (l:Location)
WHERE l.name CONTAINS 'al'
RETURN l
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	10	2	0	0	0	0.0000	l	
+NodeIndexContainsScan	10	2	4	0	1	0.0000	l	:Location(name); \$`AUTOSTRING0`
Total database accesses: 4								

11.3.10. Node Index Ends With Scan

The `NodeIndexEndsWithScan` operator examines all values stored in an index, searching for entries ending in a specific string; for example, in queries containing `ENDS WITH`. Although this is slower than an index seek (since all entries need to be examined), it is still faster than the indirection resulting from a label scan using `NodeByLabelScan`, and a property store filter.

Query

```
MATCH (l:Location)
WHERE l.name ENDS WITH 'al'
RETURN l
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	10	0	0	0	0	0.0000	1	
+NodeIndexEndsWithScan	10	0	2	0	1	0.0000	:Location(name); \$`AUTOSTRING0`	

Total database accesses: 2

11.3.11. Node Index Scan

The `NodeIndexScan` operator examines all values stored in an index, returning all nodes with a particular label having a specified property.

Query

```
MATCH (l:Location)
WHERE exists(l.name)
RETURN l
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Estimated Rows Variables	Rows Other	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
0.0000 1	10	10	0	0	0	0
0.0000 1	10 :Location(name)	10	12	0	4	

Total database accesses: 12

11.3.12. Undirected Relationship By Id Seek

The [UndirectedRelationshipByIdSeek](#) operator reads one or more relationships by id from the relationship store. As the direction is unspecified, two rows are produced for each relationship as a result of alternating the combination of the start and end node.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
Page Cache Hit Ratio	Variables					
+ProduceResults	0.0000	anon[16], n1, r 1	2	0	0	0
+UndirectedRelationshipByIdSeek	1.0000	anon[16], n1, r 1	2	1	1	0

Total database accesses: 1

11.3.13. Apply

All the different [Apply](#) operators (listed below) share the same basic functionality: they perform a

nested loop by taking a single row from the left-hand side, and using the [Argument](#) operator on the right-hand side, execute the operator tree on the right-hand side. The versions of the [Apply](#) operators differ in how the results are managed. The [Apply](#) operator (i.e. the standard version) takes the row produced by the right-hand side — which at this point contains data from both the left-hand and right-hand sides — and yields it..

Query

```
MATCH (p:Person { name:'me' })
MATCH (q:Person { name: p.secondName })
RETURN p, q
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	1	0	0	0	1	0.0000	p, q	
+Apply	1	0	0	0	1	0.0000	p, q	
+NodeIndexSeek	1	0	3	0	0	0.0000	q -- p	:Person(name)
+NodeIndexSeek	1	1	3	0	1	0.0000	p	:Person(name)

Total database accesses: 6

11.3.14. Semi Apply

The [SemiApply](#) operator tests for the presence of a pattern predicate, and is a variation of the [Apply](#) operator. If the right-hand side operator yields at least one row, the row from the left-hand side operator 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]->(:Person)
RETURN p.name
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
							Other
+ProduceResults	11	2	0	3	0	0.0000	p, p.name
+Projection	11	2	2	3	0	0.0000	p.name -- p
+SemiApply	11	2	0	3	0	0.0000	p
+Filter	2	0	2	0	0	0.0000	NODE45, REL27, p
+Expand(All)	2	2	16	0	0	0.0000	NODE45, REL27 -- p (p)-[:FRIENDS_WITH]->(NODE45)
+Argument	14	14	0	0	0	0.0000	p
+NodeByLabelScan	14	14	15	4	0	1.0000	p :Person

Total database accesses: 35

11.3.15. Anti Semi Apply

The [AntiSemiApply](#) operator tests for the absence of a pattern, and is a variation of the [Apply](#) operator. If the right-hand side operator yields no rows, the row from the left-hand side operator 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
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
							Other
+ProduceResults	4	13	0	4	1	0.8000	me, other, other.name
+Projection	4	13	13	4	1	0.8000	other.name -- me, other {other.name : other.name}
+AntiSemiApply	4	13	0	4	1	0.8000	me, other
+Expand(Into)	0	0	50	0	0	0.0000	REL62 -- me, other (me)-[:REL62:FRIENDS_WITH]->(other)
+Argument	14	14	0	0	0	0.0000	me, other
+CartesianProduct	14	14	0	4	1	0.8000	me -- other
+NodeByLabelScan	14	14	15	4	0	1.0000	other :Person
+NodeIndexSeek	1	1	3	4	1	0.8000	me :Person(name)

Total database accesses: 81

11.3.16. Let Semi Apply

The `LetSemiApply` operator tests for the presence of a pattern predicate, and is a variation of the `Apply` operator. When a query contains multiple pattern predicates separated with `OR`, `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. In the example, `LetSemiApply` will be used to check for the presence of the `FRIENDS_WITH` relationship from each person.

Query

```
MATCH (other:Person)
WHERE (other)-[:FRIENDS_WITH]->(:Person) OR (other)-[:WORKS_IN]->(:Location)
RETURN other.name
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	13	14	0	4	0	1.0000	anon[27], other, other.name	
+Projection	13	14	14	4	0	1.0000	other.name -- anon[27], other	{other.name : other.name}
+SelectOrSemiApply	13	14	0	4	0	1.0000	anon[27] -- other	'anon[27]'
+Filter	15	0	12	5	0	1.0000	NODE87, REL73, other	'NODE87` : Location
+Expand(All)	15	12	24	5	0	1.0000	NODE87, REL73 -- other	(other)-[REL73:WORKS_IN]->(:NODE87)
+Argument	14	12	0	5	0	1.0000	other	
+LetSemiApply	14	14	0	4	0	1.0000	anon[27] -- other	
+Filter	2	0	2	0	0	0.0000	NODE53, REL35, other	'NODE53` : Person
+Expand(All)	2	2	16	0	0	0.0000	NODE53, REL35 -- other	(other)-[REL35:FRIENDS_WITH]->(:NODE53)
+Argument	14	14	0	0	0	0.0000	other	
+NodeByLabelScan	14	14	15	5	0	1.0000	other	:Person

Total database accesses: 83

11.3.17. Let Anti Semi Apply

The [LetAntiSemiApply](#) operator tests for the absence of a pattern, and is a variation of the [Apply](#) operator. When a query contains multiple negated pattern predicates — i.e. predicates separated with **OR**, where at least one predicate contains **NOT** — [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. In the example, [LetAntiSemiApply](#) will be used to check for the absence of the **FRIENDS_WITH** relationship from each person.

Query

```
MATCH (other:Person)
WHERE NOT ((other)-[:FRIENDS_WITH]->(:Person)) OR (other)-[:WORKS_IN]->(:Location)
RETURN other.name
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other	
+ProduceResults	11	14	0	3	0	1.0000	anon[32], other, other.name		
+Projection	11	14	14	3	0	1.0000	other.name	-- anon[32], other	{other.name : other.name}
+SelectOrSemiApply	11	14	0	3	0	1.0000	anon[32]	-- other	`anon[32]`
+Filter	15	0	2	4	0	1.0000	NODE93, REL79, other		'NODE93':Location
+Expand(All)	15	2	4	4	0	1.0000	NODE93, REL79	-- other	(other)-[REL79:WORKS_IN]->(`NODE93`)
+Argument	14	2	0	4	0	1.0000	other		
+LetAntiSemiApply	14	14	0	3	0	1.0000	anon[32]	-- other	
+Filter	2	0	2	0	0	0.0000	NODE58, REL40, other		'NODE58':Person
+Expand(All)	2	2	16	0	0	0.0000	NODE58, REL40	-- other	(other)-[REL40:FRIENDS_WITH]->(`NODE58`)
+Argument	14	14	0	0	0	0.0000	other		
+NodeByLabelScan	14	14	15	4	0	1.0000	other		:Person

Total database accesses: 53

11.3.18. Select Or Semi Apply

The [SelectOrSemiApply](#) operator tests for the presence of a pattern predicate and evaluates a predicate, and is a variation of the [Apply](#) operator. This operator allows for the mixing of normal

predicates and pattern predicates that check for the presence of a pattern. First, the normal expression predicate is evaluated, and, only if it returns `false`, is the costly pattern predicate evaluated.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
							Other
+ProduceResults	11	2	0	3	0	1.0000	other, other.name
+Projection	11	2	2	3	0	1.0000	other.name -- other
+SelectOrSemiApply	11	2	14	3	0	1.0000	other other other.age > \$`AUTOTOINT0`
+Filter	2	0	2	0	0	0.0000	NODE71, REL53, other `NODE71`:Person
+Expand(All)	2	2	16	0	0	0.0000	NODE71, REL53 -- other (other)-[:FRIENDS_WITH]->(:NODE71)
+Argument	14	14	0	0	0	0.0000	other
+NodeByLabelScan	14	14	15	4	0	1.0000	other :Person

Total database accesses: 49

11.3.19. Select Or Anti Semi Apply

The `SelectOrAntiSemiApply` operator is used to evaluate `OR` between a predicate and a negative pattern predicate (i.e. a pattern predicate preceded with `NOT`), and is a variation of the `Apply` operator. If the predicate returns `true`, the pattern predicate is not tested. If the predicate returns `false` or `null`, `SelectOrAntiSemiApply` will instead test the pattern predicate.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	1.0000	4	12	0	3	0	other, other.name	
+Projection	1.0000	4	12	12	3	0	other.name -- other	{other.name : other.name}
+SelectOrAntiSemiApply	1.0000	4	12	14	3	0	other	other.age > \$`AUToint0`
+Filter	0.0000	2	0	2	0	0	NODE75, REL57, other	'NODE75':Person
+Expand(All)	0.0000	2	2	16	0	0	NODE75, REL57 -- other	(other)-[:FRIENDS_WITH]->(NODE75)
+Argument	0.0000	14	14	0	0	0	other	
+NodeByLabelScan	1.0000	14	14	15	4	0	other	:Person

Total database accesses: 59

11.3.20. Let Select Or Semi Apply

The [LetSelectOrSemiApply](#) operator is planned for pattern predicates that are combined with other predicates using OR. This is a variation of the [Apply](#) operator.

Query

```
MATCH (other:Person)
WHERE (other)-[:FRIENDS_WITH]->(:Person) OR (other)-[:WORKS_IN]->(:Location) OR other.age = 5
RETURN other.name
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
			Other				
+ProduceResults	13	14	0	4	0	1.0000	anon[27], other, other.name
+Projection	13	14	14	4	0	1.0000	other.name -- anon[27], other {other.name : other.name}
+SelectOrSemiApply	13	14	0	4	0	1.0000	anon[27] -- other `anon[27]`
+Filter	15	0	12	5	0	1.0000	NODE87, REL73, other `NODE87` : Location
+Expand(All)	15	12	24	5	0	1.0000	NODE87, REL73 -- other (other)-[REL73:WORKS_IN]->(NODE87)
+Argument	14	12	0	5	0	1.0000	other
+LetSelectOrSemiApply	14	14	14	4	0	1.0000	anon[27] -- other other.age = \$`AUToint0`
+Filter	2	0	2	0	0	0.0000	NODE53, REL35, other `NODE53` : Person
+Expand(All)	2	2	16	0	0	0.0000	NODE53, REL35 -- other (other)-[REL35:FRIENDS_WITH]->(NODE53)
+Argument	14	14	0	0	0	0.0000	other
+NodeByLabelScan	14	14	15	5	0	1.0000	other :Person

Total database accesses: 97

11.3.21. Let Select Or Anti Semi Apply

The `LetSelectOrAntiSemiApply` operator is planned for negated pattern predicates — i.e. pattern predicates preceded with `NOT` — that are combined with other predicates using `OR`. This operator is a variation of the `Apply` operator.

Query

```
MATCH (other:Person)
WHERE NOT (other)-[:FRIENDS_WITH]->(:Person) OR (other)-[:WORKS_IN]->(:Location) OR other.age = 5
RETURN other.name
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	11	14	0	3	0	1.0000	anon[31], other, other.name	
+Projection	11	14	14	3	0	1.0000	other.name -- anon[31], other	{other.name : other.name}
+SelectOrSemiApply	11	14	0	3	0	1.0000	anon[31] -- other	`anon[31]`
+Filter	15	0	2	4	0	1.0000	NODE91, REL77, other	`NODE91` : Location
+Expand(All)	15	2	4	4	0	1.0000	NODE91, REL77 -- other	(other)-[REL77:WORKS_IN]->(`NODE91`)
+Argument	14	2	0	4	0	1.0000	other	
+LetSelectOrAntiSemiApply	14	14	14	3	0	1.0000	anon[31] -- other	other.age = \$`AUTINT0`
+Filter	2	0	2	0	0	0.0000	NODE57, REL39, other	`NODE57` : Person
+Expand(All)	2	2	16	0	0	0.0000	NODE57, REL39 -- other	(other)-[REL39:FRIENDS_WITH]->(`NODE57`)
+Argument	14	14	0	0	0	0.0000	other	
+NodeByLabelScan	14	14	15	4	0	1.0000	other	: Person

Total database accesses: 67

11.3.22. Conditional Apply

The [ConditionalApply](#) operator checks whether a variable is not `null`, and if so, the right child operator will be executed. This operator is a variation of the [Apply](#) operator.

Query

```
MERGE (p:Person { name: 'Andres' })
ON MATCH SET p.exists = TRUE
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults		1	0	0	0	0		0
0.0000 p								
+EmptyResult		1	0	0	2	-1		
2.0000 p								
+AntiConditionalApply		1	1	0	2	-1		
2.0000 p								
+MergeCreateNode		1	0	0	0	0		0
0.0000 p								
+ConditionalApply		1	1	0	2	-1		
2.0000 p								
+SetProperty		1	1	3	2	-1		
2.0000 p								
+Argument		1	1	0	2	-1		
2.0000 p								
+Optional		1	1	0	2	0		
1.0000 p								
+ActiveRead		1	1	0	2	0		
1.0000 p								
+NodeIndexSeek		1	1	3	2	0		
1.0000 p		:Person(name)						

Total database accesses: 6

11.3.23. Anti Conditional Apply

The **AntiConditionalApply** operator checks whether a variable is `null`, and if so, the right child operator will be executed. This operator is a variation of the **Apply** operator.

Query

```
MERGE (p:Person { name: 'Andres' })
ON CREATE SET p.exists = TRUE
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	1	0	0	0	0	0	p	
+EmptyResult	1	0	0	0	0	0	p	
+AntiConditionalApply	1	1	0	0	0	0	p	
+SetProperty	1	0	0	0	0	0	p	
+MergeCreateNode	1	0	0	0	0	0	p	
+Optional	1	1	0	0	0	1	p	
+ActiveRead	1	1	0	0	0	1	p	
+NodeIndexSeek	1	1	3	0	0	1	:Person(name)	

Total database accesses: 3

11.3.24. Roll Up Apply

The **RollUpApply** operator is used to execute an expression which takes as input a pattern, and returns a list with content from the matched pattern; for example, when using a pattern expression or pattern comprehension in a query. This operator is a variation of the [Apply](#) operator.

Query

```
MATCH (p:Person)
RETURN p.name, [(p)-[:WORKS_IN]->(location)| location.name] AS cities
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	14	14	0	3	0		anon[33], cities, p, p.name	
+Projection	14	14	14	3	0		cities, p.name -- anon[33], p	{p.name : p.name, cities : 'anon[33]'}
+RollUpApply	14	14	0	3	0		anon[33] -- p	anon[33]
+Projection	0	15	15	0	0		anon[32] -- REL38, location, p	{ : location.name}
+Expand(All)	0	15	29	0	0		REL38, location -- p	(p)-[:REL38:WORKS_IN]->(location)
+Argument	1	14	0	0	0		p	
+NodeByLabelScan	14	14	15	4	0		p	:Person

Total database accesses: 73

11.3.25. Argument

The [Argument](#) operator indicates the variable to be used as an argument to the right-hand side of an [Apply](#) operator.

Query

```
MATCH (s:Person { name: 'me' })
MERGE (s)-[:FRIENDS_WITH]->(s)
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	1	1	1	1	0			

+ProduceResults		1	0	0	0	0
0.0000 anon[40], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+EmptyResult		1	0	0	4	1
0.8000 anon[40], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Apply		1	1	0	4	1
0.8000 anon[40], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+AntiConditionalApply		1	1	0	2	0
1.0000 anon[40], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+MergeCreateRelationship		1	1	1	2	0
1.0000 anon[40] -- s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Argument		1	1	0	2	0
1.0000 s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+AntiConditionalApply		1	1	0	2	0
1.0000 anon[40], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Optional		1	1	0	2	0
1.0000 anon[40], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+ActiveRead		0	0	0	0	0
0.0000 anon[40], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Expand(Into)		0	0	4	0	0
0.0000 anon[40] -- s (s)-[:FRIENDS_WITH]->(s)						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+LockNodes		1	0	0	0	0
0.0000 s	s					
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Argument		1	1	0	0	0
0.0000 s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Optional		1	1	0	4	0
1.0000 anon[40], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+ActiveRead		0	0	0	2	0
1.0000 anon[40], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Expand(Into)		0	0	4	2	0
1.0000 anon[40] -- s (s)-[:FRIENDS_WITH]->(s)						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Argument		1	1	0	2	0
1.0000 s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+NodeIndexSeek	:Person(name)	1	1	3	4	1
0.8000 s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+

Total database accesses: 12

11.3.26. Expand All

Given a start node, and depending on the pattern relationship, the `Expand(All)` operator will traverse incoming or outgoing relationships.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
		Other					
+ProduceResults	0.6667 anon[30], fof, p	0 1 0 2 1					
+Expand(All)	0.6667 anon[30], fof -- p (p)-[:FRIENDS_WITH]->(fof)	0 1 2 2 1					
+NodeIndexSeek	0.6667 p	1 1 3 2 1					
		:Person(name)					

Total database accesses: 5

11.3.27. Expand Into

When both the start and end node have already been found, the [Expand\(Into\)](#) operator is used to find all relationships connecting the two nodes. As both the start and end node of the relationship are already in scope, the node with the smallest degree will be used. This can make a noticeable difference when dense nodes appear as end points.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	0	0	0	2		1	anon[30], anon[53], fof, p	
+Filter	0	0	0	2		1	anon[30], anon[53], fof, p	not `anon[30]` = `anon[53]`
+Expand(Into)	0	0	0	2		1	anon[30]	-- anon[53], fof, p (p)-[:FRIENDS_WITH]->(fof)
+Expand(All)	0	0	1	2		1	anon[53], fof -- p	(p)<--(fof)
+NodeIndexSeek	1	1	3	2		1	p	:Person(name)

Total database accesses: 4

11.3.28. Optional Expand All

The `OptionalExpand(All)` operator is analogous to `Expand(All)`, apart from when no relationships match the direction, type and property predicates. In this situation, `OptionalExpand(all)` will return a single row with the relationship and end node set to `null`.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	14	15	0	4	0	1.0000	l, p, works_in	
+OptionalExpand(All)	14	15	44	4	0	1.0000	l, works_in -- p works_in.duration > \$` AUTOINT0` ; (p)-[works_in:WORKS_IN]->(1)	
+NodeByLabelScan	14	14	15	5	0	1.0000	p :Person	

Total database accesses: 59

11.3.29. Optional Expand Into

The `OptionalExpand(Into)` operator is analogous to `Expand(Into)`, apart from when no matching relationships are found. In this situation, `OptionalExpand(Into)` will return a single row with the relationship and end node set to `null`. As both the start and end node of the relationship are already in scope, the node with the smallest degree will be used. This can make a noticeable difference when dense nodes appear as end points.

Query

```
MATCH (p:Person)-[works_in:WORKS_IN]->(1)
OPTIONAL MATCH (1)-->(p)
RETURN p
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
				Other			
+ProduceResults	15	15	0	3	0	1.0000	anon[61], l, p, works_in
+OptionalExpand(Into)	15	15	48	3	0	1.0000	anon[61] -- l, p, works_in (l)-->(p)
+Expand(All)	15	15	29	3	0	1.0000	l, works_in -- p (p)-[works_in:WORKS_IN]->(l)
+NodeByLabelScan	14	14	15	4	0	1.0000	p :Person

Total database accesses: 92

11.3.30. VarLength Expand All

Given a start node, the `VarLengthExpand(All)` operator will traverse variable-length relationships.

Query

```
MATCH (p:Person)-[:FRIENDS_WITH *1..2]-(q:Person)
RETURN p, q
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	4	6	0	2	0	1.0000	anon[17], p, q	
+Filter	4	6	6	2	0	1.0000	anon[17], p, q	q:Person
+VarLengthExpand(All)	4	6	28	2	0	1.0000	anon[17], q -- p (p)-[:FRIENDS_WITH*..2]-(q)	
+NodeByLabelScan	14	14	15	3	0	1.0000	p	:Person

Total database accesses: 49

11.3.31. VarLength Expand Into

When both the start and end node have already been found, the `VarLengthExpand(Into)` operator is used to find all variable-length relationships connecting the two nodes.

Query

```
MATCH (p:Person)-[:FRIENDS_WITH *1..2]-(p:Person)
RETURN p
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	0	0	0	2	0	1.0000	anon[17], p	
+VarLengthExpand(Into)	0	0	28	2	0	1.0000	anon[17] -- p (p)-[:FRIENDS_WITH*..2]-(p)	
+NodeByLabelScan	14	14	15	3	0	1.0000	p :Person	

Total database accesses: 43

11.3.32. VarLength Expand Pruning

Given a start node, the `VarLengthExpand(Pruning)` operator will traverse variable-length relationships much like the `VarLengthExpand(All)` operator. However, as an optimization, some paths will not be explored if they are guaranteed to produce an end node that has already been found (by means of a previous path traversal). This will only be used in cases where the individual paths are not of interest. This operator guarantees that all the end nodes produced will be unique.

Query

```
MATCH (p:Person)-[:FRIENDS_WITH *3..4]-(q:Person)
RETURN DISTINCT p, q
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	0	0	0	2	0	1.0000	p, q	
+Distinct	0	0	0	2	0	1.0000	p, q	p, q
+Filter	0	0	0	2	0	1.0000	p, q	q:Person
+VarLengthExpand(Pruning)	0	0	0	0	0	0.0000	q -- p	(p)-[:FRIENDS_WITH*3..4]-(q)
+NodeByLabelScan	14	14	15	3	0	1.0000	p	:Person

Total database accesses: 15

11.3.33. Assert Same Node

The `AssertSameNode` operator is used to ensure that no unique constraints are violated. The example looks for the presence 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 unique 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

```
MERGE (t:Team { name: 'Engineering', id: 42 })
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	
Page Cache Hit Ratio	Variables	Other				
+ProduceResults	0.0000	t	1	0	0	0
+EmptyResult	0.0000	t	1	0	0	0
+AntiConditionalApply	0.0000	t	1	1	0	0
+MergeCreateNode	0.0000	t	1	0	0	0
+Optional	0.0000	t	1	1	0	0
+ActiveRead	0.0000	t	0	1	0	0
+AssertSameNode	0.0000	t	0	1	0	1
+NodeUniqueIndexSeek(Locking)	0.0000	t	1	1	2	0
+NodeUniqueIndexSeek(Locking)	0.0000	t	1	1	2	0

Total database accesses: 4

11.3.34. Drop Result

The `DropResult` operator produces zero rows. It is applied when it can be deduced through static analysis that the result of an expression will be empty, such as when a predicate guaranteed to return `false` (e.g. `1 > 5`) is used in a query.

Query

```
MATCH (p)
WHERE FALSE RETURN p
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
Variables						
+ProduceResults 0.0000 p	0	0	0	0	0	0
+DropResult 0.0000 p	0	0	0	0	0	0
+AllNodesScan 0.0000 p	35	0	0	0	0	0

Total database accesses: 0

11.3.35. Empty Result

The [EmptyResult](#) operator eagerly loads all incoming data and discards it.

Query

```
CREATE (:Person)
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
Variables						
+ProduceResults 0.0000 anon[8]	1	0	0	0	0	0
+EmptyResult 0.0000 anon[8]	1	0	0	0	0	0
+CreateNode 0.0000 anon[8]	1	1	2	0	0	0

Total database accesses: 2

11.3.36. Produce Results

The `ProduceResults` operator prepares the result so that it is consumable by the user, such as transforming internal values to user values. It is present in every single query that returns data to the user, and has little bearing on performance optimisation.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
0.0000	n	35	35	0	0	0	0
1.0000	n	35	35	36	1	0	0
Total database accesses: 36							

11.3.37. Load CSV

The `LoadCSV` operator loads data from a CSV source into the query. It is used whenever the `LOAD CSV` clause is used in a query.

Query

```
LOAD CSV FROM 'https://neo4j.com/docs/cypher-refcard/3.3/csv/artists.csv' AS line
RETURN line
```

Query Plan

```
Compiler CYPHER 3.4
```

```
Planner COST
```

```
Runtime INTERPRETED
```

```
Runtime version 3.4
```

Operator Ratio	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
Variables						
+ProduceResults	1	4	0	0	0	0
0.0000 line						
+LoadCSV	1	4	0	0	0	0
0.0000 line						

Total database accesses: 0

11.3.38. Hash joins in general

Hash joins have two inputs: the build input and probe input. The query planner assigns these roles so that the smaller of the two inputs is the build input. The build input is pulled in eagerly, and is used to build a probe table. Once this is complete, the probe table is checked for each row coming from the probe input side.

In query plans, the build input is always the left operator, and the probe input the right operator.

There are four hash join operators:

- [NodeHashJoin](#)
- [ValueHashJoin](#)
- [NodeLeftOuterHashJoin](#)
- [NodeRightOuterHashJoin](#)

11.3.39. Node Hash Join

The [NodeHashJoin](#) operator is a variation of the [hash join](#). [NodeHashJoin](#) executes the hash join on node ids. As primitive types and arrays can be used, it can be done very efficiently.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
				Other			
+ProduceResults	10	0	0	0	0	1.0000	anon[37], anon[56], andy, loc, loc.name, matt
+Projection	10	0	0	0	0	1.0000	loc.name -- anon[37], anon[56], andy, loc, matt {loc.name : loc.name}
+Filter	10	0	0	0	0	1.0000	anon[37], anon[56], andy, loc, matt not `anon[37]` = `anon[56]`
+NodeHashJoin	10	0	0	4	0	1.0000	anon[37], andy -- anon[56], loc, matt loc
+Expand(All)	19	0	0	1	0	1.0000	anon[56], loc -- matt (matt)-[:WORKS_IN]->(loc)
+NodeIndexSeek	1	0	2	1	0	1.0000	matt :Person(name)
+Expand(All)	19	0	1	0	0	0.0000	anon[37], loc -- andy (andy)-[:WORKS_IN]->(loc)
+NodeIndexSeek	1	1	3	0	0	0.0000	andy :Person(name)

Total database accesses: 6

11.3.40. Value Hash Join

The [ValueHashJoin](#) operator is a variation of the [hash join](#). This operator allows for arbitrary values to be used as the join key. It is most frequently used to solve predicates of the form: `n.prop1 = m.prop2` (i.e. equality predicates between two property columns).

Query

```
MATCH (p:Person),(q:Person)
WHERE p.age = q.age
RETURN p,q
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	0	0	0	0	0	0	p, q	0
+ValueHashJoin	0	0	28	1	0	1	p -- q	p.age = q.age
+NodeByLabelScan	14	14	15	1	0	1	:Person	q
+NodeByLabelScan	14	14	15	2	0	1	:Person	p

Total database accesses: 58

11.3.41. Node Left/Right Outer Hash Join

The [NodeLeftOuterHashJoin](#) and [NodeRightOuterHashJoin](#) operators are variations of the [hash join](#). The query below can be planned with either a left or a right outer join. The decision depends on the cardinalities of the left-hand and right-hand sides; i.e. how many rows would be returned, respectively, for `(a:Person)` and `(a)→(b:Person)`. If `(a:Person)` returns fewer results than `(a)→(b:Person)`, a left outer join — indicated by [NodeLeftOuterHashJoin](#) — is planned. On the other hand, if `(a:Person)` returns more results than `(a)→(b:Person)`, a right outer join — indicated by [NodeRightOuterHashJoin](#) — is planned instead.

Query

```
MATCH (a:Person)
OPTIONAL MATCH (a)-->(b:Person)
USING JOIN ON a
RETURN a.name, b.name
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
				Other			
+ProduceResults	14	14	0	1	0	1.0000	anon[36], a, a.name, b, b.name
+Projection	14	14	16	1	0	1.0000	a.name, b.name -- anon[36], a, b {a.name : a.name, b.name : b.name}
+NodeRightOuterHashJoin	14	14	0	4	0	1.0000	anon[36], b -- a a
+NodeByLabelScan	14	14	15	4	0	1.0000	a :Person
+Expand(All)	2	2	16	3	0	1.0000	anon[36], a -- b (b)<--(a)
+NodeByLabelScan	14	14	15	4	0	1.0000	b :Person

Total database accesses: 62

11.3.42. Triadic Selection

The **TriadicSelection** operator 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 into a set, and uses the set to check if the friend-of-friends are already connected to me. The example finds the names of all friends of my friends that are not already my friends.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	0	2	0	4	0	1.0000	anon[18], anon[35], anon[37], me, other, other.name	
+Projection	0	2	2	4	0	1.0000	other.name -- anon[18], anon[35], anon[37], me, other	{other.name : other.name}
+TriadicSelection	0	2	0	4	0	1.0000	anon[18], anon[35], anon[37], me, other	me, anon[35], other
+Filter	0	2	0	0	0	0.0000	anon[18], anon[35], anon[37], me, other	not `anon[18]` = `anon[37]`
+Expand(All)	0	6	10	0	0	0.0000	anon[37], other -- anon[18], anon[35], me	()-[:FRIENDS_WITH]-(other)
+Argument	4	4	0	0	0	0.0000	anon[18], anon[35], me	
+Expand(All)	4	4	18	1	0	1.0000	anon[18], anon[35] -- me	(me)-[:FRIENDS_WITH]-()
+NodeByLabelScan	14	14	15	0	0	0.0000	me	:Person

Total database accesses: 45

11.3.43. Cartesian Product

The [CartesianProduct](#) operator produces a cartesian product of the two inputs — each row coming from the left child operator will be combined with all the rows from the right child operator.

[CartesianProduct](#) generally exhibits bad performance and ought to be avoided if possible.

Query

```
MATCH (p:Person),(t:Team)
RETURN p, t
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	140	140	0	10	0	1.0000	p, t	
+CartesianProduct	140	140	0	11	0	1.0000	t -- p	
+NodeByLabelScan	14	140	150	1	0	1.0000	p	:Person
+NodeByLabelScan	10	10	11	11	0	1.0000	t	:Team

Total database accesses: 161

11.3.44. Foreach

The **Foreach** operator executes a nested loop between the left child operator and the right child operator. In an analogous manner to the **Apply** operator, it takes a row from the left-hand side and, using the **Argument** operator, provides it to the operator tree on the right-hand side. **Foreach** will yield all the rows coming in from the left-hand side; all results from the right-hand side are pulled in and discarded.

Query

```
FOREACH (value IN [1,2,3]| CREATE (:Person { age: value }))
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
+ProduceResults		1	0	0	0	0	0
0.0000							
+EmptyResult		1	0	0	0	0	0
0.0000							
+Foreach		1	1	0	0	0	0
0.0000							
+CreateNode	anon[36] -- value	1	3	12	0	0	0
0.0000							
+Argument		1	3	0	0	0	0
0.0000	value						
+EmptyRow		1	1	0	0	0	0
0.0000							

Total database accesses: 12

11.3.45. Eager

For isolation purposes, the **Eager** operator ensures that operations affecting subsequent operations are executed fully for the whole dataset before continuing execution. Information from the stores is fetched in a lazy manner; i.e. the pattern matching might not be fully exhausted before updates are applied. To guarantee reasonable semantics, the query planner will insert **Eager** operators into the query plan to prevent updates from influencing pattern matching; this scenario is exemplified by the query below, where the **DELETE** clause influences the **MATCH** clause. The **Eager** operator can cause high memory usage when importing data or migrating graph structures. In such cases, the operations should be split into simpler steps; e.g. importing nodes and relationships separately. Alternatively, the records to be updated can be returned, followed by an update statement.

Query

```
MATCH (a)-[r]-(b)
DELETE r,a,b
MERGE ()
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	1	0	0	0	0	0	anon[38], a, b, r	
0.0000								
+EmptyResult	1	0	0	0	0	0	anon[38], a, b, r	
0.0000								
+Apply	1	504	0	0	0	0	a, b, r -- anon[38]	
0.0000								
+AntiConditionalApply	1	504	0	0	0	0	anon[38]	
0.0000								
+MergeCreateNode	1	0	0	0	0	0	anon[38]	
0.0000								
+Optional	35	504	0	0	0	0	anon[38]	
0.0000								
+ActiveRead	35	504	0	0	0	0	anon[38]	
0.0000								
+AllNodesScan	35	504	540	0	0	0	anon[38]	
0.0000								
+Eager	36	36	0	15	0	0	a, b, r	
1.0000								
+Delete(3)	36	36	39	45	0	0	a, b, r	
3.0000								
+Eager	36	36	0	17	0	0	a, b, r	
1.0000								
+Expand(All)	36	36	71	2	0	0	b, r -- a	(a)-[r:]->(b)
1.0000								
+AllNodesScan	35	35	36	3	0	0	a	
1.0000								

Total database accesses: 686

11.3.46. Eager Aggregation

The [EagerAggregation](#) operator evaluates a grouping expression and uses the result to group rows into different groupings. For each of these groupings, [EagerAggregation](#) will then evaluate all aggregation

functions and return the result. To do this, [EagerAggregation](#), as the name implies, needs to pull in all data eagerly from its source and build up state, which leads to increased memory pressure in the system.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
+ProduceResults	4	6	0	0	0	1.0000	location, people
+EagerAggregation	4	6	30	3	0	1.0000	location, people location
+Filter	15	15	15	3	0	1.0000	anon[19], l, p p:Person
+Expand(AAll)	15	15	25	3	0	1.0000	anon[19], p -- l (l)-[:WORKS_IN]-(p)
+NodeByLabelScan	10	10	11	4	0	1.0000	l :Location

Total database accesses: 81

11.3.47. Node Count From Count Store

The [NodeCountFromCountStore](#) operator uses the count store to answer questions about node counts. This is much faster than the [EagerAggregation](#) operator which achieves the same result by actually counting. However, as the count store only stores a limited range of combinations, [EagerAggregation](#) 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

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	4	1	0	0	0	1.0000	people	
+NodeCountFromCountStore	4	1	1	0	0	1.0000	people	count((:Some(Person))) AS people

Total database accesses: 1

11.3.48. Relationship Count From Count Store

The [RelationshipCountFromCountStore](#) operator uses the count store to answer questions about relationship counts. This is much faster than the [EagerAggregation](#) operator which achieves the same result by actually counting. However, as the count store only stores a limited range of combinations, [EagerAggregation](#) 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

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	4	1	0	0	0	1.0000	jobs	
+RelationshipCountFromCountStore	4	1	2	0	0	1.0000	jobs	count((:Person)-[:WORKS_IN]->()) AS jobs

Total database accesses: 2

11.3.49. Distinct

The `Distinct` operator removes duplicate rows from the incoming stream of rows. To ensure only distinct elements are returned, `Distinct` will pull in data lazily from its source and build up state. This may lead to increased memory pressure in the system.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
							Other
+ProduceResults	14	6	0	2	0	1.0000	1
+Distinct	14	6	0	2	0	1.0000	1
+Filter	15	15	15	2	0	1.0000	anon[19], 1, p p:Person
+Expand(All)	15	15	25	2	0	1.0000	anon[19], p -- 1 (l)-[:WORKS_IN]-(p)
+NodeByLabelScan	10	10	11	3	0	1.0000	1 :Location

Total database accesses: 51

11.3.50. Filter

The `Filter` operator 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

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	14	0	0	2	0	1.0000	p	
+Filter	14	0	14	2	0	1.0000	p	p.name =~ \$`AUTOSTRING0`
+NodeIndexScan	14	14	16	2	4	0.3333	p	:Person(name)

Total database accesses: 30

11.3.51. Limit

The `Limit` operator returns the first 'n' rows from the incoming input.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	3	3	0	0	0	0.0000	p	
+Limit	3	3	0	0	0	0.0000	p	3
+NodeByLabelScan	14	3	4	0	0	0.0000	p	:Person

Total database accesses: 4

11.3.52. Skip

The **Skip** operator skips 'n' rows from the incoming rows.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
							Other
+ProduceResults	14	13	0	0	0	1.0000	anon[35], anon[59], p, p
+Projection	14	13	0	0	0	1.0000	p -- anon[35], anon[59], p {p : `anon[35]`}
+Skip	14	13	0	0	0	0.0000	anon[35], anon[59], p \$` AUTOINTO`
+Sort	14	14	0	2	0	1.0000	anon[35], anon[59], p anon[59]
+Projection	14	14	14	2	0	1.0000	anon[59] -- anon[35], p { : `anon[35]`, : `anon[35].id`}
+Projection	14	14	0	2	0	1.0000	anon[35] -- p { : `p`}
+NodeByLabelScan	14	14	15	3	0	1.0000	p :Person

Total database accesses: 29

11.3.53. Sort

The **Sort** operator sorts rows by a provided key. In order to sort the data, all data from the source operator needs to be pulled in eagerly and kept in the query state, which will lead to increased memory pressure in the system.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
				Other			
+ProduceResults	14	14	0	0	0	1	anon[24], anon[37], p, p
+Projection	14	14	0	0	0	1	p -- anon[24], anon[37], p {p : `anon[24]`}
+Sort	14	14	0	2	0	1	anon[24], anon[37], p anon[37]
+Projection	14	14	14	2	0	1	anon[37] -- anon[24], p { : `anon[24]`, : `anon[24].name`}
+Projection	14	14	0	2	0	1	anon[24] -- p { : `p`}
+NodeByLabelScan	14	14	15	3	0	1	p :Person

Total database accesses: 29

11.3.54. Top

The **Top** operator returns the first 'n' rows sorted by a provided key. Instead of sorting the entire input, only the top 'n' rows are retained.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
				Other			
+ProduceResults	2	2	0	0	0	1	anon[24], anon[37], p, p
0.0000							
+Projection	2	2	0	0	0	1	p -- anon[24], anon[37], p {p : `anon[24]`}
0.0000							
+Top	2	2	0	2	0	1	anon[24], anon[37], p anon[37]; 2
1.0000							
+Projection	14	14	14	2	0	1	anon[37] -- anon[24], p { : `anon[24]`, : `anon[24].name`}
1.0000							
+Projection	14	14	0	2	0	1	anon[24] -- p { : `p`}
1.0000							
+NodeByLabelScan	14	14	15	3	0	1	p :Person
1.0000							

Total database accesses: 29

11.3.55. Union

The **Union** operator concatenates the results from the right child operator with the results from the left child operator.

Query

```
MATCH (p:Location)
RETURN p.name
UNION ALL MATCH (p:Country)
RETURN p.name
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	10	11	0	3	0	1.0000	p.name	
+Union	10	11	0	4	0	1.0000	p.name	
+Projection	1	1	1	0	0	0.0000	p.name -- p {p.name : `p`.name}	
+NodeByLabelScan	1	1	2	1	0	1.0000	p :Country	
+Projection	10	10	10	2	0	1.0000	p.name -- p {p.name : `p`.name}	
+NodeByLabelScan	10	10	11	3	0	1.0000	p :Location	

Total database accesses: 24

11.3.56. Unwind

The **Unwind** operator returns one row per item in a list.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	10	5	0	0	0	0.0000	value	
+Unwind	10	5	0	0	0	0.0000	value	range(\$`AUTOINT0`, \$`AUTOINT1`)
Total database accesses: 0								

11.3.57. Lock Nodes

The `LockNodes` operator locks the start and end node when creating a relationship.

Query

```
MATCH (s:Person { name: 'me' })
MERGE (s)-[:FRIENDS_WITH]->(s)
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	1	0	0	0	0	0.0000	anon[40], s	
+EmptyResult	1	0	0	4	1	0.8000	anon[40], s	
+Apply	1	1	0	4	1	0.8000	anon[40], s	
+AntiConditionalApply	1	1	0	2	0	1.0000	anon[40], s	
+MergeCreateRelationship	1	1	1	2	0	1.0000	anon[40] -- s	
+Argument	1	1	0	2	0			

```

1.0000 | s           |           |           |           |
| | | +AntiConditionalApply |           1 |   1 |   0 |           2 |           0 |
1.0000 | anon[40], s |           |           |           |
| | | \\\             |           +-----+           +-----+
| | | +Optional      |           1 |   1 |   0 |           2 |           0 |
1.0000 | anon[40], s |           |           |           |
| | | |               |           +-----+           +-----+
| | | +ActiveRead    |           0 |   0 |   0 |           0 |           0 |
0.0000 | anon[40], s |           |           |           |
| | | |               |           +-----+           +-----+
| | | +Expand(Into)  |           0 |   0 |   4 |           0 |           0 |
0.0000 | anon[40] -- s | (s)-[:FRIENDS_WITH]->(s) |
| | | |               |           +-----+           +-----+
| | | +LockNodes    |           1 |   0 |   0 |           0 |           0 |
0.0000 | s           | s           |           |           |
| | | |               |           +-----+           +-----+
| | | +Argument     |           1 |   1 |   0 |           0 |           0 |
0.0000 | s           |           |           |           |
| | | |               |           +-----+           +-----+
| | +Optional       |           1 |   1 |   0 |           4 |           0 |
1.0000 | anon[40], s |           |           |           |
| | | |               |           +-----+           +-----+
| | +ActiveRead     |           0 |   0 |   0 |           2 |           0 |
1.0000 | anon[40], s |           |           |           |
| | | |               |           +-----+           +-----+
| | +Expand(Into)   |           0 |   0 |   4 |           2 |           0 |
1.0000 | anon[40] -- s | (s)-[:FRIENDS_WITH]->(s) |
| | | |               |           +-----+           +-----+
| | +Argument       |           1 |   1 |   0 |           2 |           0 |
1.0000 | s           |           |           |           |
| | | |               |           +-----+           +-----+
| +NodeIndexSeek   |           1 |   1 |   3 |           4 |           1 |
0.8000 | s           | :Person(name) |           |           |
+-----+           +-----+           +-----+
+-----+           +-----+

```

Total database accesses: 12

11.3.58. Optional

The **Optional** operator is used to solve some **OPTIONAL MATCH** queries. It will pull data from its source, simply passing it through if any data exists. However, if no data is returned by its source, **Optional** will yield a single row with all columns set to **null**.

Query

```

MATCH (p:Person { name:'me' })
OPTIONAL MATCH (q:Person { name: 'Lulu' })
RETURN p, q

```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	1	1	0	1	1	1	p, q	
+Apply	1	1	0	1	1	1	p -- q	
\								
+Optional	1	1	0	1	0	1	q	
+NodeIndexSeek	1	0	2	1	0	1	:Person(name)	q
+NodeIndexSeek	1	1	3	1	1	1	:Person(name)	p

Total database accesses: 5

11.3.59. Project Endpoints

The [ProjectEndpoints](#) operator projects the start and end node of a relationship.

Query

```
CREATE (n)-[p:KNOWS]->(m)
WITH p AS r
MATCH (u)-[r]->(v)
RETURN u, v
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Hit Ratio	Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache
				Other			
+ProduceResults	0.0000 m, n, p, r, u, v	18	1	0	0	0	0
+Apply	0.0000 m, n, p -- r, u, v	18	1	0	0	0	0
\							
+ProjectEndpoints	0.0000 u, v -- r	18	0	0	0	0	0
+Argument	0.0000 r	1	1	0	0	0	0
+Projection	0.0000 r -- m, n, p	1	1	0	0	0	0
+CreateRelationship	0.0000 p -- m, n	1	1	2	0	0	0
+CreateNode(2)	0.0000 m, n	1	1	4	0	0	0

Total database accesses: 6

11.3.60. Projection

For each incoming row, the **Projection** operator evaluates a set of expressions and produces a row with the results of the expressions.

Query

```
RETURN 'hello' AS greeting
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
Variables	Other					
+ProduceResults	1	1	0	0	0	0
0.0000 greeting						
+Projection	1	1	0	0	0	0
0.0000 greeting {greeting : \$`AUTOSTRING0`}						

Total database accesses: 0

11.3.61. Empty Row

The `EmptyRow` operator returns a single row with no columns.

Query

```
FOREACH (value IN [1,2,3]| CREATE (:Person { age: value }))
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
+ProduceResults	0.0000	1	0	0	0	0	0
+EmptyResult	0.0000	1	0	0	0	0	0
+Foreach	0.0000	1	1	0	0	0	0
+CreateNode	0.0000 anon[36] -- value	1	3	12	0	0	0
+Argument	0.0000 value	1	3	0	0	0	0
+EmptyRow	0.0000	1	1	0	0	0	0

Total database accesses: 12

11.3.62. Procedure Call

The `ProcedureCall` operator indicates an invocation to a procedure.

Query

```
CALL db.labels() YIELD label
RETURN *
ORDER BY label
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Estimated Rows Variables	Rows Other	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
+ProduceResults 0.0000	label	10000	4	0	0	0
+Sort 0.0000	label	10000 label	4	0	0	0
+ProcedureCall 0.0000	label	10000 db.labels() :: (label :: String)	4	1	0	0

Total database accesses: 1

11.3.63. Create Node

The [CreateNode](#) operator is used to create a node.

Query

```
CREATE (:Person { name: 'Jack' })
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Estimated Rows Variables	Rows Other	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
+ProduceResults 0.0000	anon[8]	1	0	0	0	0
+EmptyResult 0.0000	anon[8]	1	0	0	0	0
+CreateNode 0.0000	anon[8]	1	1	4	0	0

Total database accesses: 4

11.3.64. Create Relationship

The `CreateRelationship` operator is used to create a relationship.

Query

```
MATCH (a:Person { name: 'Max' }),(b:Person { name: 'Chris' })
CREATE (a)-[:FRIENDS_WITH]->(b)
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	1	0	0	0	0	0.0000	anon[70], a, b	
+EmptyResult	1	0	0	3	1	0.7500	anon[70], a, b	
+CreateRelationship	1	1	1	3	1	0.7500	anon[70] -- a, b	
+CartesianProduct	1	1	0	3	1	0.7500	a -- b	
+NodeIndexSeek	1	1	3	3	0	1.0000	b	:Person(name)
+NodeIndexSeek	1	1	3	3	1	0.7500	a	:Person(name)

Total database accesses: 7

11.3.65. Delete

The `Delete` operator is used to delete a node or a relationship.

Query

```
MATCH (me:Person { name: 'me' })-[w:WORKS_IN { duration: 190 }]->(london:Location { name: 'London' })
DELETE w
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	0	0	0	0	0	0	0.0000	london, me, w
+EmptyResult	0	0	0	-4	-2	0.6667	london, me, w	
+Delete	0	1	1	-4	-2	0.6667	london, me, w	
+Eager	0	1	0	0	0	0.0000	london, me, w	
+Filter	0	1	1	4	2	0.6667	london, me, w	w.duration = \$`AUToint1`
+Expand(Into)	0	1	4	4	2	0.6667	w -- london, me (me)-[w:WORKS_IN]->(london)	
+CartesianProduct	1	1	0	4	2	0.6667	me -- london	
+NodeIndexSeek	1	1	3	4	1	0.8000	london	:Location(name)
+NodeIndexSeek	1	1	3	4	2	0.6667	me	:Person(name)

Total database accesses: 12

11.3.66. Detach Delete

The **DetachDelete** operator is used in all queries containing the **DETACH DELETE** clause, when deleting nodes and their relationships.

Query

```
MATCH (p:Person)
DETACH DELETE p
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults	14	0	0	0	0	0	p	
+EmptyResult	14	0	0	15	0	1.0000	p	
+DetachDelete	14	14	0	15	0	1.0000	p	
+NodeByLabelScan	14	14	15	16	0	1.0000	:Person	

Total database accesses: 15

11.3.67. Merge Create Node

The `MergeCreateNode` operator is used when creating a node as a result of a `MERGE` clause failing to find the node.

Query

```
MERGE (:Person { name: 'Sally' })
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults		1	0	0	0	0	anon[7]	
0.0000								
+EmptyResult		1	0	0	0	0	anon[7]	
0.0000								
+AntiConditionalApply		1	1	0	0	0	anon[7]	
0.0000								
+MergeCreateNode		1	1	4	0	0	anon[7]	
0.0000								
+Optional		1	1	0	0	1	anon[7]	
0.0000								
+ActiveRead		1	0	0	0	1	anon[7]	
0.0000								
+NodeIndexSeek		1	0	2	0	1	:Person(name)	
0.0000								

Total database accesses: 6

11.3.68. Merge Create Relationship

The `MergeCreateRelationship` operator is used when creating a relationship as a result of a `MERGE` clause failing to find the relationship.

Query

```
MATCH (s:Person { name: 'Sally' })
MERGE (s)-[:FRIENDS_WITH]->(s)
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables	Other
+ProduceResults		1	0	0	0	0	anon[7]	
0.0000								
+EmptyResult		1	0	0	0	0	anon[7]	
0.0000								
+AntiConditionalApply		1	1	0	0	0	anon[7]	
0.0000								
+MergeCreateRelationship		1	1	1	0	1	anon[7]	
0.0000								

+ProduceResults		1	0	0	0	0
0.0000 anon[43], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+EmptyResult		1	0	0	0	1
0.0000 anon[43], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Apply		1	0	0	0	1
0.0000 anon[43], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+AntiConditionalApply		1	0	0	0	0
0.0000 anon[43], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+MergeCreateRelationship		1	0	0	0	0
0.0000 anon[43] -- s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Argument		1	0	0	0	0
0.0000 s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+AntiConditionalApply		1	0	0	0	0
0.0000 anon[43], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Optional		1	0	0	0	0
0.0000 anon[43], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+ActiveRead		0	0	0	0	0
0.0000 anon[43], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Expand(Into)		0	0	0	0	0
0.0000 anon[43] -- s (s)-[:FRIENDS_WITH]->(s)						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+LockNodes		1	0	0	0	0
0.0000 s	s					
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Argument		1	0	0	0	0
0.0000 s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Optional		1	0	0	0	0
0.0000 anon[43], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+ActiveRead		0	0	0	0	0
0.0000 anon[43], s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Expand(Into)		0	0	0	0	0
0.0000 anon[43] -- s (s)-[:FRIENDS_WITH]->(s)						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+Argument		1	0	0	0	0
0.0000 s						
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+
+NodeIndexSeek		1	0	2	0	1
0.0000 s	:Person(name)					
+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+	+-----+-----+

Total database accesses: 2

11.3.69. Set Labels

The **SetLabels** operator is used when setting labels on a node.

Query

```
MATCH (n)
SET n:Person
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Estimated Rows Variables	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
+ProduceResults 0.0000 n	35	0	0	0	0	0
+EmptyResult 1.0000 n	35	0	0	2	0	0
+SetLabels 1.0000 n	35	35	35	2	0	0
+AllNodesScan 1.0000 n	35	35	36	3	0	0

Total database accesses: 71

11.3.70. Remove Labels

The `RemoveLabels` operator is used when deleting labels from a node.

Query

```
MATCH (n)
REMOVE n:Person
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Variables	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
+ProduceResults	0.0000 n	35	0	0	0	0	0
+EmptyResult	1.0000 n	35	0	0	2	0	0
+RemoveLabels	1.0000 n	35	35	35	2	0	0
+AllNodesScan	1.0000 n	35	35	36	3	0	0

Total database accesses: 71

11.3.71. Set Node Property From Map

The `SetNodePropertyFromMap` operator is used when setting properties from a map on a node.

Query

```
MATCH (n)
SET n = { weekday: 'Monday', meal: 'Lunch' }
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
+ProduceResults	35	0	0	0	0	0.0000	n
+EmptyResult	35	0	0	3	0	1.0000	n
+SetNodePropertyFromMap	35	35	211	3	0	1.0000	n
+AllNodesScan	35	35	36	4	0	1.0000	n

Total database accesses: 247

11.3.72. Set Relationship Property From Map

The `SetRelationshipPropertyFromMap` operator is used when setting properties from a map on a relationship.

Query

```
MATCH (n)-[r]->(m)
SET r = { weight: 5, unit: 'kg' }
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	
Page Cache Hit Ratio	Variables	Other				
+ProduceResults	18	0	0	0	0	0
0.0000 m, n, r						
+EmptyResult	18	0	0	3	0	0
1.0000 m, n, r						
+SetRelationshipPropertyFromMap	18	18	105	3	0	0
1.0000 m, n, r						
+Expand(All)	18	18	53	3	0	0
1.0000 n, r -- m (m)<-[r:]->(n)						
+AllNodesScan	35	35	36	4	0	0
1.0000 m						

Total database accesses: 194

11.3.73. Set Property

The `SetProperty` operator is used when setting a property on a node or relationship.

Query

```
MATCH (n)
SET n.checked = TRUE
```

Query Plan

Compiler CYPHER 3.4

Planner COST

Runtime INTERPRETED

Runtime version 3.4

Operator Ratio	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit
Variables						
+ProduceResults 0.0000 n	35	0	0	0	0	0
+EmptyResult 1.0000 n	35	0	0	1	0	0
+SetProperty 1.0000 n	35	35	38	1	0	0
+AllNodesScan 1.0000 n	35	35	36	2	0	0

Total database accesses: 74

11.3.74. Create Unique Constraint

The `CreateUniqueConstraint` operator creates a unique constraint on a property for all nodes having a certain label. 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

Compiler CYPHER 3.4

Planner PROCEDURE

Runtime PROCEDURE

Runtime version 3.4

Operator
+CreateUniqueConstraint

Total database accesses: ?

11.3.75. Drop Unique Constraint

The `DropUniqueConstraint` operator removes a unique constraint from a property for all nodes having a certain label. The following query will drop a unique constraint on the `name` property of nodes with the `Country` label.

Query

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

Query Plan

Compiler CYPHER 3.4

Planner PROCEDURE

Runtime PROCEDURE

Runtime version 3.4

Operator
+DropUniqueConstraint

Total database accesses: ?

11.3.76. Create Node Property Existence Constraint

The [CreateNodePropertyExistenceConstraint](#) operator creates an existence constraint on a property for all nodes having a certain label. This will only appear in Enterprise Edition.

Query

```
CREATE CONSTRAINT ON (p:Person) ASSERT exists(p.name)
```

Query Plan

Compiler CYPHER 3.4

Planner PROCEDURE

Runtime PROCEDURE

Runtime version 3.4

Operator
+CreateNodePropertyExistenceConstraint

Total database accesses: ?

11.3.77. Drop Node Property Existence Constraint

The [DropNodePropertyExistenceConstraint](#) operator removes an existence constraint from a property for all nodes having a certain label. This will only appear in Enterprise Edition.

Query

```
DROP CONSTRAINT ON (p:Person) ASSERT exists(p.name)
```

Query Plan

```
Compiler CYPHER 3.4
Planner PROCEDURE
Runtime PROCEDURE
Runtime version 3.4
+-----+
| Operator          |
+-----+
| +DropNodePropertyExistenceConstraint |
+-----+
Total database accesses: ?
```

11.3.78. Create Node Key Constraint

The `CreateNodeKeyConstraint` operator creates a Node Key which ensures 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. This will only appear in Enterprise Edition.

Query

```
CREATE CONSTRAINT ON (e:Employee) ASSERT (e.firstname, e.surname) IS NODE KEY
```

Query Plan

```
Compiler CYPHER 3.4
Planner PROCEDURE
Runtime PROCEDURE
Runtime version 3.4
+-----+
| Operator          |
+-----+
| +CreateNodeKeyConstraint |
+-----+
Total database accesses: ?
```

11.3.79. Drop Node Key Constraint

The `DropNodeKeyConstraint` operator removes a Node Key from a set of properties for all nodes having a certain label. This will only appear in Enterprise Edition.

Query

```
DROP CONSTRAINT ON (e:Employee) ASSERT (e.firstname, e.surname) IS NODE KEY
```

Query Plan

```
Compiler CYPHER 3.4
```

```
Planner PROCEDURE
```

```
Runtime PROCEDURE
```

```
Runtime version 3.4
```

Operator
+DropNodeKeyConstraint

```
Total database accesses: ?
```

11.3.80. Create Relationship Property Existence Constraint

The `CreateRelationshipPropertyExistenceConstraint` operator creates an existence constraint on a property for all relationships of a certain type. This will only appear in Enterprise Edition.

Query

```
CREATE CONSTRAINT ON ()-[l:LIKED]-() ASSERT exists(l.when)
```

Query Plan

```
Compiler CYPHER 3.4
```

```
Planner PROCEDURE
```

```
Runtime PROCEDURE
```

```
Runtime version 3.4
```

Operator
+CreateRelationshipPropertyExistenceConstraint

```
Total database accesses: ?
```

11.3.81. Drop Relationship Property Existence Constraint

The `DropRelationshipPropertyExistenceConstraint` operator removes an existence constraint from a property for all relationships of a certain type. This will only appear in Enterprise Edition.

Query

```
DROP CONSTRAINT ON ()-[l:LIKED]-() ASSERT exists(l.when)
```

Query Plan

```
Compiler CYPHER 3.4
Planner PROCEDURE
Runtime PROCEDURE
Runtime version 3.4
+-----+
| Operator          |
+-----+
| +DropRelationshipPropertyExistenceConstraint |
+-----+
Total database accesses: ?
```

11.3.82. Create Index

The `CreateIndex` operator creates an index on a property for all nodes having a certain label. The following query will create an index on the `name` property of nodes with the `Country` label.

Query

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

Query Plan

```
Compiler CYPHER 3.4
Planner PROCEDURE
Runtime PROCEDURE
Runtime version 3.4
+-----+
| Operator          |
+-----+
| +CreateIndex      |
+-----+
Total database accesses: ?
```

11.3.83. Drop Index

The `DropIndex` operator removes an index from a property for all nodes having a certain label. The following query will drop an index on the `name` property of nodes with the `Country` label.

Query

```
DROP INDEX ON :Country(name)
```

Query Plan

```
Compiler CYPHER 3.4
Planner PROCEDURE
Runtime PROCEDURE
Runtime version 3.4
+-----+
| Operator   |
+-----+
| +DropIndex |
+-----+
Total database accesses: ?
```

11.4. 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](#)

11.4.1. Shortest path with fast algorithm

Query

```
MATCH (ms:Person { name: 'Martin Sheen' }), (cs:Person { name: 'Charlie Sheen' }), p = shortestPath((ms)-[:ACTED_IN*]-(cs))
WHERE ALL (r IN relationships(p) 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

```
Compiler CYPHER 3.4
```

```
Planner COST
```

```
Runtime SLOTTED
```

```
Runtime version 3.4
```

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
							Other
+ProduceResults	1	1	0	12	1	0.9231	anon[104], cs, ms, p
+ShortestPath	1	1	5	12	1	0.9231	anon[104], p -- cs, ms {p0 : all(r IN relationships(p) WHERE exists(r.role))}
+CartesianProduct	1	1	0	12	1	0.9231	ms -- cs
+NodeIndexSeek	1	1	3	12	0	1.0000	cs :Person(name)
+NodeIndexSeek	1	1	3	12	1	0.9231	ms :Person(name)

Total database accesses: 11

11.4.2. 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

```
Compiler CYPHER 3.4
```

```
Planner COST
```

```
Runtime SLOTTED
```

```
Runtime version 3.4
```

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
				Other			
+ProduceResults	0	1	0	10	0	1.0000	anon[86], anon[104], cs, ms, p
+AntiConditionalApply	0	1	0	10	0	1.0000	anon[86], anon[104], cs, ms, p
+Top	0	0	0	0	0	0.0000	anon[86], anon[104], cs, ms, p anon[86]; 1
+Projection	0	0	0	0	0	0.0000	anon[86] -- anon[104], cs, ms, p { : length(p)}
+Filter	0	0	0	0	0	0.0000	anon[104], cs, ms, p length(p) > \$` AUTOINT2`
+Projection	0	0	0	0	0	0.0000	anon[104], cs, ms, p {p : PathExpression(NodePathStep(NodeFromSlot(0,cs),MultiRelationshipPathStep(ReferenceFromSlot(1,anon[104]),BOTH,NilPathStep))))}
+VarLengthExpand(Into)	0	0	0	0	0	0.0000	anon[104], cs, ms, p (cs)-[:*]-(ms)
+Argument	1	0	0	0	0	0.0000	anon[104], cs, ms, p
+Apply	1	1	0	10	0	1.0000	anon[104], cs, ms, p
+Optional	1	1	0	8	0	1.0000	anon[104], cs, ms, p

```

+-----+
| | +ShortestPath      |          0 |   1 |   1 |     8 |          0 |
1.0000 | anon[104], p -- cs, ms | {p0 : length(p) > $` AUTOINT2`}
| | |           +-----+
+-----+
+-----+
| | +Argument        |          1 |   1 |     0 |     8 |          0 |
1.0000 | cs, ms
| | |           +-----+
+-----+
+-----+
| +CartesianProduct  |          1 |   1 |     0 |    10 |          0 |
1.0000 | cs -- ms
| | |           +-----+
+-----+
+-----+
| | +NodeIndexSeek   |          1 |   1 |     3 |     9 |          0 |
1.0000 | ms           | :Person(name)
| | |           +-----+
+-----+
+-----+
| +NodeIndexSeek   |          1 |   1 |     3 |    10 |          0 |
1.0000 | cs           | :Person(name)
| | |           +-----+
+-----+
+-----+
Total database accesses: 7

```

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.

An [ErrorPlan](#) operator will appear in the execution plan in cases where (i) `cypher.forbid_exhaustive_shortestpath` is set to `true`, and (ii) the fast algorithm is not able to find the shortest 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

Compiler CYPHER 3.4

Planner COST

Runtime SLOTTED

Runtime version 3.4

Operator	Estimated Rows	Rows	DB Hits	Page Cache Hits	Page Cache Misses	Page Cache Hit Ratio	Variables
				Other			
+ProduceResults	1	1	0	10	0	100.00%	anon[136], anon[104], cs, ms, p
+Filter	1	1	0	10	0	100.00%	anon[136], anon[104], cs, ms, p anon[136]
+Projection	1	1	0	10	0	100.00%	anon[136] -- anon[104], cs, ms, p {p : p, : length(p) > \$`AUToint2`}
+ShortestPath	1	1	1	10	0	100.00%	anon[104], p -- cs, ms {}
+CartesianProduct	1	1	0	10	0	100.00%	cs -- ms
+NodeIndexSeek	1	1	3	9	0	100.00%	ms :Person(name)
+NodeIndexSeek	1	1	3	10	0	100.00%	cs :Person(name)

Total database accesses: 7

Chapter 12. Deprecations, additions and compatibility

Cypher is a language that is constantly evolving. New features get added to the language continuously, and occasionally, some features become deprecated and are subsequently removed.

- Removals, deprecations, additions and extensions
 - Version 3.0
 - Version 3.1
 - Version 3.2
 - Version 3.3
 - Version 3.4
- Compatibility
- Supported language versions

12.1. Removals, deprecations, additions and extensions

The following tables lists all the features which have been removed, deprecated, added or extended in Cypher. Replacement syntax for deprecated and removed features are also indicated.

12.1.1. Version 3.0

Feature	Type	Change	Details
<code>has()</code>	Function	Removed	Replaced by <code>exists()</code>
<code>str()</code>	Function	Removed	Replaced by <code>toString()</code>
<code>{parameter}</code>	Syntax	Deprecated	Replaced by <code>\$parameter</code>
<code>properties()</code>	Function	Added	
<code>CALL [...YIELD]</code>	Clause	Added	
<code>point() - Cartesian 2D</code>	Function	Added	
<code>point() - WGS 84 2D</code>	Function	Added	
<code>distance()</code>	Function	Added	
User-defined procedures	Functionality	Added	
<code>toString()</code>	Function	Extended	Now also allows Boolean values as input

12.1.2. Version 3.1

Feature	Type	Change	Details
<code>rels()</code>	Function	Deprecated	Replaced by <code>relationships()</code>
<code>toInt()</code>	Function	Deprecated	Replaced by <code>toInteger()</code>
<code>lower()</code>	Function	Deprecated	Replaced by <code>toLower()</code>
<code>upper()</code>	Function	Deprecated	Replaced by <code>toUpper()</code>

Feature	Type	Change	Details
<code>toBoolean()</code>	Function	Added	
Map projection	Syntax	Added	
Pattern comprehension	Syntax	Added	
User-defined functions	Functionality	Added	
<code>CALL...YIELD...WHERE</code>	Clause	Extended	Records returned by <code>YIELD</code> may be filtered further using <code>WHERE</code>

12.1.3. Version 3.2

Feature	Type	Change	Details
<code>CYPHER planner=rule</code> (Rule planner)	Functionality	Removed	All queries now use the cost planner. Any query prepended thus will fall back to using Cypher 3.1.
<code>CREATE UNIQUE</code>	Clause	Removed	Running such queries will fall back to using Cypher 3.1 (and use the rule planner)
<code>START</code>	Clause	Removed	Running such queries will fall back to using Cypher 3.1 (and use the rule planner)
<code>MATCH (n)-[rs*]-() RETURN rs</code>	Syntax	Deprecated	Replaced by <code>MATCH p=(n)-[*]-() RETURN relationships(p) AS rs</code>
<code>MATCH (n)-[:A B C {foo: 'bar'}]-() RETURN n</code>	Syntax	Deprecated	Replaced by <code>MATCH (n)-[:A B C {foo: 'bar'}]-() RETURN n</code>
<code>MATCH (n)-[x:A B C]-() RETURN n</code>	Syntax	Deprecated	Replaced by <code>MATCH (n)-[x:A B C]-() RETURN n</code>
<code>MATCH (n)-[x:A B C*]-() RETURN n</code>	Syntax	Deprecated	Replaced by <code>MATCH (n)-[x:A B C*]-() RETURN n</code>
User-defined aggregation functions	Functionality	Added	
Composite indexes	Index	Added	
Node Key	Index	Added	Neo4j Enterprise Edition only
<code>CYPHER runtime=compiled</code> (Compiled runtime)	Functionality	Added	Neo4j Enterprise Edition only
<code>reverse()</code>	Function	Extended	Now also allows a list as input
<code>max(), min()</code>	Function	Extended	Now also supports aggregation over a set containing both strings and numbers

12.1.4. Version 3.3

Feature	Type	Change	Details
<code>START</code>	Clause	Removed	As in Cypher 3.2, any queries using the <code>START</code> clause will revert back to Cypher 3.1 <code>planner=rule</code> . However, there are built-in procedures for accessing explicit indexes that will enable users to use the current version of Cypher and the cost planner together with these indexes. An example of this is <code>CALL db.index.explicit.searchNodes('my_index', 'email:me*')</code> .
<code>CYPHER runtime=slotted</code> (Faster interpreted runtime)	Functionality	Added	Neo4j Enterprise Edition only
<code>max()</code> , <code>min()</code>	Function	Extended	Now also supports aggregation over sets containing lists of strings and/or numbers, as well as over sets containing strings, numbers, and lists of strings and/or numbers

12.1.5. Version 3.4

Feature	Type	Change	Details
Spatial point types	Functionality	Amendment	A point — irrespective of which Coordinate Reference System is used — can be stored as a property and is able to be backed by an index. Prior to this, a point was a virtual property only.
point() - Cartesian 3D	Function	Added	
point() - WGS 84 3D	Function	Added	
randomUUID()	Function	Added	
Temporal types	Functionality	Added	Supports storing, indexing and working with the following temporal types: Date, Time, LocalTime, DateTime, LocalDateTime and Duration.
Temporal functions	Functionality	Added	Functions allowing for the creation and manipulation of values for each temporal type — Date, Time, LocalTime, DateTime, LocalDateTime and Duration.
Temporal operators	Functionality	Added	Operators allowing for the manipulation of values for each temporal type — Date, Time, LocalTime, DateTime, LocalDateTime and Duration.
toString()	Function	Extended	Now also allows temporal values as input (i.e. values of type Date, Time, LocalTime, DateTime, LocalDateTime or Duration).

12.2. Compatibility

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 method is to set the version for a particular query. Prepending a query with `CYPHER 2.3` will execute the query with the version of Cypher included in Neo4j 2.3.

Below is an example using the `has()` function:

```
CYPHER 2.3
MATCH (n:Person)
WHERE has(n.age)
RETURN n.name, n.age
```

12.3. Supported language versions

Neo4j 3.4 supports the following versions of the Cypher language:

- Neo4j Cypher 3.4
- Neo4j Cypher 3.3
- Neo4j Cypher 3.2
- 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 13. 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](#)
- [Cypher query options](#)

13.1. Clauses

Clause	Category	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 (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 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 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>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	Category	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 from 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 from 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 [ASC[ENDING] DESC[ENDING]]	Reading sub-clause	A sub-clause following RETURN or WITH , specifying that the output should be sorted in either ascending (the default) or descending order.
REMOVE	Writing	Remove properties and labels from nodes and relationships.
RETURN ... [AS]	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.
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 ... [AS]	Projecting	Expands a list into a sequence of rows.

Clause	Category	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 ... [AS]	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.

13.2. Operators

Operator	Category	Description
%	Mathematical	Modulo division
*	Mathematical	Multiplication
*	Temporal	Multiplying a duration with a number
+	Mathematical	Addition
+	String	Concatenation
+	List	Concatenation
+	Temporal	Adding two durations, or a duration and a temporal instant
-	Mathematical	Subtraction or unary minus
-	Temporal	Subtracting a duration from a temporal instant or from another duration
.	General	Property access
/	Mathematical	Division
/	Temporal	Dividing a duration by a number
<	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

Operator	Category	Description
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
NOT	Boolean	Negation
OR	Boolean	Disjunction
STARTS WITH	String comparison	Case-sensitive prefix search
XOR	Boolean	Exclusive disjunction
[]	General	Subscript (dynamic property access)
[]	List	Subscript (accessing element(s) in a list)
^	Mathematical	Exponentiation

13.3. Functions

Function	Category	Description
abs()	Numeric	Returns the absolute value of a number.
acos()	Trigonometric	Returns the arccosine of a number in radians.
all()	Predicate	Tests whether the predicate holds for all elements in a list.
any()	Predicate	Tests whether the predicate holds for at least one element in a list.
asin()	Trigonometric	Returns the arcsine of a number in radians.
atan()	Trigonometric	Returns the arctangent of a number in radians.
atan2()	Trigonometric	Returns the arctangent2 of a set of coordinates in radians.
avg()	Aggregating	Returns the average of a set of values.
ceil()	Numeric	Returns the smallest floating point number that is greater than or equal to a number and equal to a mathematical integer.
coalesce()	Scalar	Returns the first non- <code>null</code> value in a list of expressions.
collect()	Aggregating	Returns a list containing the values returned by an expression.
cos()	Trigonometric	Returns the cosine of a number.
cot()	Trigonometric	Returns the cotangent of a number.
count()	Aggregating	Returns the number of values or rows.
date()	Temporal	Returns the current <i>Date</i> .
date({year [, month, day]})	Temporal	Returns a calendar (Year-Month-Day) <i>Date</i> .
date({year [, week, dayOfWeek]})	Temporal	Returns a week (Year-Week-Day) <i>Date</i> .

Function	Category	Description
<code>date({year [, quarter, dayOfQuarter]})</code>	Temporal	Returns a quarter (Year-Quarter-Day) <i>Date</i> .
<code>date({year [, ordinalDay]})</code>	Temporal	Returns an ordinal (Year-Day) <i>Date</i> .
<code>date(string)</code>	Temporal	Returns a <i>Date</i> by parsing a string.
<code>date({map})</code>	Temporal	Returns a <i>Date</i> from a map of another temporal value's components.
<code>date.realtime()</code>	Temporal	Returns the current <i>Date</i> using the <code>realtime</code> clock.
<code>date.statement()</code>	Temporal	Returns the current <i>Date</i> using the <code>statement</code> clock.
<code>date.transaction()</code>	Temporal	Returns the current <i>Date</i> using the <code>transaction</code> clock.
<code>date.truncate()</code>	Temporal	Returns a <i>Date</i> obtained by truncating a value at a specific component boundary. Truncation summary .
<code>datetime()</code>	Temporal	Returns the current <i>DateTime</i> .
<code>datetime({year [, month, day, ...]})</code>	Temporal	Returns a calendar (Year-Month-Day) <i>DateTime</i> .
<code>datetime({year [, week, dayOfWeek, ...]})</code>	Temporal	Returns a week (Year-Week-Day) <i>DateTime</i> .
<code>datetime({year [, quarter, dayOfQuarter, ...]})</code>	Temporal	Returns a quarter (Year-Quarter-Day) <i>DateTime</i> .
<code>datetime({year [, ordinalDay, ...]})</code>	Temporal	Returns an ordinal (Year-Day) <i>DateTime</i> .
<code>datetime(string)</code>	Temporal	Returns a <i>DateTime</i> by parsing a string.
<code>datetime({map})</code>	Temporal	Returns a <i>DateTime</i> from a map of another temporal value's components.
<code>datetime({epochSeconds})</code>	Temporal	Returns a <i>DateTime</i> from a timestamp.
<code>datetime.realtime()</code>	Temporal	Returns the current <i>DateTime</i> using the <code>realtime</code> clock.
<code>datetime.statement()</code>	Temporal	Returns the current <i>DateTime</i> using the <code>statement</code> clock.
<code>datetime.transaction()</code>	Temporal	Returns the current <i>DateTime</i> using the <code>transaction</code> clock.
<code>datetime.truncate()</code>	Temporal	Returns a <i>DateTime</i> obtained by truncating a value at a specific component boundary. Truncation summary .
<code>degrees()</code>	Trigonometric	Converts radians to degrees.
<code>distance()</code>	Spatial	Returns a floating point number representing the geodesic distance between any two points in the same CRS.
<code>duration({map})</code>	Temporal	Returns a <i>Duration</i> from a map of its components.
<code>duration(string)</code>	Temporal	Returns a <i>Duration</i> by parsing a string.
<code>duration.between()</code>	Temporal	Returns a <i>Duration</i> equal to the difference between two given instants.
<code>duration.inDays()</code>	Temporal	Returns a <i>Duration</i> equal to the difference in whole days or weeks between two given instants.

Function	Category	Description
duration.inMonths()	Temporal	Returns a <i>Duration</i> equal to the difference in whole months, quarters or years between two given instants.
duration.inSeconds()	Temporal	Returns a <i>Duration</i> equal to the difference in seconds and fractions of seconds, or minutes or hours, between two given instants.
e()	Logarithmic	Returns the base of the natural logarithm, e.
endNode()	Scalar	Returns the end node of a relationship.
exists()	Predicate	Returns true if a match for the pattern exists in the graph, or if the specified property exists in the node, relationship or map.
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 list l_{result} containing the values resulting from an expression which has been applied to each element in a list $list$.
filter()	List	Returns a list l_{result} containing all the elements from a list $list$ that comply with a predicate.
floor()	Numeric	Returns the largest floating point number that is less than or equal to a number and equal to a mathematical integer.
haversin()	Trigonometric	Returns half the versine of a number.
head()	Scalar	Returns the first element in a list.
id()	Scalar	Returns the id of a relationship or node.
keys()	List	Returns a list containing the string representations for all the property names of a node, relationship, or map.
labels()	List	Returns a list containing the string representations for all the labels of a node.
last()	Scalar	Returns the last element in a list.
left()	String	Returns a string containing the specified number of leftmost characters of the original string.
length()	Scalar	Returns the length of a path.
localdatetime()	Temporal	Returns the current <i>LocalDateTime</i> .
localdatetime({year [, month, day, ...]})	Temporal	Returns a calendar (Year-Month-Day) <i>LocalDateTime</i> .
localdatetime({year [, week, dayOfWeek, ...]})	Temporal	Returns a week (Year-Week-Day) <i>LocalDateTime</i> .
localdatetime({year [, quarter, dayOfQuarter, ...]})	Temporal	Returns a quarter (Year-Quarter-Day) <i>DateTime</i> .
localdatetime({year [, ordinalDay, ...]})	Temporal	Returns an ordinal (Year-Day) <i>LocalDateTime</i> .
localdatetime(string)	Temporal	Returns a <i>LocalDateTime</i> by parsing a string.

Function	Category	Description
<code>localdatetime({map})</code>	Temporal	Returns a <i>LocalDateTime</i> from a map of another temporal value's components.
<code>localdatetime.realtime()</code>	Temporal	Returns the current <i>LocalDateTime</i> using the <code>realtime</code> clock.
<code>localdatetime.statement()</code>	Temporal	Returns the current <i>LocalDateTime</i> using the <code>statement</code> clock.
<code>localdatetime.transaction()</code>	Temporal	Returns the current <i>LocalDateTime</i> using the <code>transaction</code> clock.
<code>localdatetime.truncate()</code>	Temporal	Returns a <i>LocalDateTime</i> obtained by truncating a value at a specific component boundary. Truncation summary .
<code>localtime()</code>	Temporal	Returns the current <i>LocalTime</i> .
<code>localtime({hour [, minute, second, ...]})</code>	Temporal	Returns a <i>LocalTime</i> with the specified component values.
<code>localtime(string)</code>	Temporal	Returns a <i>LocalTime</i> by parsing a string.
<code>localtime({time [, hour, ...]})</code>	Temporal	Returns a <i>LocalTime</i> from a map of another temporal value's components.
<code>localtime.realtime()</code>	Temporal	Returns the current <i>LocalTime</i> using the <code>realtime</code> clock.
<code>localtime.statement()</code>	Temporal	Returns the current <i>LocalTime</i> using the <code>statement</code> clock.
<code>localtime.transaction()</code>	Temporal	Returns the current <i>LocalTime</i> using the <code>transaction</code> clock.
<code>localtime.truncate()</code>	Temporal	Returns a <i>LocalTime</i> obtained by truncating a value at a specific component boundary. Truncation summary .
<code>log()</code>	Logarithmic	Returns the natural logarithm of a number.
<code>log10()</code>	Logarithmic	Returns the common logarithm (base 10) of a number.
<code>lTrim()</code>	String	Returns the original string with leading whitespace removed.
<code>max()</code>	Aggregating	Returns the maximum value in a set of values.
<code>min()</code>	Aggregating	Returns the minimum value in a set of values.
<code>nodes()</code>	List	Returns a list containing all the nodes in a path.
<code>none()</code>	Predicate	Returns true if the predicate holds for no element in a list.
<code>percentileCont()</code>	Aggregating	Returns the percentile of the given value over a group using linear interpolation.
<code>percentileDisc()</code>	Aggregating	Returns the nearest value to the given percentile over a group using a rounding method.
<code>pi()</code>	Trigonometric	Returns the mathematical constant <i>pi</i> .
<code>point() - Cartesian 2D</code>	Spatial	Returns a 2D point object, given two coordinate values in the Cartesian coordinate system.

Function	Category	Description
point() - Cartesian 3D	Spatial	Returns a 3D point object, given three coordinate values in the Cartesian coordinate system.
point() - WGS 84 2D	Spatial	Returns a 2D point object, given two coordinate values in the WGS 84 coordinate system.
point() - WGS 84 3D	Spatial	Returns a 3D point object, given three coordinate values in the WGS 84 coordinate system.
properties()	Scalar	Returns a map containing all the properties of a node or relationship.
radians()	Trigonometric	Converts degrees to radians.
rand()	Numeric	Returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e. [0, 1).
randomUUID()	Scalar	Returns a string value corresponding to a randomly-generated UUID.
range()	List	Returns a list comprising all integer values within a specified range.
reduce()	List	Runs an expression against individual elements of a list, storing the result of the expression in an accumulator.
relationships()	List	Returns a list containing all the relationships in a path.
replace()	String	Returns a string in which all occurrences of a specified string in the original string have been replaced by another (specified) string.
reverse()	List	Returns a list in which the order of all elements in the original list have been reversed.
reverse()	String	Returns a string in which the order of all characters in the original string have been reversed.
right()	String	Returns a string containing the specified number of rightmost characters of the original string.
round()	Numeric	Returns the value of a number rounded to the nearest integer.
rTrim()	String	Returns the original string with trailing whitespace removed.
sign()	Numeric	Returns the signum of a number: 0 if the number is 0, -1 for any negative number, and 1 for any positive number.
sin()	Trigonometric	Returns the sine of a number.
single()	Predicate	Returns true if the predicate holds for exactly one of the elements in a list.
size()	Scalar	Returns the number of items in a list.
size() applied to pattern expression	Scalar	Returns the number of sub-graphs matching the pattern expression.
size() applied to string	Scalar	Returns the number of Unicode characters in a string.

Function	Category	Description
<code>split()</code>	String	Returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.
<code>sqrt()</code>	Logarithmic	Returns the square root of a number.
<code>startNode()</code>	Scalar	Returns the start node of a relationship.
<code>stDev()</code>	Aggregating	Returns the standard deviation for the given value over a group for a sample of a population.
<code>stDevP()</code>	Aggregating	Returns the standard deviation for the given value over a group for an entire population.
<code>substring()</code>	String	Returns a substring of the original string, beginning with a 0-based index start and length.
<code>sum()</code>	Aggregating	Returns the sum of a set of numeric values.
<code>tail()</code>	List	Returns all but the first element in a list.
<code>tan()</code>	Trigonometric	Returns the tangent of a number.
<code>time()</code>	Temporal	Returns the current <i>Time</i> .
<code>time({hour [, minute, ...]})</code>	Temporal	Returns a <i>Time</i> with the specified component values.
<code>time(string)</code>	Temporal	Returns a <i>Time</i> by parsing a string.
<code>time({time [, hour, ..., timezone]})</code>	Temporal	Returns a <i>Time</i> from a map of another temporal value's components.
<code>time.realtime()</code>	Temporal	Returns the current <i>Time</i> using the <i>realtime</i> clock.
<code>time.statement()</code>	Temporal	Returns the current <i>Time</i> using the <i>statement</i> clock.
<code>time.transaction()</code>	Temporal	Returns the current <i>Time</i> using the <i>transaction</i> clock.
<code>time.truncate()</code>	Temporal	Returns a <i>Time</i> obtained by truncating a value at a specific component boundary. Truncation summary .
<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 a string value to a boolean value.
<code>toFloat()</code>	Scalar	Converts an integer or string value to a floating point number.
<code>toInteger()</code>	Scalar	Converts a floating point or string value to an integer value.
<code>toLower()</code>	String	Returns the original string in lowercase.
<code>toString()</code>	String	Converts an integer, float, boolean or temporal (i.e. Date, Time, LocalTime, DateTime, LocalDateTime or Duration) value to a string.
<code>toUpper()</code>	String	Returns the original string in uppercase.

Function	Category	Description
<code>trim()</code>	String	Returns the original string with leading and trailing whitespace removed.
<code>type()</code>	Scalar	Returns the string representation of the relationship type.

13.4. Expressions

Name	Description
<code>CASE Expression</code>	A generic conditional expression, similar to if/else statements available in other languages.

13.5. Cypher query options

Name	Type	Description
<code>CYPHER \$version query</code>	Version	This will force ' <code>query</code> ' to use Neo4j Cypher <code>\$version</code> . The default is 3.3 .
<code>CYPHER planner=rule query</code>	Planner	This will force ' <code>query</code> ' to use the rule planner. As the rule planner was removed in 3.2, doing this will cause ' <code>query</code> ' to fall back to using Cypher 3.1.
<code>CYPHER planner=cost query</code>	Planner	Neo4j 3.4 uses the cost planner for all queries.
<code>CYPHER runtime=interpreted query</code>	Runtime	This will force the query execution engine to use the interpreted runtime. This is the only option in Neo4j Community Edition.
<code>CYPHER runtime=slotted query</code>	Runtime	This will cause the query execution engine to use the slotted runtime. This is only available in Neo4j Enterprise Edition.
<code>CYPHER runtime=compiled query</code>	Runtime	This will cause the query execution engine to use the compiled runtime if it supports ' <code>query</code> '. This is only available in Neo4j Enterprise Edition.

Drivers

This chapter contains the complete documentation of the official Neo4j drivers.

Driver version: 1.6.

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 14. 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.

14.1. About the official drivers

Both official and community database drivers are available to provide access to Neo4j for applications. Official drivers exist for:

- C# — works with any .NET language
- Java — works with any JVM language
- JavaScript
- Python

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.



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](#).

14.2. Driver versions and installation

The 1.x series drivers have been built for Neo4j 3.x. The driver major version number correlates with the [Bolt protocol](#) version, the minor version number describes the driver feature set and the patch number defines the patch level of the driver in the regular sense. While minor versions of drivers will generally be released at the same time across languages, patch levels may vary.

It is recommended to always use the latest driver release within a major series. This will ensure that all server functionality is made available to client applications. To install a driver or to find out more about which driver versions are available, use the relevant language distribution system.

Example 1. Acquire the driver

Use `npm` to find out the latest version of the driver:

```
npm show neo4j-driver@* version
```

To install the latest version of the driver:

```
npm install neo4j-driver
```

You can also choose to install a certain version of the driver.

Below is the syntax for installing a certain version of the driver.

```
npm install neo4j-driver@$JAVASCRIPT-DRIVER-VERSION
```

In the following example we are installing driver version 1.6.1.

```
npm install neo4j-driver@1.6.1
```

You can review the release notes for this driver [here](https://github.com/neo4j/neo4j-javascript-driver/releases) (<https://github.com/neo4j/neo4j-javascript-driver/releases>).

14.3. Supported language and framework versions

For each language/framework, there is a set of versions that are officially supported by Neo4j.

Table 375. Supported languages/frameworks for the 1.x driver series

Language/framework	Versions supported
Java	Oracle JDK 7/8 and OpenJDK 7/8 (latest patch releases)
Python	CPython 2.7, 3.4, 3.5 and 3.6 (latest patch releases)
JavaScript	our drivers are built to work with all LTS versions of Node.js, specifically the 4.x and 6.x series runtimes (see https://github.com/nodejs/LTS).
.NET	our drivers target the .NET standard 1.3 (see https://github.com/dotnet/standard/blob/master/docs/versions.md).

14.4. A "Hello World" example

The example below shows the minimal configuration necessary to interact with Neo4j through a driver.

Example 4. Hello World

```
const driver = neo4j.driver(uri, neo4j.auth.basic(user, password));
const session = driver.session();

const resultPromise = session.writeTransaction(tx => tx.run(
  'CREATE (a:Greeting) SET a.message = $message RETURN a.message + ", from node " + id(a)',
  {message: 'hello, world'}));

resultPromise.then(result => {
  session.close();

  const singleRecord = result.records[0];
  const greeting = singleRecord.get(0);

  console.log(greeting);

  // on application exit:
  driver.close();
});
```

14.5. Driver API docs

For a comprehensive listing of all driver functionality, refer to the API documentation for the specific language driver.

Example 5. API docs

<https://neo4j.com/docs/api/javascript-driver/1.6/>

Chapter 15. Client applications

This section describes how to manage database connections within an application.

15.1. 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 6. The driver lifecycle

```
const driver = neo4j.driver(uri, neo4j.auth.basic(user, password));

driver.onCompleted = () => {
    console.log('Driver created');
};

driver.onError = error => {
    console.log(error);
};

const session = driver.session();
session.run('CREATE (i:Item)').then(() => {
    session.close();

    // ... on application exit:
    driver.close();
});
```

15.2. 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 376. Available URI schemes

URI Scheme	Driver object type
bolt	Direct driver
bolt+routing	Routing driver

15.2.1. 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.

15.2.2. 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.

15.2.3. 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 Causal 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.

In the standard Neo4j configuration, routing contexts are defined on the server side by means of *server policies*. Thus the driver communicates the routing context to the cluster in the form of a server policy. It then obtains refined routing information back from the cluster, based on the server policy.

The address in the URI of a Routing driver with routing context must be that of a Core server.

Example 7. Configure a Routing driver with routing context

This example will assume that Neo4j has been configured for server policies as described in [Neo4j Operations Manual](#) [Load balancing for multi-data center systems](#). In particular, a server policy called `europe` has been defined. Additionally, we have a server `neo01.graph.example.com` to which we wish to direct the driver.

This URI will use the server policy `europe`:

```
bolt+routing://neo01.graph.example.com?policy=europe
```

Server-side configuration to enable Routing drivers with routing context



A prerequisite for using a Routing driver with routing context is that the Neo4j database is operated on a [Causal Cluster](#) with the [Multi-data center licensing](#) option enabled. Additionally, the routing contexts must be defined within the cluster as *routing policies*. For details on how to configure multi-data center routing policies for a Causal Cluster, please refer to [Operations Manual](#) [Causal Clustering](#).

15.3. 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.

15.3.1. 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 8. Basic authentication

```
const driver = neo4j.driver(uri, neo4j.auth.basic(user, password));
```



The basic authentication scheme can also be used to authenticate against an LDAP server.

15.3.2. 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 9. Kerberos authentication

```
const driver = neo4j.driver(uri, neo4j.auth.kerberos(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) (<https://neo4j.com/docs/add-on/kerberos/1.0>) installed.

15.3.3. Custom authentication

For advanced deployments, where a custom security provider has been built, the custom authentication helper can be used.

Example 10. Custom authentication

```
const driver = neo4j.driver(uri, neo4j.auth.custom(principal, credentials, realm, scheme, parameters));
```

15.4. Configuration

15.4.1. Encryption

TLS encryption is enabled for all connections by default. This can be disabled through configuration as follows:

Example 11. Unencrypted

```
const driver = neo4j.driver(uri, neo4j.auth.basic(user, password),
{
  encrypted: 'ENCRYPTION_OFF'
});
```

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.

15.4.2. 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 377. 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 12. Trust

```
const driver = neo4j.driver(uri, neo4j.auth.basic(user, password),
{
  encrypted: 'ENCRYPTION_ON',
  trust: 'TRUST_ALL_CERTIFICATES'
});
```

15.4.3. Connection pool management

The driver maintains a pool of connections. The pooled connections are reused by sessions and transactions to avoid the overhead added by establishing new connections for every query. The connection pool always starts up empty. New connections are created on demand by sessions and transactions. When a session or a transaction is done with its execution, the connection will be returned to the pool to be reused.

Application users can tune connection pool settings to configure the driver for different use cases based on client performance requirements and database resource consumption limits.

Detailed descriptions of connection pool settings available via driver configuration are listed below:

[MaxConnectionLifetime](#)

Pooled connections older than this threshold will be closed and removed from the pool. The actual removal happens during connection acquisition so that the new session returned is never backed by an old connection. Setting this option to a low value will cause a high connection churn and might result in a performance drop. It is recommended to pick a value smaller than the maximum lifetime exposed by the surrounding system infrastructure (such as operating system, router, load balancer, proxy and firewall). Negative values result in lifetime not being checked. Default value: 1h.

MaxConnectionPoolSize

This setting defines the maximum total number of connections allowed, per host, to be managed by the connection pool. In other words, for a direct driver, this sets the maximum number of connections towards a single database server. For a routing driver this sets the maximum amount of connections per cluster member. If a session or transaction tries to acquire a connection at a time when the pool size is at its full capacity, it must wait until a free connection is available in the pool or the request to acquire a new connection times out. The connection acquiring timeout is configured via [ConnectionAcquisitionTimeout](#). Default value: This is different for different drivers, but is a number in the order of 100.

ConnectionAcquisitionTimeout

This setting limits the amount of time a session or transaction can spend waiting for a free connection to appear in the pool before throwing an exception. The exception thrown in this case is [ClientException](#). Timeout only applies when connection pool is at its max capacity. Default value: 1m.

Example 13. Connection pool management

```
const driver = neo4j.driver(uri, neo4j.auth.basic(user, password),  
{  
    maxConnectionLifetime: 30*60*60,  
    maxConnectionPoolSize: 50,  
    connectionAcquisitionTimeout: 2*60  
});
```

15.4.4. Connection timeout

To configure the maximum time allowed to establish a connection, pass a duration value to the driver configuration. For example:

Example 14. Connection timeout

This feature is not available **in** the JavaScript driver.

15.4.5. Load balancing strategy

A routing driver contains a load balancer to route queries evenly among many cluster members. The built-in load balancer provides two strategies: [least-connected](#) and [round-robin](#). The [least-connected](#) strategy in general gives a better performance as it takes query execution time and server load into consideration when distributing queries among the cluster members. Default value: [least-connected](#).

Example 15. Load balancing strategy

```
const driver = neo4j.driver(uri, neo4j.auth.basic(user, password),  
{  
    loadBalancingStrategy: "least_connected"  
}  
);
```

15.4.6. 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 16. Max retry time

```
const maxRetryTimeMs = 15 * 1000; // 15 seconds  
const driver = neo4j.driver(uri, neo4j.auth.basic(user, password),  
{  
    maxTransactionRetryTime: maxRetryTimeMs  
}  
);
```

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.

15.5. 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 17. Service unavailable

```
const driver = neo4j.driver(uri, neo4j.auth.basic(user, password), {maxTransactionRetryTime:  
3000});  
const session = driver.session();  
  
const writeTxPromise = session.writeTransaction(tx => tx.run('CREATE (a:Item)'));  
  
writeTxPromise.catch(error => {  
    if (error.code === neo4j.error.SERVICE_UNAVAILABLE) {  
        console.log('Unable to create node: ' + error.code);  
    }  
});
```

Chapter 16. Sessions and transactions

This section describes how to create units of work and provide a logical context for that work.

16.1. 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 18. Session

```
const session = driver.session();

session.run('CREATE (a:Person {name: $name})', { 'name': personName}).then(() => {
    session.close(() => {
        console.log('Person created, session closed');
    });
});
```

16.2. 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 and types](#).

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.

16.2.1. 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 19. Auto-commit transaction

```
function addPerson(name) {
  const session = driver.session();
  return session.run('CREATE (a:Person {name: $name})', {name: name}).then(result => {
    session.close();
    return result;
  });
}
```

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.

16.2.2. 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 20. Transaction function

```
const session = driver.session();
const writeTxPromise = session.writeTransaction(tx => tx.run('CREATE (a:Person {name: $name})',
{'name': personName}));

writeTxPromise.then(result => {
  session.close();

  if (result) {
    console.log('Person created');
  }
});
```

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.

16.2.3. 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.

16.2.4. 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 378. 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.

16.3. Causal chaining

When working with a causal cluster, transactions can be chained to ensure causal consistency. 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 carried out on different physical cluster members.



Due to protocol limitations, [auto-commit transactions](#) cannot currently take part in the causal chain. While this is expected to change in a future protocol release, `session.run` calls should be avoided in places where causal consistency is important.

Causal chaining is carried out by passing [bookmarks](#) between transactions. Each bookmark records a point in transactional history and can be used to inform cluster members to carry out units of work in a particular sequence. Internally, a bookmark is passed from server to client on a successful COMMIT and back from client to server on BEGIN. On receipt of one or more bookmarks, the transaction server will block until it has fast forwarded to catch up with the latest of these.

Within a session, bookmark propagation is carried out automatically and does not require any explicit signal or setting from the application. To opt out of this mechanism for unrelated units of work, applications can use multiple sessions. This avoids the small latency overhead of the causal chain. Propagation between sessions can be achieved by extracting the last bookmarks from one or more sessions and passing these into the construction of another. This is generally the only case in which an application will need to work with bookmarks directly.

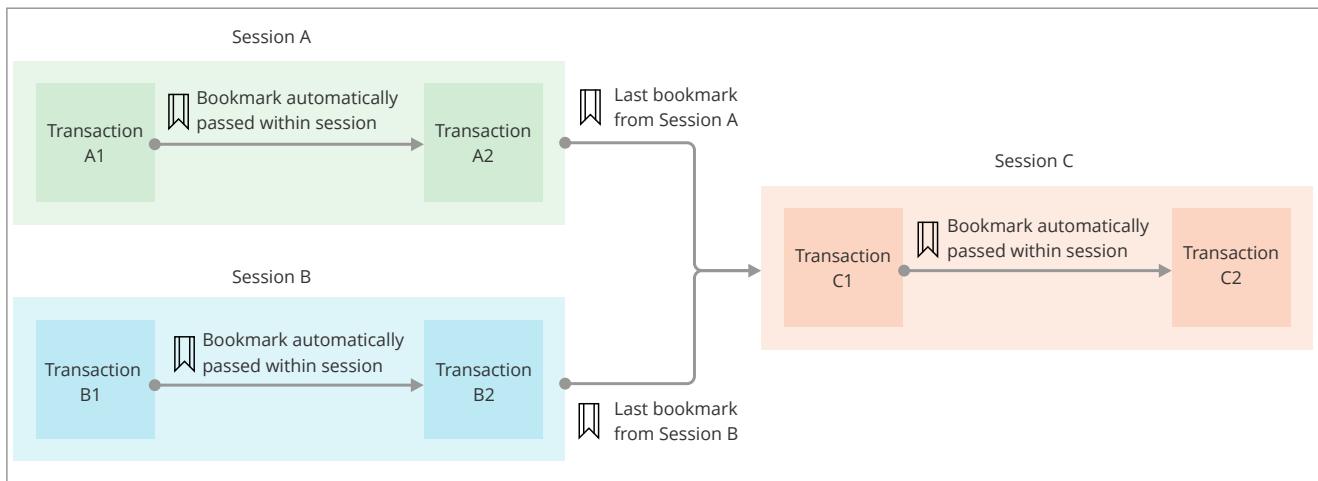


Figure 29. Passing bookmarks

Example 21. Passing bookmarks between sessions

This example illustrates the passing of bookmarks between sessions.

We are using three separate sessions: a, b and c. In *session a* we run two separate transactions. In the first one we create the person [Alice](#), and in the second one we record that she works at [Wayne Enterprises](#). The bookmark being passed between the two transactions is handled by the session. The bookmark from the last transaction is saved into an array for future use.

In *session b* we also run two separate transactions. In the first one we create the person [Bob](#), and in the second one we record that he works at [LexCorp](#). Again, the bookmark being passed between the two transactions is handled by the session. The bookmark from the last transaction is saved into an array for future use.

In the last session, *session c*, we wish to create a friendship between Alice and Bob. This can only be done if both [Alice](#) and [Bob](#) have been created first. In order to ensure this, we pass the bookmarks from the last transactions in *session a* and *session b*, respectively.

```

// Create a company node
function addCompany(tx, name) {
  return tx.run('CREATE (a:Company {name: $name})', {'name': name});
}

// Create a person node
function addPerson(tx, name) {
  return tx.run('CREATE (a:Person {name: $name})', {'name': name});
}

// Create an employment relationship to a pre-existing company node.
// This relies on the person first having been created.
function addEmployee(tx, personName, companyName) {
  return tx.run('MATCH (person:Person {name: $personName}) ' +
    'MATCH (company:Company {name: $companyName}) ' +
    'CREATE (person)-[:WORKS_FOR]->(company)', {'personName': personName, 'companyName': companyName});
}

// Create a friendship between two people.
function makeFriends(tx, name1, name2) {
  return tx.run('MATCH (a:Person {name: $name1}) ' +
    'MATCH (b:Person {name: $name2}) ' +
    'MERGE (a)-[:KNOWS]->(b)', {'name1': name1, 'name2': name2});
}

// To collect friend relationships
const friends = [];

// Match and display all friendships.
function findFriendships(tx) {
  const result = tx.run('MATCH (a)-[:KNOWS]->(b) RETURN a.name, b.name');

  result.subscribe({
    onNext: record => {
      const name1 = record.get(0);
      const name2 = record.get(1);

      friends.push({'name1': name1, 'name2': name2});
    }
  });
}

// To collect the session bookmarks
const savedBookmarks = [];

// Create the first person and employment relationship.
const session1 = driver.session(neo4j.WRITE);
const first = session1.writeTransaction(tx => addCompany(tx, 'Wayne Enterprises')).then(
  () => session1.writeTransaction(tx => addPerson(tx, 'Alice'))).then(
  () => session1.writeTransaction(tx => addEmployee(tx, 'Alice', 'Wayne Enterprises'))).then(
  () => {
    savedBookmarks.push(session1.lastBookmark());

    return session1.close();
  });
}

// Create the second person and employment relationship.
const session2 = driver.session(neo4j.WRITE);
const second = session2.writeTransaction(tx => addCompany(tx, 'LexCorp')).then(
  () => session2.writeTransaction(tx => addPerson(tx, 'Bob'))).then(
  () => session2.writeTransaction(tx => addEmployee(tx, 'Bob', 'LexCorp'))).then(
  () => {
    savedBookmarks.push(session2.lastBookmark());

    return session2.close();
  });
}

// Create a friendship between the two people created above.
const last = Promise.all([first, second]).then(ignore => {
  const session3 = driver.session(neo4j.WRITE, savedBookmarks);

  return session3.writeTransaction(tx => makeFriends(tx, 'Alice', 'Bob')).then(
    () => session3.readTransaction(findFriendships).then(
      () => session3.close()
    )
  );
});

```



If you try to extract a bookmark from a database which is not running in Causal Cluster mode, you will receive a `null` result.

16.4. 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.

Access modes can be supplied in two ways: per transaction or per session. An access mode specified at session creation can be overridden by the access mode of a transaction within that session. In the general case, access mode should always be specified at transaction level, using [transaction functions](#). The session-level setting is only necessary for explicit and auto-commit 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 22. Read-write transaction

```
const session = driver.session();

const writeTxPromise = session.writeTransaction(tx => tx.run('CREATE (a:Person {name: $name})',
  {name: personName}));

writeTxPromise.then(() => {
  const readTxPromise = session.readTransaction(tx => tx.run('MATCH (a:Person {name: $name})
    RETURN id(a)', {name: personName}));

  readTxPromise.then(result => {
    session.close();

    const singleRecord = result.records[0];
    const createdNodeId = singleRecord.get(0);

    console.log('Matched created node with id: ' + createdNodeId);
  });
});
```

16.5. Asynchronous programming



Java, .NET and JavaScript all support asynchronous programming. The examples here highlight specifically how Java and .NET provide for this programming model alongside their blocking API.

In addition to the methods listed in the previous sections, there also exist several asynchronous methods which allow for better integration with applications written in an asynchronous style. Asynchronous methods are named as their synchronous counterparts but with an additional `async` prefix.

Example 23. Asynchronous programming examples

Chapter 17. Working with Cypher values

This section describes the types and values used by Cypher and how they map to native language types.

17.1. 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 how Cypher works with types and to understand how the Cypher types are mapped in the driver.

The table below shows the available data types. All can be potentially found in results although not all can be used as parameters.

Cypher Type	Parameter	Result
null*	□	□
List	□	□
Map	□	□
Boolean	□	□
Integer	□	□
Float	□	□
String	□	□
ByteArray	□	□
Date	□	□
Time	□	□
LocalTime	□	□
DateTime	□	□
LocalDateTime	□	□
Duration	□	□
Point	□	□
Node**		□
Relationship**		□
Path**		□

* The null marker is not a type but a placeholder for absence of value. For information on how to work with null in Cypher, please refer to [Working with null](#).

** Nodes, relationships and paths are passed in results as snapshots of the original graph entities. While the original entity IDs are included in these snapshots, no permanent link is retained back to the underlying server-side entities, which may be deleted or otherwise altered independently of the client copies. Graph structures may not be used as parameters because it depends on application context whether such a parameter would be passed by reference or by value, and Cypher provides no mechanism to denote this. Equivalent functionality is available by simply passing either the ID for pass-by-reference, or an extracted map of properties for pass-by-value.

The Neo4j driver maps Cypher types to and from native language types as depicted in the table below. Custom types (those not available in the language or standard library) are highlighted in **bold**.

Example 24. Map Neo4j types to native language types

Neo4j type	JavaScript type
null	null
List	Array
Map	Object
Boolean	Boolean
Integer	Integer *
Float	Number
String	String
ByteArray	Int8Array
Date	Date
Time	Time
LocalTime	LocalTime
DateTime	DateTime
LocalDateTime	LocalDateTime
Duration	Duration
Point	Point
Node	Node
Relationship	Relationship
Path	Path

* JavaScript has no native integer type so a custom type is provided. For convenience, this can be disabled through configuration so that the native Number type is used instead. Note that this can lead to a loss of precision.

17.2. 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.

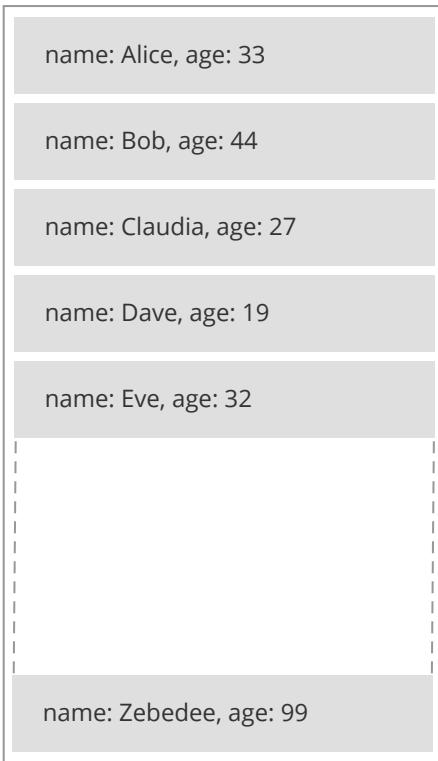


Figure 30. Result stream

17.2.1. 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).

17.2.2. The buffer

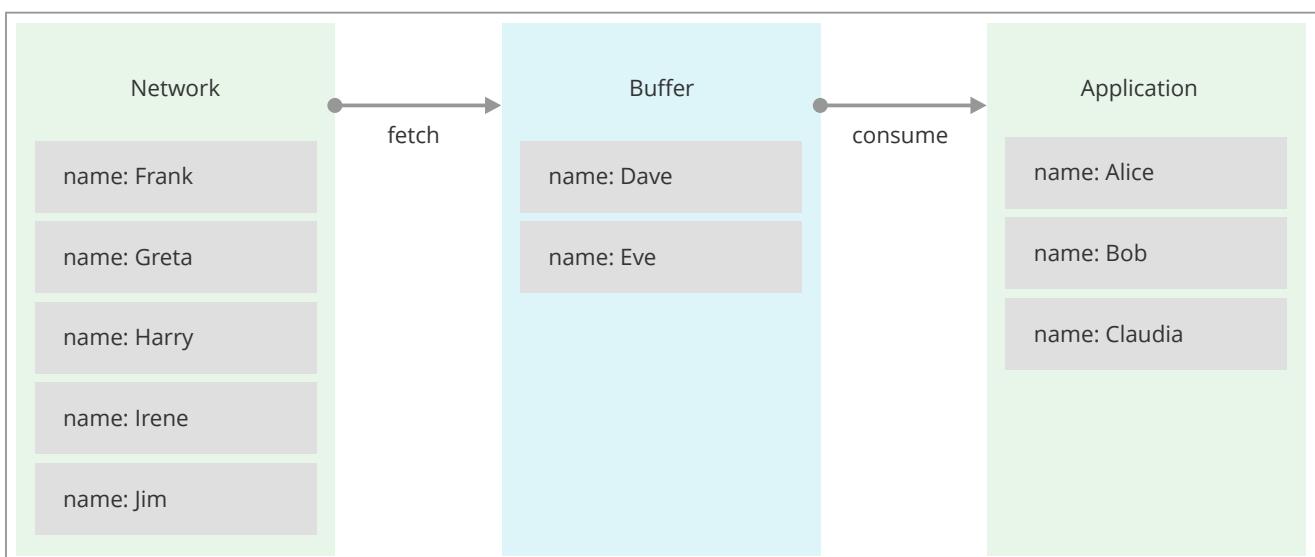


Figure 31. Result buffer

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.

17.2.3. 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 25. Consuming the stream

```
const session = driver.session();
const result = session.run('MATCH (a:Person) RETURN a.name ORDER BY a.name');
const collectedNames = [];

result.subscribe({
  onNext: record => {
    const name = record.get(0);
    collectedNames.push(name);
  },
  onCompleted: () => {
    session.close();

    console.log('Names: ' + collectedNames.join(', '));
  },
  onError: error => {
    console.log(error);
  }
});
```

17.2.4. 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 26. Retain results for further processing

```
const session = driver.session();

const readTxPromise = session.readTransaction(tx => tx.run('MATCH (a:Person) RETURN a.name AS name'));

const addEmployeesPromise = readTxPromise.then(result => {
  const nameRecords = result.records;

  let writeTxsPromise = Promise.resolve();
  for (let i = 0; i < nameRecords.length; i++) {
    const name = nameRecords[i].get('name');

    writeTxsPromise = writeTxsPromise.then(() =>
      session.writeTransaction(tx =>
        tx.run(
          'MATCH (emp:Person {name: $person_name}) ' +
          'MERGE (com:Company {name: $company_name}) ' +
          'MERGE (emp)-[:WORKS_FOR]->(com)' +
          '{' + 'person_name': name, 'company_name': companyName})));
  }

  return writeTxsPromise.then(() => nameRecords.length);
});

addEmployeesPromise.then(employeesCreated => {
  session.close();
  console.log('Created ' + employeesCreated + ' employees');
});
```

17.3. 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 18. 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.



18.1. 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.

18.2. 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" : [ ]  
}
```

18.3. 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" : [ ]
  }
}
```

18.4. 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" : "Wed, 20 Jun 2018 21:48:27 +0000"
    },
    "errors" : [ ]
  }
}
```

18.5. 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" : "Wed, 20 Jun 2018 21:48:27 +0000"
    },
    "errors" : [ ]
  }
}
```

18.6. 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" : "Wed, 20 Jun 2018 21:48:25 +0000"  
  },  
  "errors" : [ ]  
}
```

18.7. 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" : [ ]
}
```

18.8. 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" : [ ]
}
```

18.9. 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" : [ ]
}
```

18.10. 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" : [ ]
}

```

18.11. 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 ^"  
  } ]  
}
```

18.12. 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 19. 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](#)).

19.1. Authenticating

19.1.1. 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."  
  } ]  
}
```

19.1.2. 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"
}
```

19.1.3. 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."
  } ]
}
```

19.1.4. 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."
  } ]
}
```

19.2. User status and password changing

19.2.1. 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"  
}
```

19.2.2. 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"  
}
```

19.2.3. 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

19.3. Access when authentication and authorization are disabled

When authentication and authorization have been disabled, HTTP API requests can be sent without an [Authorization](#) header.

19.4. 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 20. 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.

20.1. 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](#).

20.2. 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.

20.3. 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.

20.3.1. 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>3.4.1</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>1.6.1</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>

```

20.3.2. 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().withoutEncryption()
        .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 ) );
        }
    }
}
```

20.3.3. 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`.
- The procedure annotation can take two arguments, `name` and `mode`.
 - `name` is used to specify a different name for the procedure than the default generated, which is `class.path.nameOfMethod`. If `mode` is specified then `name` must be specified as well.
 - `mode` is used to declare the types of interactions that the procedure will perform. The default mode is `READ`. The following modes are available:
 - `READ` This procedure will only perform read operations against the graph.

- **WRITE** This procedure will perform read and write operations against the graph.
 - **SCHEMA** This procedure will perform operations against the schema, i.e. create and drop indexes and constraints. A procedure with this mode is able to read graph data, but not write.
 - **DBMS** This procedure will perform system operations such as user management and query management. A procedure with this mode is not able to read or write graph data.
- The *context* of the procedure, which is the same as each resource that the procedure wants to use, is annotated `@Context`.
- The following can be noted about types:
- The *input* and *output* to and from a procedure must be one of the supported types, as described in [Values and types](#).
 - The Cypher types and their equivalents in Java are outlined in [the table below](#).
 - Composite types are also supported via:
 - `List<T>`, where `T` is one the supported types, and
 - `Map<String, Object>`, where the values in the map must have one of the supported types.
 - The use of `Object` is supported for the case where the type is not known beforehand. Note, however, that the actual value must still have one of the aforementioned types.

Table 379. Supported types

Cypher type	Java type
<code>String</code>	<code>String</code>
<code>Integer</code>	<code>Long</code>
<code>Float</code>	<code>Double</code>
<code>Boolean</code>	<code>Boolean</code>
<code>Point</code>	<code>org.neo4j.graphdb.spatial.Point</code>
<code>Date</code>	<code>java.time.LocalDate</code>
<code>Time</code>	<code>java.time.OffsetTime</code>
<code>LocalTime</code>	<code>java.time.LocalTime</code>
<code>DateTime</code>	<code>java.time.ZonedDateTime</code>
<code>LocalDateTime</code>	<code>java.time.LocalDateTime</code>
<code>Duration</code>	<code>java.time.temporal.TemporalAmount</code>
<code>Node</code>	<code>org.neo4j.graphdb.Node</code>
<code>Relationship</code>	<code>org.neo4j.graphdb.Relationship</code>
<code>Path</code>	<code>org.neo4j.graphdb.Path</code>

For more details, see the [API documentation for procedures](#) (<https://neo4j.com/docs/java-reference/3.4/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 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 of any type in this list, including
     *   {@link java.util.Map}</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 types above</li>
     *
     * @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: `%" , 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;
    }
}

```

20.3.4. 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 21. 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.

21.1. 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))
```

21.2. 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/3.4/javadocs/org/neo4j/procedure/UserFunction.html) (<https://neo4j.com/docs/java-reference/3.4/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);
    }
}
```

21.2.1. 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" ) );
        }
    }
}
```

21.3. 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.

21.3.1. 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)
```

21.3.2. 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/3.4/javadocs/org/neo4j/procedure/UserAggregationFunction.html) (<https://neo4j.com/docs/java-reference/3.4/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 22. 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-home> directory in case you want to load any custom settings from a file located there. 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/neo4j/neo4j-example-auth>

plugins

Appendix A: Reference

This appendix contains a complete reference of Neo4j status codes.

Chapter 23. 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.

23.1. 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

23.2. 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 an explicit 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.
Neo.ClientError.Request.InvalidUsage	The client made a request but did not consume outgoing buffers in a timely fashion.

Status Code	Description
Neo.ClientError.Request.TransactionRequired	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.
Neo.ClientError.Schema.ConstraintAlreadyExists	Unable to perform operation because it would clash with a pre-existing constraint.
Neo.ClientError.Schema.ConstraintNotFound	The request (directly or indirectly) referred to a constraint that does not exist.
Neo.ClientError.Schema.ConstraintValidationFailed	A constraint imposed by the database was violated.
Neo.ClientError.Schema.ConstraintVerificationFailed	Unable to create constraint because data that exists in the database violates it.
Neo.ClientError.Schema.ForbiddenOnConstraintIndex	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.
Neo.ClientError.Schema.IndexAlreadyExists	Unable to perform operation because it would clash with a pre-existing index.
Neo.ClientError.Schema.IndexNotApplicable	The request did not contain the properties required by the index.
Neo.ClientError.Schema.IndexNotFound	The request (directly or indirectly) referred to an index that does not exist.
Neo.ClientError.Schema.RepeatedPropertyInCompositeSchema	Unable to create composite index or constraint because a property was specified in several positions.
Neo.ClientError.Schema.TokenNameError	A token name, such as a label, relationship type or property key, used is not valid. Tokens cannot be empty strings and cannot be null.
Neo.ClientError.Security.AuthenticationRateLimit	The client has provided incorrect authentication details too many times in a row.
Neo.ClientError.Security.AuthorizationExpired	The stored authorization info has expired. Please reconnect.
Neo.ClientError.Security.CredentialsExpired	The credentials have expired and need to be updated.
Neo.ClientError.Security.EncryptionRequired	A TLS encrypted connection is required.
Neo.ClientError.Security.Forbidden	An attempt was made to perform an unauthorized action.
Neo.ClientError.Security.Unauthorized	The client is unauthorized due to authentication failure.
Neo.ClientError.Statement.ArgumentError	The statement is attempting to perform operations using invalid arguments
Neo.ClientError.Statement.ArithmeticError	Invalid use of arithmetic, such as dividing by zero.
Neo.ClientError.Statement.ConstraintVerificationFailed	A constraint imposed by the statement is violated by the data in the database.
Neo.ClientError.Statement.EntityNotFound	The statement refers to a non-existent entity.
Neo.ClientError.Statement.ExternalResourceFailed	Access to an external resource failed
Neo.ClientError.Statement.LabelNotFound	The statement is referring to a label that does not exist.
Neo.ClientError.Statement.ParameterMissing	The statement refers to a parameter that was not provided in the request.
Neo.ClientError.Statement.PropertyNotFound	The statement refers to a non-existent property.
Neo.ClientError.Statement.SemanticError	The statement is syntactically valid, but expresses something that the database cannot do.
Neo.ClientError.Statement.SyntaxException	The statement contains invalid or unsupported syntax.
Neo.ClientError.Statement.TypeError	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.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.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.ExperimentalFeature	This feature is experimental and should not be used in production systems.
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.
Neo.ClientNotification.Statement.PlannerUnsupportedWarning	This query is not supported by the COST planner.
Neo.ClientNotification.Statement.RuntimeUnsupportedWarning	This query is not supported by the chosen runtime.

Status Code	Description
Neo.ClientNotification.Statement.SuboptimalIndexForWildcardQuery	Index cannot execute wildcard query efficiently
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.Cluster.ReplicationFailure	Replication failure.
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.Request.NoThreadsAvailable	There are no available threads to serve this request at the moment. You can retry at a later time or consider increasing max thread pool size for bolt connector(s).
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	The transaction has been terminated, so no more locks can be acquired. This can occur because the transaction ran longer than the configured transaction timeout, or because a human operator manually terminated the transaction, or because the database is shutting down.
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 type. Supported types are:

- Number
- String
- Boolean
- Spatial types: Point
- Temporal types: Date, Time, LocalTime, DateTime, LocalDateTime and Duration
- Map types (plain maps, nodes, and relationships)
- Lists of any of the above

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 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 Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)

You are free to

Share

copy and redistribute the material in any medium or format

Adapt

remix, transform, and build upon the material

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.

NonCommercial

You may not use the material for commercial purposes.

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 <https://creativecommons.org/licenses/by-nc-sa/4.0/> for further details. The full license text is available at <https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode>.